Integrating SAT with MDG for Efficient Invariant Checking

Khaza Anuarul Hoque

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical & Computer Engineering)

at

Concordia University

Montréal, Québec, Canada

November 2010

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Khaza Anuarul Hoque

Entitled: "Integrating SAT with MDG for Efficient Invariant Checking"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. D. Qiu

_____ Examiner, External
Dr. J. Bentahar, CIISE To the Program

_____ Examiner
Dr. S. Hashtrudi Zad

_____ Examiner
Dr. M. Boukadoum

_____ Supervisor
Dr. O. Ait Mohamed

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20_____ _____
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

ABSTRACT

Integrating SAT with MDG for Efficient Invariant Checking

Khaza Anuarul Hoque

Multiway Decision Graph (MDG) is a canonical representation of a subset of many-sorted first-order logic. It generalizes the logic of equality with abstract types and uninterpreted function symbols. The area of Satisfiability (SAT) has been the subject of intensive research in recent years, with significant theoretical and practical contributions. From a practical perspective, a large number of very effective SAT solvers have recently been proposed, most of which based on improvements made to the original Davis-Putnam algorithm. Local search algorithms have allowed solving extremely large satisfiable instances of SAT. The combination between various verification methodologies will enhance the capabilities of each and overcome their limitations. In this thesis, we introduce a methodology and propose a new design verification tool integrating MDG and SAT, to check the safety of a design by invariant checking. Using MDG to encode the set of states provide powerful mean of abstraction. We use SAT solver searching for paths of reachable states violating the property under certain encoding constraints. In addition, we also introduce an automated conversion-verification methodology to convert a Directed Formula (DF) into Conjunctive Normal Form (CNF) formula that can be fed to a SAT solver. The formal verification of this conversion is conducted within the HOL theorem prover. Finally, we implement and conduct experiment on some examples along with a case study to show the correctness and the efficiency of our approach.

To My Family

ACKNOWLEDGEMENTS

At first, I would like to thank almighty Allah for his blessing to finish this research and for giving me opportunity for my higher studies.

It is a great pleasure for me to thank all those who have helped me in my research work. I would like to sincerely thank and express my gratitude to my supervisor, Dr. Otmane Ait Mohamed. This thesis would not have been possible without his guidance, his expert advice, his support and encouragements. Also, I sincerely thank Dr. Mounir Boukadoum, for co-supervising my research work. To all my fellow researchers and colleagues in Hardware Verification Group (HVG) at Concordia University, I thank you for your friendship, your thoughtful discussions and productive feedbacks. Most importantly, I thank the two most helpful people, Naeem Abbasi and Dr. Fariborz Fereydouni-Forouzandeh. They have always encouraged and motivated me whenever I was frustrated with the research progress. Also thanks to Dr.Sa'ed Abed for giving his valuable comments on my thesis. Without their encouragements and support, I would not have been able to continue my research work.

Last but not least, I thank my family for their constant moral support and their prayers. They are the people who are closest to me and suffered most for my higher study in abroad. Their support was invaluable in completing this thesis.

TABLE OF CONTENTS

LIST OF TABLES viii
LIST OF FIGURES ix
LIST OF ACRONYMS x

1 Introduction **1**
1.1 Motivation 1
1.2 Formal Verification Techniques 4
1.2.1 Theorem Proving 5
1.2.2 Equivalence Checking 6
1.2.3 Model Checking 6
1.3 Thesis Contribution 9
1.4 Thesis Outline 10

2 Related Work **12**
2.1 EUF Elimination 12
2.2 CNF Conversion 13
2.3 SAT and BDD Based Verification 14

3 Background **17**
3.1 Temporal Logic and Specification 17
3.1.1 Linear Time Logic 18
3.1.2 Computation Tree Logic 20
3.1.3 Full Branching-time Logic 21
3.1.4 Categories of Specification 22
3.2 Multiway Decision Graph 22
3.2.1 Abstract State Machine 22
3.2.2 Structure 23

3.3 The MDG-Tool . . . 27
3.3.1 MDGs Model Checking . . . 28
3.3.2 Invariant Specification in MDG . . . 29
3.4 Boolean Satisfiability . . . 30
3.5 The HOL Theorem Prover . . . 32
3.6 Normal Forms . . . 34

4 Integrating SAT with MDG 36
4.1 Formalization of the Problem . . . 36
4.2 Proposed Methodology . . . 37
4.2.1 Using MDG for Reachibility Analysis . . . 39
4.2.2 Preprocessing to Impose Boolean Encoding . . . 41
4.2.3 CNF Conversion of Directed Formula . . . 45
4.2.4 Verification of the Conversion . . . 47
4.2.5 Specification of Invariant Property and Correctness Formula . 50
4.2.6 Using SAT as a Verification Engine . . . 51

5 Implementation and Case Study 53
5.1 Conversion-Verification of Directed Formula . . . 53
5.1.1 Experiment Description . . . 53
5.1.2 Experimental Results . . . 54
5.2 Integrating SAT with MDG for Invariant Checking . . . 55
5.2.1 Case Study: Island Tunnel Controller (ITC) . . . 58
5.2.2 Implementation Description . . . 60
5.2.3 Experiment Results . . . 62

6 Conclusion and Future work 64

Bibliography 66

LIST OF TABLES

1.1 Deductive theorem proving vs. state exploration method 2
1.2 Raising the abstraction level 9

5.1 CNF conversion time 54
5.2 Invariant Checking Results with MDG tool and MDG-SAT approach 58
5.3 Total time for SAT-MDG approach 62
5.4 Invariant checking time: SAT-MDG and MDG tool 63

LIST OF FIGURES

1.1 Model-checking method 7

3.1 Model Structure 18
3.2 LTL formulae and time. 19
3.3 CTL formulae and time. 21
3.4 BDDs to MDGs. 24
3.5 MDG of an ALU 26
3.6 The Structure of the MDGs-tool 27
3.7 Invariant specification in MDG tool 30

4.1 Verification Methodology using MDG tool and SAT solver 38
4.2 MIN-MAX State Machine 43
4.3 Tesitin encoding to convert a propositional formula to CNF linearly . 47
4.4 Overview of the DF to CNF conversion-verification methodology . . . 48

5.1 DF size vs. CNF conversion time 55
5.2 An abstract counter 56
5.3 The Island Controller 59
5.4 The Island Controller 59

LIST OF ACRONYMS

ASM	Abstract State Machine
BDD	Binary Decision Diagram
CNF	Conjunctive Normal Form
SAT	Satisfiability Checking
HDL	Hardware Description Language
DNF	Disjunctive Normal Form
BMC	Bounded Model Checking
HOL	Higher Order Logic
CTL	Computation Tree Logic
DAG	Directed Acyclic Graph
DF	Directed Formula
FSM	Finite State Machine
LTL	Linear Temporal Logic
MDG	Multiway Decision Graph
RTL	Register Transfer Level
ITC	Island Tunnel Controller
RelP	Relational Product
PbyS	Pruning by Subsumption

Chapter 1

Introduction

1.1 Motivation

Faulty systems (bugs in digital systems) can be very dangerous and very expensive; especially for those which have safety-critical applications such as Magnetic Resonance Imaging (MRI) machines, space shuttles, microprocessors and so on. For example, bugs in Therac-25 machine caused 3 deaths and 3 serious injuries in 1985. In 1994, FDIV bug in Intel pentium processor caused them about $500 million USD followed by Mars polar lander loss in 1996 which cost NASA $165 million USD. There is a great advantage in being able to verify the correctness of such systems, whether they are hardware, software, or a combination. In the case of safety-critical systems, this is most obvious, but also applies to those that are commercially-critical, such as mass-produced chips. Formal verification methods have quite recently become usable by industry and there is a growing demand for professionals able to apply them [58]. Detection of bugs in design involves extra effort, time and cost. The overhead is even worse if the bug is detected late in the design process increasing the overall cost of the chip as well. The traditional debugging technique is simulation. However, due to the increasing size and complexity of VLSI circuits, it is impossible to simulate large designs properly. To overcome these limitations, formal

verification comes into play as a complement to simulation to detect errors in the design as early as possible.

Formal verification techniques have origins in the field of applied mathematics and have been successfully used in the past to prove that the implementation of a design meets its specification. These techniques can be categorized into two main groups: (1) state exploration based techniques [51], namely model checking and equivalence checking, and (2) deductive reasoning based techniques, namely theorem proving. Both of these techniques have their own strengths and weaknesses. Equivalence and model checkers are automatic tools and can be used by an engineer with no special knowledge about formal methods. These techniques, however, suffer from the state space explosion problem [36]. On the other hand, verification using theorem proving with higher-order-logics is not an automatic technique but can be applied to larger sized problems. Strengths and weakness of both, state exploration method and deductive theorem proving are summarized in Table 1.1 [11].

Table 1.1: Deductive theorem proving vs. state exploration method

Method	State exploration method	Deductive method
Automation	completely automatic	interactive
Domain size	finite system (large)	infinite system (complex)
Debugging	generates counter-example	expert based

Multiway Decision Graphs (MDGs) [25], are a special kind of decision diagrams that subsumes Binary Decision Diagrams (BDDs) and extends them by canonically and compactly representing a subset of first-order functions. The MDG system is a decision diagram-based verification tool, primarily designed for hardware verification. MDG tool supports both equivalence checking and model checking. With MDGs, a data value is represented by a single variable of an abstract type and operations on data are represented in terms of an uninterpreted functions. MDGs consist

of a set of algorithms. The specification and the implementation of the design are described using Hardware Description Languages (HDL) and then translated into the decision diagrams via intermediate languages. The algorithms in the system are used to efficiently and automatically deal with the decision diagrams so as to obtain a correct result.

Satisfiability Checking (SAT)-based tools to perform various forms of model checking has achieved a lot of attention these days [29, 9], as they are less sensitive to the problem sizes and the state explosion problem of classical Binary Decision Diagram (BDD)-based [21] model checkers. Expressing transition relation using Conjunctive Normal Form (CNF) along with SAT is an alternative to decision graphs and BDD-based approach. Such an approach, performance wise, is less sensitive to the problem size. As a result, many researchers have developed methods for performing Bounded Model Checking (BMC) [40, 71] using SAT. The common theme in these works is to convert the problem of interest into a SAT problem, by figuring out an appropriate propositional Boolean formula, and to utilize other non-canonical representations of state sets. These methods exploit the ability of SAT solvers to find a single satisfying solution, when it exists. In the recent years the SAT solver technology has improved significantly and a number of sophisticated packages are now available. Some of the well known state-of-the-art SAT solvers include CHAFF [62], GRASP [59] and SATO [80]. Most of the model checking techniques, in their implementation, involve state set manipulations. The state set manipulation problem can be transformed into a SAT problem. SAT solvers, thus, have the potential of enormously boosting the speed and applicability of model checking techniques.

In [9], an overview of a very interesting methodology integrating SAT and MDG model checker tool, was presented by the authors with preliminary experimental results. They used a rewriting based SAT solver to prune the transition relation of the circuits to produce a smaller one that is fed to the MDG model checker. The basic idea was to use the SAT solver as a reduction engine within

the MDG model checker tool [25]. In our work, we propose a new methodology to build a verification tool for invariant checking. An *invariant* is a liner time property that is given by a condition ϕ for the states and requires that ϕ holds for all the states. For our proposed methodology we use SAT [33] as a verification engine for MDG models. MDG as data structure, for representing transition systems or set of states, provide a powerful means for abstraction in order to suit large models intended for model checking. As an alternative of using MDG as a stand alone tool for invariant checking, we explore the benefits of combining SAT and MDG in our proposed methodology- for a new verification tool.

1.2 Formal Verification Techniques

Formal verification is a technique to prove mathematically that an implementation satisfies a given set of specification. The implementation(description of the design) to be verified can be described at different level of abstraction which leads in to different verification methods. The class of the implementation system or circuit to be verified is another issue. For example different verification approaches may be required for combinational/sequential, synchronous/asynchronous, pipelined or non-pipelined circuits. The correctness of the system is determined with respect to properties derived from specification. In practice, implementation and specification both are needed to be modeled in the tool. Then formal verification algorithm of that tool is applied to check the correctness of the system. A counter example is generated to trace error/errors, if the verification fails.

Although formal verification techniques use mathematical reasoning to establish that an implementation meets the specification, such correctness proof cannot guarantee that a real device will never malfunction. An actual device may still show unintended behavior, even if the hardware design is proved correct using formal verification tool. Wrong specification can play a major role in this or defects in physical

fabrication can cause this problem too. In formal verification, a model of the design is verified rather than a real physical implementation. As a result, a fault in the modeling process can result in false negatives (design errors that do not exist). Formal verification techniques can be divided into 3 categories, namely:

- Theorem proving
- Equivalence checking
- Model checking

1.2.1 Theorem Proving

In theorem proving, both the implementation and specification is described in formal logic. The correctness is obtained by mathematically proving their relationship formed as a theorem [48]. The logic is characterized by a proof system which refers to a set of axioms and a set of inference rules. Inference rules are applied until the desired theorem is proven. HOL(Higher Order Logic), ISABELLE, PVS (Prototype Verification System) and ACL2 [57, 64, 68, 50] are some of the high performance theorem provers. Unfortunately, using theorem prover requires expertise. Using theorem prover for verification is a white box approach, which means user is expected to know the whole design. It is not fully automated and requires a large amount of time on the part of the user in developing specifications of each component and in guiding the theorem prover through a large set of lemmas. A theorem prover may sometimes provide an insight into the reasons why a proof failed but is unable to provide counter-example in case the proof fails. The higher-order logics used in theorem proving are expressive. As a result, they allow modeling and verification of very complex systems and circuits very easily. Because they require a considerable interactive effort, theorem provers have less practice in industry and are mainly used for safety critical applications.

1.2.2 Equivalence Checking

Equivalence checking is a method to prove that two designs, represented at two different levels of abstraction of the same system are functionally equivalent [48]. A possible common scenario of equivalence checking can be comparing a circuit's gate-netlist description with its RTL description. Equivalence checking is usually divided into two classes: *Combinational* equivalence checking and *Sequential* equivalence checking. In combinational equivalence checking, the circuits to be compared are converted into canonical representation of Boolean functions, usually BDDs [21] which are then structurally compared. MDG [38] in academia, Synopsys Formality and Cadence Conformal [6] in the industry are two examples of tools that offer combination equivalence checking. On the other hand, in sequential equivalence checking, the given two designs are represented using state-encoding and later on the equivalence is then established by building the product finite state machine followed by checking whether the output is invariant for any initial states of the product machine. Sequential equivalence checking can verify between RTL and behavioral models because it only considers the behaviors of the models while ignoring details of the implementation. However, state space explosion problem restricts it from checking large designs. MDG and VIS [7] are examples of tools which can perform sequential equivalence checking.

1.2.3 Model Checking

Model checking as a verification technique was developed independently by Clarke and Emerson [36] and by Quielle and Sifakis [66] in the early 1980s. In model checking, a system is modeled as a set of states together with a set of transitions between states that represents how the system behavior evolves from one state to another over time, in response to internal and external stimulus. Model checking tools allow automatic verification of properties expressed in some temporal logic. In

case the verification fails, a counter example is generated that helps in tracing bugs in the design.

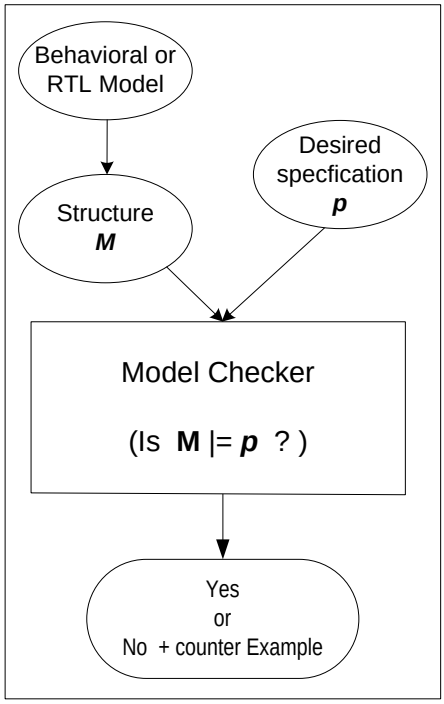

Figure 1.1: Model-checking method

Just like other verification techniques, model checking also has its own advantages and disadvantages. Model checker has two important advantages. First, model checking is fully automatic. Once the correct design of the system and the required properties has been fed in, it requires no further information or interaction from the user. Second, if the property fail to hold, the verification process is able to produce a counter-example (i.e. an instance of the behavior of the system that violates the property) which is extremely useful in helping the human designers pinpoint and fix the flaw. On the other hand, model checkers are unable to handle very large designs due to the state space explosion problem [36]. Another drawback is the problematic description of specifications as properties, this description sometimes may not give full system coverage.

SPIN [5], SMV [4], NuSMV [2], and MDG tool [77] are some of the commonly used model checking tools. They take as input, design description in a structural form along with system specifications or properties described in an appropriate temporal logic, and automatically check to see whether the system satisfies the property.

Since the model is usually a finite-state transition system, the problem of model checking is considered to be decidable. The design or model is formalized in terms of a state machine (Transition System), or a Kripke [49] structure:

$$M = (P, S, I, R, L)$$

where M is a state machine (model) with transitions to describe the circuit behavior, P is a set of atomic propositions, S is a finite set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation that must be total (i.e. for every $s \in S$ there exists $s' \in S$ such that $(s, s' \in R)$), and $L : S \rightarrow 2^P$ maps each state to the set of atomic propositions true in that state.

The property ϕ is formalized as a logical formula that the machine should satisfy. The verification problem is stated as checking the formula ϕ in the model M:

$$M \vDash \phi$$

If the model M is represented as a transition relation, then the size of the model is limited to the number of states that can be stored in the computer memory. This is about a few million states with the current state-of-the-art technology. Recently, the use of efficient state representations and manipulations using BDDs and/or SAT solving techniques has increased the size of problems that can be handled with model checking techniques by an order of magnitude.

An alternate approach based on a new class of decision graphs known as the Multiway Decision Graph (MDG) was proposed by Cerny et al. in 1997 [30]. This

approach is capable of dealing with the state-space explosion problem very effectively. In MDG based model checking, data signals are denoted by abstract variables, and data operators are represented by uninterpreted function symbols. As a result, a verification based on abstract-implicit-state-enumeration can be carried out independently of data path width. This results in a substantial reduction of the state space size. Table 1.2 shows [11] the abstraction level of MDG corresponding to traditional methods. A model-checking methodology typically consists of three major parts: a specification language, a system modeling language and a set of algorithms to perform model-checking. In existing MDG methodology, these are L_{MDG} [77], MDG-HDL [79] and MDG-tool [77, 78] respectively.

Table 1.2: Raising the abstraction level

Conventional method	Multiway Decision Graph
ROBDD [21]	MDG
Finite State Machine	Abstract State Machine
Implicit state enumeration [26]	Abstract state implicit enumeration of ASM
CTL based model-checking	Subset of first-order abstract CTL

1.3 Thesis Contribution

A desirable approach is to develop synergies between various verification methodologies, and between design and verification, in order to overcome the limitations and to enhance the capabilities of each and our work is motivated by this goal. Traditional invariant checking using MDG tool is the direct use of MDG reachability analysis algorithm [79]. We propose a new invariant checking methodology integrating SAT with MDG. SAT has already been integrated with MDG tool as a *reduction engine* [9]. In our work, we integrate SAT as a *verification engine*, to enhance the performance of invariant checking. Our study and implementation of

the methodology suggest a new verification tool combining the strengths of both: MDG and SAT. In terms of reviews of related work, proposed methodology and discussions, we believe our contribution can be specified as:

- We proposed a SAT based invariant checking methodology. For the completeness of the methodology:
  1. We implemented a *Preprocessor*, an automated *uninterpreted function* removal method to impose Boolean encoding on *Directed Formula (DF)* with adequate encoding constraints.
  2. An *Encoder* is also implemented to apply a linear CNF conversion algorithm on a Boolean Directed Formula. The *Encoder* also generates a *correctness formula* to be fed to a SAT solver.
- We proposed a technique to formally verify the correctness of CNF conversion *Directed Formula* and also implemented a *goal generator* that takes the input from the *Preprocessor* and generates a goal. Later on, an automated call to the HOL [57] theorem prover is placed to check the goal.

1.4 Thesis Outline

The rest of the thesis is organized as bellow:

- In Chapter 2, we present some of the related work in the area of *Equality with Uninterpreted function* removal, different CNF conversion algorithms and also some SAT based Verification approaches.
- In Chapter 3, we review the basics of temporal logic and specification, the structure of Multiway Decision Graph (MDG), MDG-tool followed by MDG model checking approach. Some basic review on Boolean SAT solvers, HOL theorem prover and normal forms concludes the chapter.

- In Chapter 4, we propose our methodology to integrate SAT with MDG and provide step by step description of the methodology. Also we present the methodology to formally verify the correctness of the CNF conversion algorithm.

- In Chapter 5, we present some experimental results showing the correctness of our conversion-verification approach. Also we present the case study, to show the efficiency of our proposed SAT-MDG invariant checking methodology.

- We conclude the thesis in Chapter 6 and provide some future research directions.

Chapter 2

Related Work

In this Chapter, we present some related research work in the area of SAT based verification. We divide the related works in three different categories. The first category focuses on different techniques to translate the formulas in Equality with Uninterpreted Functions(EUF) to propositional logic. The second category describes several algorithms for the conversion of propositional formula to CNF. In last and final category we discuss some related SAT based verification techniques.

2.1 EUF Elimination

Their exist two possible ways to eliminate EUFs [69], while enforcing their property of functional consistency, Ackermann constraints [12] and nested If-Then-Else operators (ITE) [67, 74]. In Ackermann's approach, the UF was replaced with a new term variable and the next application of UF with respect to the previous one was enforced by extending the resulting formula with constraints. Such constraints added for each pair of applications of that UF. Bryant and Velev presented an approach to eliminate the applications of UF with nested ITEs in [67]. In nested ITE scheme, the first application of the UF is still replaced by a new term variable. However the subsequent applications are eliminated by introducing nested ITEs with

new term variables while preserving functional consistency. We prefer nested ITE scheme which directly captures the functional consistency and readily exploit the maximal diversity property while Ackermann's can not [67]. For our methodology, we use the nested-ITE approach. However we add few small modifications to match the MDG directed formula syntax.

2.2 CNF Conversion

Lack of fast and efficient CNF generation algorithm has always been a bottle neck for CNF based SAT solvers. Hence researchers paid much attention to this point. Until recently [20], most of the CNF generation algorithms used in practice were minor variations of Tseitin linear time algorithm [73]. Another CNF conversion algorithm came from Velve [75] showing an efficient CNF generation technique with identifying gates with fan-out count of 1 and merging them with their fan-out gate to generate single set of equivalent CNF clauses. Nested ITE chains where each ITE is used only as else argument of the next ITE are similarly merged and represented with a single set of clauses without introducing intermediate variables. Such approach is good for pipelined machine verification problems, identifying certain patterns arising in formulas. Another approach for CNF generation is based on technological mapping [32] and its implemented in ABC [72]. This algorithm computes the mapping sequence, partial functions from And-Inverter-Graph (AIG)[16] nodes in order to cut of the graph for minimization of the heuristics cost function. CNF is then generated for the cuts of the nodes with respect to the final mapping by using their sum of products representation. Very recently an algorithm was presented [23] for converting Negation, ITE, Conjunction and Equivalence (NICE dags) to CNF. A new data structure called NICE dag subsume AIGs.

All the approaches described above use an intermediate representation or data structure for the boolean formula (either RBC, AIGER or NICE dag). The MDG

DF is itself a DAG, so intermediate DAG representation is not required to facilitate the conversion. The most interesting thing we observed is in most of the papers a "paper and pencil" sketch was given for the proof of their conversion approach. This motivated us to build an automated tool for the verification of conversion as well.

2.3 SAT and BDD Based Verification

Most of the efforts today are spent on developing Satisfiability Checking (SAT) based tools to perform several forms of model checking as they are less sensitive to the problem size and the state explosion problem of classical Binary Decision Diagram (BDD) based model checkers. As a result, many researchers have developed routines for performing Bounded Model Checking (BMC) [40][71] using SAT.

BDD and SAT based verification have been a major field of interest for researchers for a long time. Given that both techniques perform an implicit search in the underlaying Boolean space, it is no surprise that different approaches have been explored recently to combine both of them for target applications. Their benefits have been combined in many applications such as BMC[41, 8] and model checking [46]. In [47], the authors used BDDs to represent state sets, and a CNF formula to represent the transition relation. All valid next state combinations are enumerated using a backtracking search algorithm for SAT that exhaustively visits the entire space of primary input, present state and next state variables. However, rather than using SAT to enumerate each solution all the way down to a leaf, they invoked BDD-based image computation at intermediate points within the SAT decision procedure, which effectively obtains all solutions below that point in the search tree. In a sense, their approach can be regarded as SAT providing a disjunctive decomposition of the image computation into many subproblems, each of which is handled in the standard way using BDDs. Model checking techniques for security protocol analysis based on

reduction to Boolean logic has been explained in [15]. The main idea was, given a protocol description in multi-set formalism and an integer k, to build propositional formulas whose models correspond to attacks on the protocol. Propositional formulas are checked for satisfiability to indicate an attack on the protocol. For checking invariant properties of the form AG_p (p is globally true along all paths) of transition systems using Induction [29], Deharbe and Moreira modified a standard model checking algorithm. Set of states and image computation are expressed using BDD. Velev presented an indirect method to automatically prove the safety and liveness of a pipelined microprocessor. The term-level simulator TLSim [76], used for the symbolic simulation of the implementation and specification and a EUFM correctness formula is produced. The decision procedure EVC [76] exploits the positive equality, performs some other optimizations and converts the EUFM formula to an equivalent. An efficient SAT solver proves the formula to be a tautology in order for the implementation to be correct. In [70], A safety property checking technique of finite state machines using SAT solver was presented. Their approach demonstrates the practicality of combining a SAT-solver based safety property checking of in a real design flow using induction. All the works described above relies on BDD based state encoding,which suffers from state explosion for larger designs. In our case, We use MDG to encode the set of sates to get rid of the state explosion problem.

SAT and MDG integration was proposed in [9], while using SAT solver as a **reduction engine**. On the other hand, our proposed SAT based invariant checking methodology for MDG model, uses a SAT solver as a **verification engine**. Moreover, we implemented SAT encoding technique (CNF conversion) for MDG Direct formula (DF) and proposed another automated methodology to formally verify the correctness of the conversion. For the conversion part, we use Tseitin [73] approach while introducing "fresh variables" only for AND gates and for the verification part we use the HOL [57] theorem prover. Implementation of SAT for model checking with Multiway Decision Graph (MDG) distinguishes our approach from others.

In this Chapter, we presented some of the works related to uninterpreted function removal, different techniques of CNF conversion followed by some works related to SAT based verification. Also, we also mentioned how our work differs from those. In next Chapter, we present some preliminaries required to understand our contribution better.

Chapter 3

Background

In this chapter, we give a brief introduction to the Temporal Logic, Multiway Decision Graphs (MDGs) with MDG tool, followed by some basics of Boolean SAT solvers, the HOL theorem prover and normal forms. The intent is to familiarize the reader with the main concepts and notations that are used in the rest of the thesis. Section 3.1 presents temporal logic and how they are specified with *LTL, CTL* and *CTL**. Section 3.2 describes the underlying formal logic of MDGs, the Abstract State Machine (ASM) and the MDGs structure. Introduction to MDG tool, MDG model checker and invariant specification in MDG tool is described in Section 3.3. A brief introduction to SAT solver and SAT solving algorithm is described in Section 3.4. Section 3.5 starts by a basic description of higher-order logic concepts as well as the proof methods supported by the HOL theorem prover. We conclude the chapter in Section 3.6 describing the basic format of *Conjunctive Normal Form(CNF)* and *Disjunctive Normal Form(DNF)*.

3.1 Temporal Logic and Specification

Desired specifications in model checking methods, are usually written in propositional temporal logic formulae [65]. This allows the user to write propositions with

respect to time. The model of time is represented either in linear time (LTL) [55, 48] or branching time (CTL) [24]. A logic that combines the expressive power of LTL and CTL is known as CTL* which is also known as full branching-time logic.

3.1.1 Linear Time Logic

The structure of time in linear time logic, is a totally ordered set $(S, <)$, isomorphic to the set of natural numbers $(N, <)$ [31]:
If AP is a set of atomic propositions, a linear time structure is defined as $M(S, x, L)$, where:

- S is a set of states;
- $x : N \rightarrow S$ is an infinite sequences of states;
- $L : S \rightarrow 2^{AP}$ is a labeling of each state with the set of atomic propositions in AP at the state.

Following is an example (Fig. 3.1):

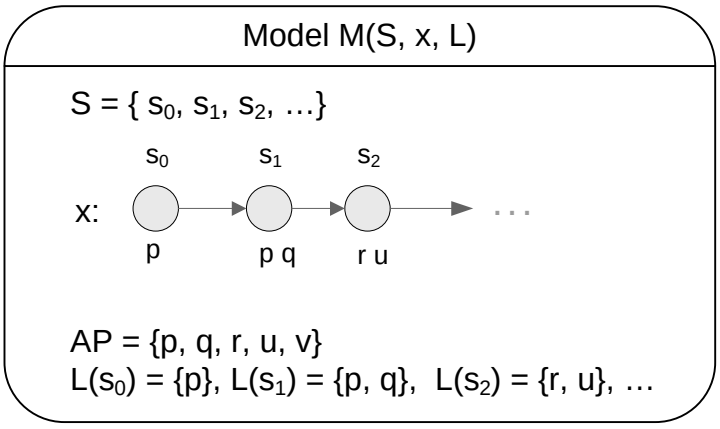

Figure 3.1: Model Structure

In propositional linear temporal logic (PLTL), one can use propositional logic as building block and apply temporal operators to specify properties. The PLTL

syntax is defined as a least set of formulae generated by the following rules [31]:

1. Each atomic proposition is a formula;
2. If p and q are formulae then $\neg p$ and $p \wedge q$ are formulae;
3. If p and q are formulae then $p\mathrm{U}q$ and $\mathrm{X}p$ are formulae.

The semantics of a formula p of PLTL with respect to a linear-time structure $M(S, x, L)$ is defined below. Here, we write $M, x \models p$ iff $p \in L(s_0)$ for atomic proposition p to mean that `in structure` M `formula` p `is true on timeline` x; x^i denotes the suffix path s_i, s_{i+1}, s_{i+2}, and so forth.

1. $M, x \models p$ `iff` $p \in L(s_0)$.
2. $M, x \models \neg p$ `iff` not $M, x \models p$.
3. $M, x \models \{p \wedge q\}$ `iff` $M, x \models p$ and $M, x \models q$.
4. $M, x \models \mathrm{X}p$ `iff` $M, x^1 \models p$.
5. $M, x \models p$ U q `iff` $\forall j(M, x^i \models q$, and $\forall_{0<i<j}(M, x^k \models p))$.
6. $M, x \models \mathrm{F}p$ `iff` $\exists j(M, x^j \models p)$.
7. $M, x \models \mathrm{G}p$ `iff` $\forall j(M, x^j \models p)$.

A PLTL formula p is satisfiable `iff` there exists a linear-time structure $M = (S, x, L)$ such that $M, x \models p$, and any such structure defines a model of p.
Examples are as following (Fig. 3.2):

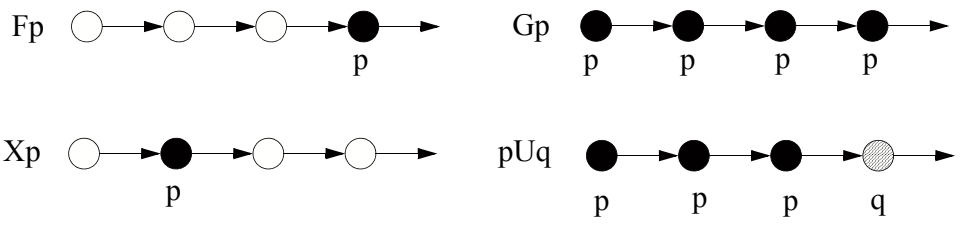

Figure 3.2: LTL formulae and time.

3.1.2 Computation Tree Logic

Computation tree logic (CTL) is based on branching time temporal logic (BTTL). It was first proposed by Clarke and Emerson [35]. In CTL, the time is modeled as a branching tree-like structure where each moment may have many different successor moments. Along each path, the timeline is isomorphic to the natural number. To specify a property in CTL, we simply apply the path operators along with temporal operators to the propositional building blocks. There are two strict restrictions in CTL:

1. Only single linear time operator F, G, X or U can follow a path quantifier;
2. Time operators cannot be combined directly with the propositional connectives.

The syntax of CTL is governed by the following rules:

1. Every proposition is a CTL formula;
2. If p and q are CTL formula, then so are p, $(p \wedge q)$, AXp, EXp, A(pUq), E(pUq).

The remaining operators can be derived from the above rules. The truth of a formula is determined on a given state and not on a branch of the time structure. The structure resembles an infinite computation tree. A temporal formula p is satisfied by a model M with transitions T, if it is true for all the initial states s_0 of the model. The semantics of CTL formula is given below:

1. $M, s_0 \models p$ iff $p \in L(s_0)$.
2. $M, s_0 \models \neg p$ iff not $M, s_0 \models p$.
3. $M, s_0 \models \{p \wedge q\}$ iff $M, s_0 \models p$ and $M, s_0 \models q$.
4. $M, s_0 \models$ AXp iff for all states $s_0^{'}$ with $(s_0, s_0^{'}) \in T$, $M, s_0^{'} \models p$.

5. $M, s_0 \models \mathrm{EX}p$ iff for some state s_0' with $(s_0, s_0') \in T$, $M, s_0' \models p$.

6. $M, s_0 \models \mathrm{A}(p\mathrm{U}q)$ iff for all paths $(s_0, s_1, \dots)$, there exists a $j > 0$ with $M, s_j \models q$, and $M, s_i \models p$ holds $\forall 0 \leq i < j$.

7. $M, s_0 \models \mathrm{E}(p\mathrm{U}q)$ iff for some path $(s_0, s_1, \dots)$, there exists a $j > 0$ with $M, s^j \models q$, and $M, s^i \models p$ holds $\forall 0 \leq i < j$.

Figure 3.3 shows intuitive meanings of some CTL formulae.

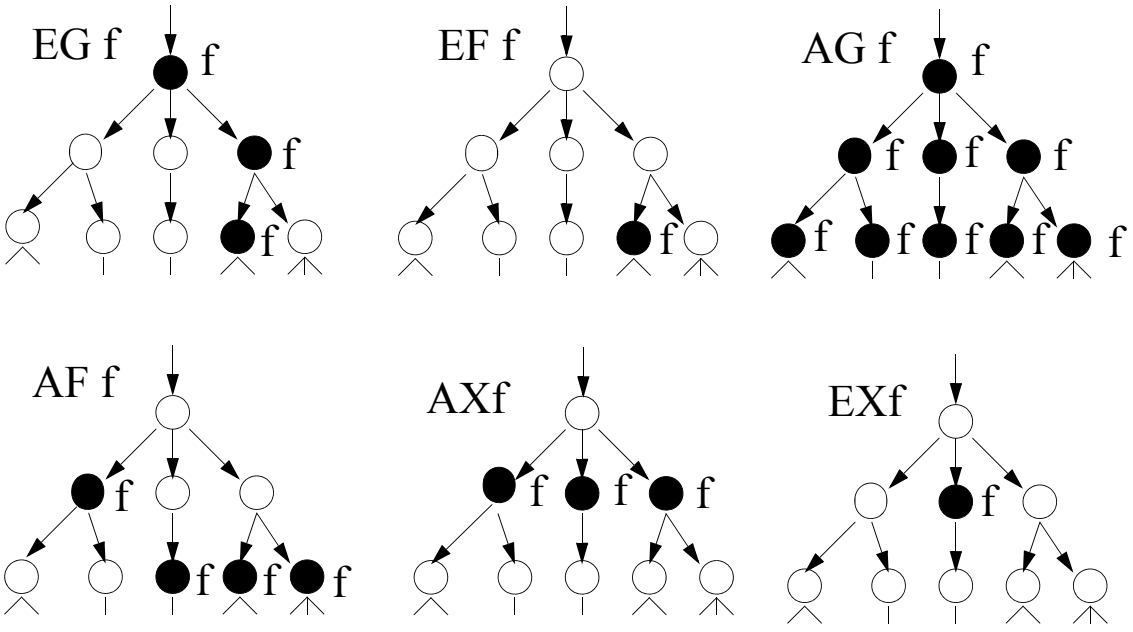

Figure 3.3: CTL formulae and time.

3.1.3 Full Branching-time Logic

Full branching-time logic is the class of logic formulas that combines the branching-time and linear-time operators. In CTL*, a path quantifier can be a prefix to an assertion composed of arbitrary combination of the temporal operators: F, G, X and U. Like CTL, the tree is formed by designating an initial state s_0 in model M, and

then unwinding the structure into an infinite tree with s_0 as the root. The semantics of the path quantifiers and temporal operators remain the same.

3.1.4 Categories of Specification

The specifications are written as properties of the system. They are categorized as follows:

1. Safety property – ensures that nothing 'bad' will ever happen. A temporal logic formula is a safety property if it can be written as AG_{φ}(in CTL or CTL*) or G_{ϕ}(in PLTL), where φ is a propositional frormula.

2. Liveness property – ensures that something 'good' will eventually happen. Depicted as $\models \mathrm{F}p$ or $\models \mathrm{AF}p$, where p will eventually be true at some point in the future.

3. Precedence property – ensures precedence order of events. Depicted as $\models p\mathrm{U}q$, where q is true in present time or p is true until q becomes true.

Among the three, safety property is the most used when writing specifications of a design under verification.

3.2 Multiway Decision Graph

3.2.1 Abstract State Machine

In MDG, a state machine is described using finite sets of input, state and output variables, which are pair-wise disjoint. The behavior of a state machine is defined by its transition/output relations including a set of reset states. An abstract description of the state machine, called Abstract State Machine (ASM) [37], is obtained by letting some data input, state or output variables be of an abstract sort, and

the datapath operations be uninterpreted function symbols. As ROBDDs are used to represent sets of states and transition/output relations for finite state machines (FSM), MDGs are used to compactly encode sets of (abstract) states and transition/output relations for ASMs. This technique replaces the *implicit enumeration technique*[27] with the *implicit abstract enumeration* [25].

3.2.2 Structure

MDGs are graph representation of a class of quantifier-free and negation-free first-order many sorted formulae. It subsumes the class of Bryant's (ROBDDs) [19] while accommodating abstract data and Uninterpreted Function symbols. MDG can be seen as a Directed Acyclic Graph (DAG) with one root, whose leaves are labeled by formulae of the logic True (T) [25], such that:

1. Every leaf node is labeled by the formula T, except if the graph G has a single node, which may be labeled Tor F.

2. The internal nodes are labeled by terms, and the edges issuing from an internal node v are labeled by terms of the same sort as the label of v.

Following is an example: Let graph G represent Boolean formula$(\neg x \wedge F_0) \vee (x \wedge F_1)$, Where, F_0 and F_1 are the Boolean formulas represented by the sub-graphs G_0 and G_1 respectively. In many sorted first-order logic the graph G can be viewed as representing a formula: $((x = 0) \wedge F_0) \vee (x = 1) \wedge F_1))$.
Three possible generalizations of G and the corresponding formulas are shown in (Fig. 3.4. F_0, F_1 and F_2 are first-order formulas represented by the sub-graphs G_0, G_1 and G_2 respectively:

1. From G to G': $x \in \{0, 1\} \rightarrow x \in \{0, 2, 3\}$, and Graph G' represents the formula

$$((x = 0) \wedge F_0) \vee ((x = 2) \wedge F_1) \vee ((x = 3) \wedge F_2).$$

2. From G to G'': $x \in \{0,1\} \rightarrow x \in \{a, y, f(a,y)\}$, and Graph G'' represents the formula

$$((x = a) \wedge F_0) \vee ((x = y) \wedge F_1) \vee ((x = f(a,y)) \wedge F_2).$$

3. From G to G''': $x \in \{0,1\} \rightarrow g(x) \in \{0,2,3\}$, and Graph G''' represents the formula

$$((g(x) = 0) \wedge F_0) \vee ((g(x) = 2) \wedge F_1) \vee ((g(x) = 3) \wedge F_2).$$

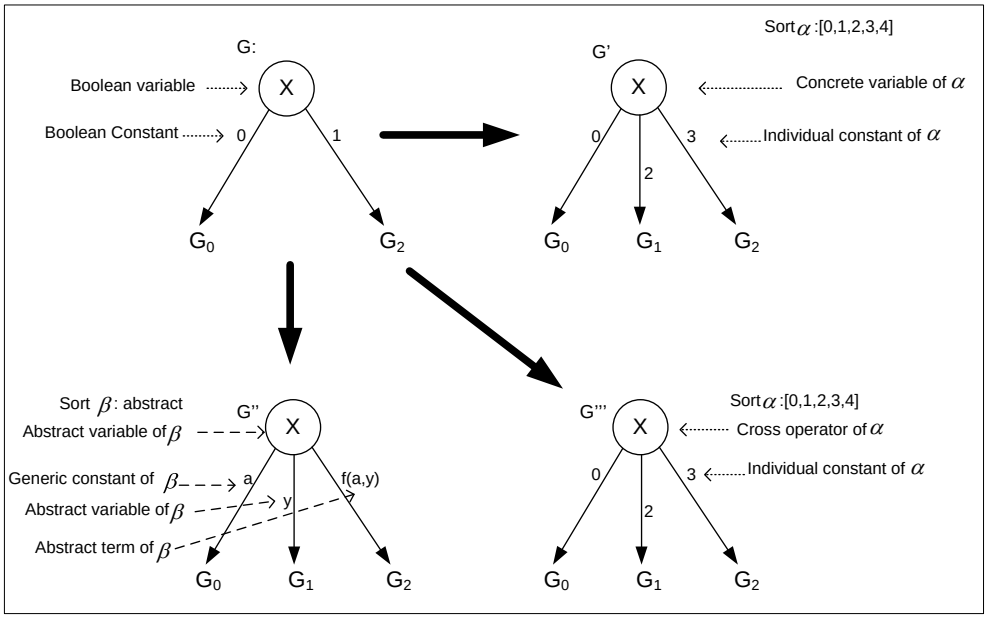

Figure 3.4: BDDs to MDGs.

The above generalized decision graph G', G'' and G''' are examples of Multiway Decision Graphs (MDGs).

As in ordinary many-sorted First Order Logic (FOL), terms are made out of sorts, constants, variables, and function symbols. Two kinds of sorts are distinguished: concrete and abstract. Concrete sort is equipped with finite enumerations, lists of individual constants. Concrete sorts are used to represent control signals.

Abstract sort has no enumeration available and represents a data signal. MDGs represent and manipulate a certain subset of first order formulae, which we call *Directed Formulae (DFs)*.

$$
\begin{array}{lll}
\text{Sort } S & ::= & S \mid \underline{S} \\
\text{Abstract Sort } S & ::= & \alpha \mid \beta \mid \gamma \mid \cdots \\
\text{Concrete Sort } \underline{S} & ::= & \underline{\alpha} \mid \underline{\beta} \mid \underline{\gamma} \mid \cdots \\
\text{Generic Constant } C & ::= & a \mid b \mid c \mid \cdots \\
\text{Concrete Constant } \underline{C} & ::= & \underline{a} \mid \underline{b} \mid \underline{c} \mid \cdots \\
\text{Variable } \mathcal{X} & ::= & V \mid \underline{V} \\
\text{Abstract Variable } V & ::= & x \mid y \mid z \mid \cdots \\
\text{Concrete Variable } \underline{V} & ::= & \underline{x} \mid \underline{y} \mid \underline{z} \mid \cdots \\
\text{Directed Formulae } DF & ::= & Disj \mid \top \mid \bot \\
Disj & ::= & Conj \vee Disj \mid Conj \\
Conj & ::= & Eq \wedge Conj \mid Eq \\
Eq & ::= & \underline{A} = \underline{C}(A \in \mathcal{T}(\mathcal{F}, V)) \\
 & & \mid \underline{V} = \underline{C} \\
 & & \mid V = A(A \in \mathcal{T}(\mathcal{F}, \mathcal{X}))
\end{array}
$$

Let $\mathcal{F}$ be a set of function symbols and $\mathcal{V}$ a set of variables. We denote the set of terms freely generated from $\mathcal{F}$ and $\mathcal{V}$ by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The syntax of a Directed Formula is given by the grammar given above. [13]. The underline is used to differentiate between the concrete and abstract variables.

The vocabulary consists of generic constants, concrete constants (individuals), abstract variables, concrete variables and function symbols. DFs are always disjunctions of conjunctions of equations or $\top$ (true) or $\bot$ (false). The conjunction *Conj* is defined to be an equation only (*Eq*) or a conjunction of at least two equations. Atomic formulae are the equations, generated by the clause *Eq*. An equation can be

an equality of concrete terms and an individual constant, equality of a concrete variable and an individual constant, or equality of an abstract variable and an abstract term.

A directed formula DF of type $U \rightarrow V$ is a formula in Disjunctive Normal Form (DNF) plus $\top$ (truth) and $\bot$ (false), where U and V are two disjoint sets of variables. Just as ROBDD must be *reduced* and *ordered*, MDGs must obey a set of well-formedness conditions given in [25]. DFs are used for two purposes: to represent sets (viz. sets of states as well as sets of input vectors and output vectors) and to represent relations (viz. the transition and output relations) as well as the set of possible initial states and the sets of states that arise during reachability analysis.

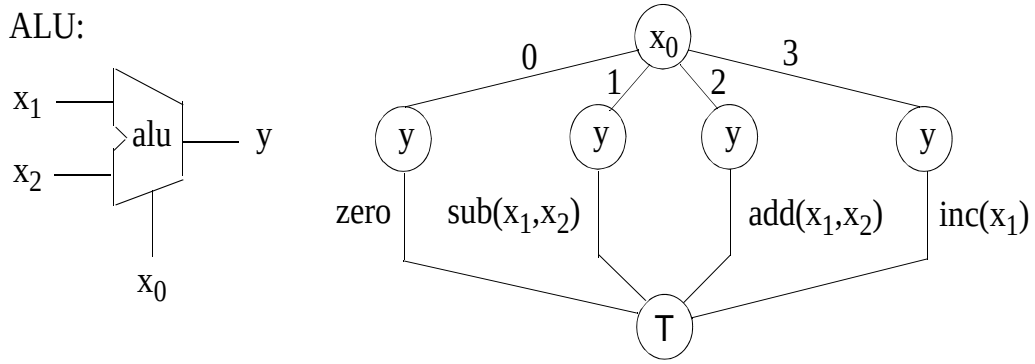

Figure 3.5: MDG of an ALU

Figure 3.5 shows MDG reforestation of a simple Arithmetic Logic Unit (ALU). Depending on the value of selection signal x_0 the output value is chosen between the addition or the subtraction of x_1 and x_2, increment of x_1 or the output value can be *zero* as well. The Directed Formula(DF) representation of the MDG is as following:

$[(x_0 = 0) \wedge y = zero] \vee$

$[(x_0 = 1) \wedge y = sub(x_1, x_2)] \vee$

$[(x_0 = 2) \wedge y = add(x_1, x_2)] \vee$

$[(x_0 = 3) \wedge y = inc(x_1)] \vee$

3.3 The MDG-Tool

The MDG-tool [79] is a well known academic tool. It supports invariant checking, sequential equivalence checking, and model checking. The MDGs tool uses a Prolog-style hardware description language called the MDG-HDL(*MDG-HDL*) [25]. MDG-HDL supports structural, behavioral and mixed styles of coding. A structural specification is usually a netlist of components connected by signals. A behavioral description consists of a tabular representation of the transition and output relations in the form of a truth table.

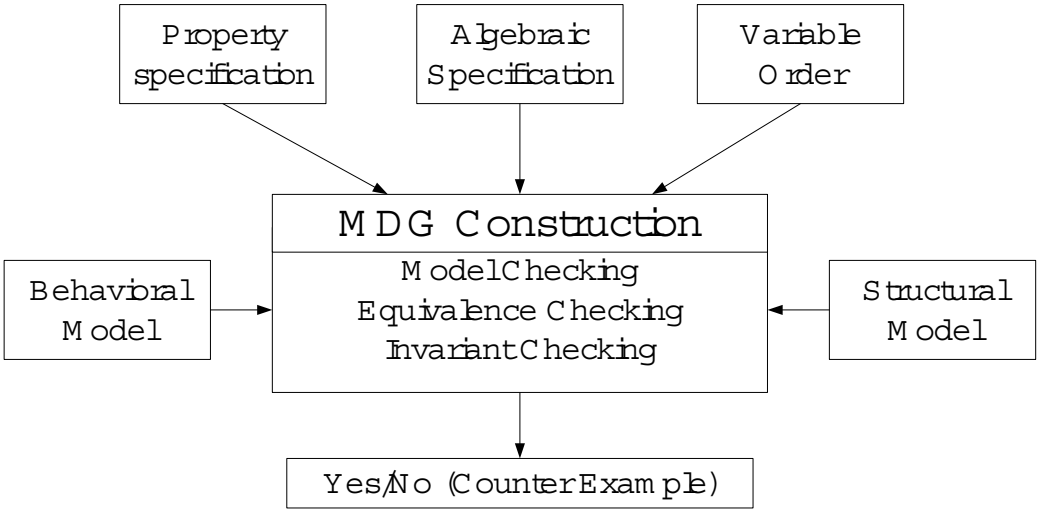

Figure 3.6: The Structure of the MDGs-tool

The first step in the verification is to describe the design specifications and implementations using MDG-HDL, as shown in Figure 3.6. An MDG-HDL algebraic specification consists of sorts, function types, and generic constants. Rewrite rules needed for interpreting function symbols are also provided. Symbol ordering (like for ROBDD) can either be specified by the user, or can be dynamically generated by the MGD tool. Symbol ordering can critically affect the size of the generated MDGs and the performance of the verification.

3.3.1 MDGs Model Checking

The MDGs model checking is based on an abstract implicit state enumeration. The circuit under verification is expressed as an Abstract State Machine (ASM) and the properties to be verified are expressed as formulae in $\mathcal{L}_{MDG}$ [77, 11]. The ASM description of the digital systems is at a higher level of abstraction. $\mathcal{L}_{MDG}$ atomic formulae are Boolean constants (True and False), or equations of the form $(t_1 = t_2)$, where t_1 is an ASM variable (input, output or state variable) and t_2 is either an ASM system variable, an individual constant, an ordinary variable or a function of ordinary variables. Ordinary variables are defined to memorize the values of the system variables in the current state. The basic $Next\_let\_formulas$ contain temporal operator **X** (next time) is defined follows [13]:

- Each atomic formula is a $Next\_let\_formula$;
- If p, q are $Next\_let\_formulas$, then so are: !p (not p), p&q (p and q), p|q (p or q), p $\rightarrow$ q (p implies q), **X**p (next-time p) and LET (v=t) IN p, where t is a system variable and v an ordinary variable.

Using the temporal operators **AG** (*always*), **AF** (*eventually*) and **AU** (*until*), the supported $\mathcal{L}_{MDG}$ properties are defined in the following BNF grammar:

$$
\begin{aligned}
\text{Property} ::= \; & A(Next\_let\_formula) \\
& | \, AG(Next\_let\_formula) \\
& | \, AF(Next\_let\_formula) \\
& | \, A(Next\_let\_formula) \, U \, (Next\_let\_formula) \\
& | \, AG(Next\_let\_formula) \Rightarrow F(Next\_let\_formula) \\
& | \, AG((Next\_let\_formula) \Rightarrow \\
& \qquad ((Next\_let\_formula) \, U \, Next\_let\_formula)))
\end{aligned}
$$

Model checking in MDGs is carried out by automatically building additional circuits that represent the Next let formulas appearing in the property to be verified,

composing it with the original circuit, and then checking a simpler property on the composite machine [77].

3.3.2 Invariant Specification in MDG

An invariant file specifies the invariant condition to be checked during reachability analysis[79]. An invariant condition can be specified by a combinational circuit whose output signals are named by the variables that occur in the condition. By convention, an assignment of values to those variables satisfies the condition iff the outputs of the combinational circuit take those values for some assignment of values to the inputs. An MDG representing the invariant is obtained from the MDG representing the functionality of the combinational circuit by existentially quantifying the concrete inputs. The variables representing abstract inputs are left in the graph as implicitly quantified secondary variables [82].

For example, for the equivalence checking of two ASMs, we need to specify the equality of two corresponding signals as the invariant. This is expressed by the simple fork as shown in Figure 3.7 (a). The fork may yield different MDGs depending on the sort of the signals. If u, x and y are of the Boolean sort, then u is existentially quantified and we get the MDG as shown in Figure 3.7 (b) which simply represents $x = y$. If x and y are of an abstract sort, then we get an MDG as shown in Figure 3.7 (c) which represents the formula $(x = u) \wedge (y = u)$. Taking the secondary variable u to be existentially quantified, the invariant is $\exists u((x = u) \wedge (y = u))$, which is logically equivalent to $x = y$.

This combinational circuit is described completely in this invariant specification file, including the following predicates: $signal/2$, $component/2$, $outputs/1$ and $order\_cond/1$ which gives the node order for the variables and the cross-function symbols appeared in the circuit.

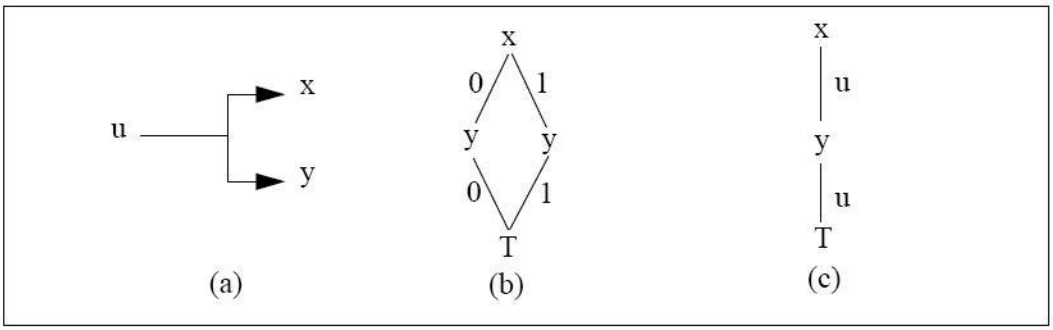

Figure 3.7: Invariant specification in MDG tool

3.4 Boolean Satisfiability

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem. It has many applications in the field of computer aided design such as test generation, logic verification and timing analysis. Given a Boolean formula, the objective is to either find a boolean assignment to the variables so that the formula evaluates to true, or establish that such an assignment does not exist. The Boolean formula is typically expressed in Conjunctive Normal Form (CNF), also called product-of-sums form. Each sum term (clause) in the CNF is a sum of single literals, where a literal is a variable or its negation.

Most state-of-the-art SAT solvers are based on the Davis-Putnam algorithm [28]. The basic algorithm begins from an empty assignment, and proceeds by assigning a 0 or 1 value to a free variable at a time. After each assignment, the algorithm determines the direct and transitive implications of that assignment on other variables. This process is sometimes also called the Boolean Constraint Propagation (BCP). If no contradiction is detected during the implication procedure, the algorithm picks the next free variable, and this process is repeated. Otherwise, the algorithm attempts a new partial assignment by complementing the most recently assigned variable for which only one value has been tried so far. This step is called backtracking. The algorithm terminates either when all clauses have been satisfied

and a solution has been found, or when all possible assignments have been exhausted. The algorithm is complete in that it will find a solution if it exists.

Algorithm 1 Davis_Putnam()

```
1: while true do
2:    if DecideNextBranch()==false then
3:       return(SATISFIABLE);
4:    end if
5:    while Deduce()==CONFLICT do
6:       if ResolveConflict()==false then
7:          RETURN(UNSATISFIABLE);
8:       end if
9:    end while
10: end while
```

Algorithm 1 shows the Pseudo code adapted from the basic Davis-Putnam search procedure. The function `DecideNextBranch()` selects a variable that is not currently assigned, and gives it a value. This variable assignment is referred to as a decision. The `Deduce()` function carries out Boolean Constraint Propagation (BCP). It propagates variable assignments based on the current decision. Basically, if a clause consists of only literals with value 0 and one unassigned literal, then that unassigned literal must take on a value of 1. Clauses in this state are said to be unit, and this rule is referred to as the unit clause rule. In the pseudo-code, `Deduce()` carries out BCP transitively until either there are no more implications (in which case it returns `SATISFIABLE`) or a conflict is produced (in which case it returns `UNSATISFIABLE`). A conflict occurs when implications for setting the same variable to both 1 and 0 are produced.

Algorithm 2 ResolveConflict()

```
1: d = most recent decision not tried both ways;
2: if d == NULL then
3:    return(false);
4: end if
5: flip the value of d;
6: mark d as tried both ways;
7: undo any invalidated implications;
8: return(true);
```

In order to deal with a conflict, we can flip the value of the decision assignment, undo all the implications by the decision, and allow BCP to then proceed as normal. If both values have already been tried for this decision, then we backtrack through the decision stack until we encounter a decision that has not been tried both ways, and proceed from there in the manner described in Algorithm 2.

3.5 The HOL Theorem Prover

The HOL system is an LCF [45, 11] (Logic of Computable Functions) style proof system. It was originally intended for hardware verification, but because of its ability to handle a variety of applications, it is now considered a general purpose proof system. Some of these applications include security systems, verification of fault-tolerant computers, compiler verification, program refinement calculus, software and algorithms verification, modeling, and automation theory. HOL provides a wide range of proof commands, rewriting tools and decision procedures. The system is user-programmable and proof tools can be developed for specific applications without compromising reliability [44].

The set of types, type operators, constants, and axioms available in HOL are organized in the form of theories. The theories are arranged in a hierarchy. These theories include various formalized mathematical concepts such as lists, products, sums, numbers, primitive recursion, and arithmetic etc. On top of these, users are allowed to introduce application-dependent theories by adding relevant types, constants, axioms, and definitions.

The HOL system supports higher order logic with three main expressions:

- Variables can range over functions and predicates.
- The logic is typed.
- There is no separate syntactic category of formulae.

The basic interface to the system is a Standard Meta Language (SML) interpreter. SML [54] is both the implementation language of the system and the Meta Language in which proofs are written. The HOL system supports both forward and backward proof methods common in a natural-deduction style calculus. In the forward proof method, the steps of a proof are implemented by applying inference rules chosen by the user, and HOL checks that the steps are safe. All derived inference rules are built on top of a small number of primitive inference rules. This approach has some limitations since it is hard to know where to state the proof and, for large proofs, to determine which sequence of rules to apply. The results are strong and the user can have great confidence since the most primitive rules are used to prove a theorem. In the backward proof method, the user sets the desired theorem as a goal. Small programs written in SML called tactics and tacticals are applied to break the goal into a list of subgoals. Tactics and tacticals are repeatedly applied to the subgoals until they can be resolved. In practice, forward proof is often used within backward proof to convert each goals assumptions into a suitable form.

Theorems in the HOL system are represented by values of the ML abstract type thm. There is no way to construct a theorem except by carrying out a proof based on the primitive inference rules and axioms. HOL system has many built-in inference rules and ultimately all theorems are proved in terms of these axioms and basic inferences of the calculus. By applying a set of primitive inference rules, a theorem can be created. Once a theorem is proved, it can be used in further proofs without recomputation of its own proof. In this way, the ML type system protects the HOL logic from arbitrary construction of a theorem, so that every computed value of the type-representing theorem is a theorem. The user can have a great deal of confidence in the results of the system.

The HOL system has been used in hardware verification, reasoning about security, verification of fault-tolerant computers, and reasoning about real-time systems. It has also found application in compiler verification, program refinement calculus,

software and algorithms verification, modeling, and automation theory. HOL also has a rudimentary library facility which enable theories to be shared. This provides a file structure and documentation format for self contained HOL developments. Many basic reasoners are given as libraries such as **mesonLib**, **bossLib**, and **simpLib**. These libraries integrate rewriting, conversion and decision procedures and free the user from performing low-level proof.

3.6 Normal Forms

Definition 1. A formula is in Disjunctive Normal Form (DNF) if it is a disjunction of minterms (conjunctions of literals). In other words, a DNF formula is a sum of products and looks like:

$$(x_{11} \wedge x_{12} \wedge ..... \wedge x_{1n_1}) \vee (x_{21} \wedge ... \wedge x_{2n_2}) \vee ..... \vee (x_{m1} \wedge \ .... \wedge x_{mn_m})$$

where each x_{ij} is a literal. Literal is a variable or it's negation. In short:

$$\bigvee_i \bigwedge_j x_{ij}$$

Definition 2. A *literal* L is either an atom p or the negation of an atom $\neg p$. A formula C is in $Conjunctive\ Normal\ Form(CNF)$ if it is a conjunction of clauses, where each clause D is a disjunction of literals:

$$L ::= p|\neg p$$
$$D ::= L|L \vee D$$
$$C ::= D|D \wedge C$$

Definition 3. Given a formula ϕ in propositional logic, we say that ϕ is *satisfiable* if it has a valuation in which it evaluates to `T`. For example, the formula $p \vee q \rightarrow p$ is *satisfiable* since it computes `T` if we assign `T` to p. Clearly, $p \vee q \rightarrow p$ is not valid. Thus, satisfiability is a weaker concept since every valid formula is by definition also satisfiable but not vice versa. However, these two notions are just mirror images of

each other, the mirror being negation.

Proposition 4. *Let ϕ be a formula of propositional logic. Then ϕ is satisfiable iff $\neg\phi$ is not valid.*

Proof: First, assume that ϕ is satisfiable. By definition, there exists a valuation of ϕ in which ϕ evaluates to `T`; but that means that $\neg\phi$ evaluates to `F` for that same valuation. Thus, $\neg\phi$ cannot be valid.

Second, assume that $\neg\phi$ is not valid. Then there must be a valuation of $\neg\phi$ in which $\neg\phi$ evaluates to `F`. Thus, ϕ evaluates to `T` and is therefore satisfiable. (Note that the valuations of are exactly the valuations of $\neg\phi$).

This result is extremely useful since it essentially says that we need to provide a decision procedure for only one of these concepts. For example, lets say that we have a procedure P for deciding whether any $\neg\phi$ is valid. We obtain a decision procedure for satisfiability simply by asking P whether $\neg\phi$ is valid. If it is, ϕ is not *satisfiable*; otherwise ϕ is *satisfiable*. Similarly, we may transform any decision procedure for satisfiability into one for validity.

In this Chapter, we presented some of the basics required for better understanding the rest of chapters of this book. We provided the basics of temporal logics, Multiway Decision Graph, MDG tool, boolean SAT solvers as well as HOL theorem prover. In the next chapter, we describe our proposed methodology in detail.

Chapter 4

Integrating SAT with MDG

In this chapter, we present the proposed methodology for SAT based invariant checking. SAT has already been integrated with MDG tool as a *reduction engine* in [9]. This chapter provides a step by step description of the complete methodology using SAT as *verification engine* with MDG.

4.1 Formalization of the Problem

Given a state machine M with initial states I and transition relation Tr, we would like to check whether a property P holds for all the reachable states. The reachable states are those which can be reached by Tr transitions starting from an initial state. Let S denote the entire set of states. A system is a *safe system*, where all the reachable states satisfy P.

Introduction of various type of paths through the graph of a transition relation is required to formalize the problem more precisely. We write $Tr(x, y)$ to indicate that x is related to y by a transition relation Tr. We define the sequence of states to be a path through Tr.

$$path(s_{[0\ldots n]}) \triangleq \bigwedge_{0 \leq i < n} Tr(s_i, s_{i+1})$$

Here " $\triangleq$ " sign means "is defined to be" and $s_{[0...n]}$ denotes a sequence of states (set of state) e.g. $s_0, s_1, s_2....s_n$. A path can have a length n, if it makes n transitions. For us, we are interested to show that, starting from the initial state and repeated application of transition relation always leads to a state that satisfies P. We want to show that,

$$\forall i.\forall s_0....si.(I(s_0) \wedge path(s_{[0...i]}) \rightarrow P(s_i))$$

where, $i \geq 0$ and $s_i \in S$. Similarly, proving backward from bad states involves showing that, starting from a state that violates P and going backwards through Tr always leads to a non-initial state, which is

$$\forall i.\forall s_0....si.\neg(I(s_0) \leftarrow path(s_{[0...i]}) \wedge \neg P(s_i))$$

To get a more symmetric view at the problem, we say there are no paths that start in an initial state and end in a non-P-state, that is,

$$\forall i.\forall s_0....si.\neg(I(s_0) \wedge path(s_{[0...i]}) \wedge \neg P(s_i))$$

4.2 Proposed Methodology

We propose a methodology (Figure 4.1) to formulate and verify a formula (we call this formula *Correctness formula*), to check the safety of a system or design. In our methodology, we are interested to check if the formula,

$$\forall i.\forall s_0....si.\neg(I(s_0) \wedge path(s_{[0...i]}) \wedge \neg P(s_i))$$

holds for $i = 0, i = 1, i = 2$ and so on. It is similar to check $I(s_0) \wedge path(s_{[0...i]}) \wedge \neg P(s_i)$ is a `contradiction` for each i, for s_0 to s_i; i.e. $\neg(I(s_0) \wedge path(s_{[0...i]}) \wedge \neg P(s_i))$ is a `tautology`. If the property P is violated in a reachable state, then, $\exists i.I(s_0) \wedge path(s_{[0...i]}) \wedge \neg P(s_i)$ is satisfiable. A SAT solution refers that there exists a path

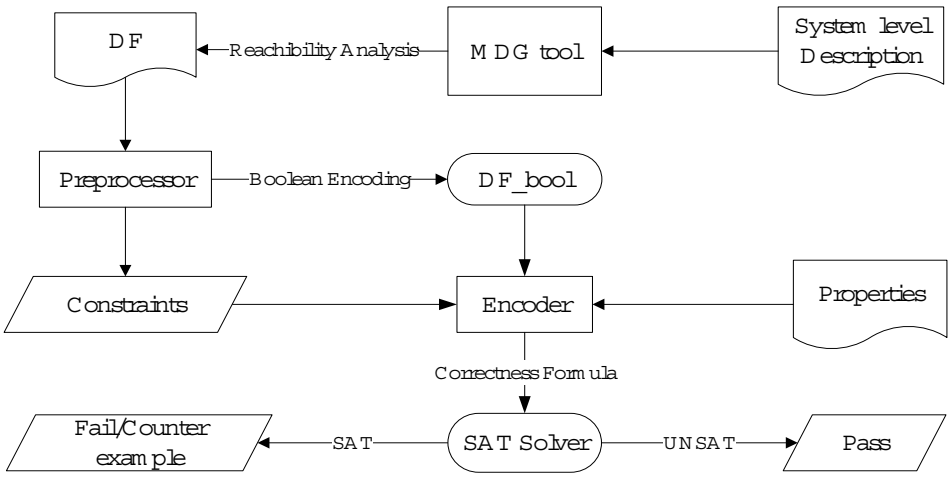

Figure 4.1: Verification Methodology using MDG tool and SAT solver

of length i starting from initial states that violates P and it can be used for tracing errors.

Automation of this approach with MDG involves four main tasks:

- Compute the reachable states, starting from the initial state. It will give us the path by which each possible reachable state can be reached, by each transition, until all the reachable states have been visited.
- The reachable states are computed in *Directed Formula* format. Removal of *Uninterpreted functions* and introduction of boolean encoding is required to convert the formula suitable for boolean SAT solvers.
- Perform the *CNF* conversion of the boolean formula using a linear algorithm to avoid exponential blow up (direct conversion from *DNF* to *CNF* has exponential blow up).
- Fed the formula to a SAT solver to check the satisfiability.

Using SAT solver with MDG tool is a new and efficient approach for invariant checking. The steps in the methodology are as following:

1. We use MDG tool to compute the sets of reachable states for the given MDG model (behavioral/RTL) written in MDG-HDL language. Any other implementation of MDG reachability analysis algorithm can be used instead of MDG tool. We conjunct all the sets of states which gives us the a set $S_{reachable}$ consisting of all the reachable states for the system in DF format.

2. Boolean Encoding is imposed by the preprocessor to reduce the DF in propositional logic. After removal of *uninterpreted functions* the encoder generates a pure boolean formula DF_{bool} with certain encoding constraints.

3. We get formula B_{DF} after conjunction of DF_{bool} with Encoding constrains and *negated* invariant property.

4. The B_{DF} is converted into CNF using Tseitin algorithm. The output is SAT encoded CNF formula in DIMACS [3] format. At this stage we call this formula *Correctness Formula*.

5. The SAT encoded correctness formula is fed to a SAT solver to prove $\neg(S_{reachable} \wedge \neg P \wedge constraints)$ a `tautology` or $(S_{reachable} \wedge \neg P \wedge constraints)$ is a `contradiction`.

Detail description of these steps is explained in the following subsections.

4.2.1 Using MDG for Reachibility Analysis

The presence of uninterpreted symbols in the logic means that we must distinguish between a state machine M and its abstract description D in the logic. This is called Abstract State Machine, a state machine given an abstract description in terms of DFs, or equivalently MDGs, as defined in [25, 11].

Definition 1. An Abstract Description of a State Machine (ASM) M is a tuple $D = (X, Y, Z, Y', IS, Tr, Or)$, where:

- X : finite set of input variables,

- Y : finite set of state variables,
- Z : finite set of output variables,
- IS : MDG of type $U_0 \rightarrow Y$, where U_0 is a set of disjoint abstract variables, IS is the abstract description of the set of initial states,
- Tr : MDG of type $X \cup Y \rightarrow Y'$. Tr is the abstract description of the transition relation,
- Or : MDG of type $X \cup Y \rightarrow Z$. Or is the abstract description of the output relation.

Algorithm 3 shows how the analysis of the reachable states of M is performed based on the abstract description D.

Algorithm 3 MDG Reachibility Analysis

```
1: R := IS;
2: Q := IS;
3: i := 0;
4: while Q ≠ F do
5:     i := i + 1;
6:     IN := new_inputs(i); – Produce new inputs
7:     NS := next_states(IN, Q, Tr); – Compute next state
8:     Q := frontier(NS, R); – Set difference
9:     R := union(R, Q); – – Merge with set of states reached previously
10: end while
```

The algorithm is initialized by the construction of the initial MDG structure in Lines 1-3. In line 4-10, within the while loop, the set of reachable states is computed. When the frontier set (Q) becomes empty (F), the while loop terminates. A new MDG input is produced in line 6. In line 7, Next state is computed by the function *next_state* using the RelP operation, that takes the MDGs representing the set of inputs, the current state and the transition relation as assignment, respectively. In line 8, The function *frontier*, computes the set difference using the PbyS operation, that approximates the set difference between the newly reachable state in the current iteration from the reachable state in the first iteration. Finally, the

set of all reachable states so far is computed, in line 9. The MDG tool applies the reachibility algorithm[10] and gives all the possible sets of reachable states in terms of DF. We conjunct *initial state* with *frontier sets* and *output relations* computed by MDG tool for each transitions to construct the complete DF, representing all the sets of reachable states e.g. $DF_{complete} = DF_0 \wedge DF_1 \wedge DF_2 \wedge DF_3 ..... \wedge DF_n$. Here n is the number of transitions the reachibility analysis algorithm needs to terminate. DF_0 indicates initial state and rest of the each DF is the conjunction of *frontier sets* and *outputs relations*.

4.2.2 Preprocessing to Impose Boolean Encoding

The naive structure of DF contains Uninterpreted Functions and predicates. We convert the DF formula to a boolean formula. The preprocessor eliminates the EUF applications and introduces boolean encoding with adequate constraints. We describe step by step procedure in the following subsections.

Boolean Encoding for Clauses with Constraints

Consider a directed formula $(r = 0) \wedge (f = 1) \vee (r = 1) \wedge (f = 0)$. We introduce Boolean variables r_0, f_1, r_1 and f_0 respectively for abstracting the clause $(r = 0), (f = 1), (r = 1)$ and $(f = 0)$. Constraints are introduced at the same time. For this example, we know that $(r = 0)$ and $(r = 1)$ can not be true at the same time. Meanwhile one of them must be true, forcing them to be mutually exclusive, otherwise the equation will not be satisfiable. A similar constraint is also applicable to $(f = 0)$ and $(f = 1)$. So, after reduction to propositional logic the directed formula looks like:

$$(r_0) \wedge (f_1) \vee (r_1) \wedge (f_0)$$

The constraints for the above example are: $r_0 \oplus r_1$ and $f_0 \oplus f_1$

EUF Elimination

The logic of *Equality with Uninterpreted Functions (EUF)* was first presented by Burch and Dill [22]. The syntax of EUF logic in directed formula is given in [13]. Our EUF elimination approach is inspired by using nested ITEs [67]. We introduce domain variables replacing each function application term with a nested ITE structure that directly holds the functional consistency. For example, if $g(x_1, y_1)$, $g(x_2, y_2)$ and $g(x_3, y_3)$ are three applications of UF $g()$, then the first application will be eliminated by a new term variable c_1. The second one will be replaced by $ITE((x_2 = x_1) \wedge (y_2 = y_1), c_1, c_2)$, where c_2 is a new term variable. The third one will be replaced by $ITE((x_3 = x_1) \wedge (y_3 = y_1), c_1, ITE((x_3 = x_2) \wedge (y_3 = y_2), c_2, c_3))$, where c_3 is a new term variable.
For ITE terms, we define $encITE$ as:

$$encTr(ITE(G, T_1, T_2)) = encDF(G) \wedge encTr(T_1) \vee \neg encDF(G) \wedge encTr(T_2)$$

where $encTr(T_1)$ and $encTr(T_2)$ represent Boolean encoded terms and $encDF(G)$ represents an encoded propositional of formula G. For some cases, we modified Bryant's encoding slightly for the MDG DF case. For example, if the formula inside ITE contains a comparison between two different constants (such cases sometime occurs in MDG DF), then it is always false. So, we define the encoding for such cases as:

$$encTr(ITE(G_{const_1 = const_2}, T_1, T_2)) = encTr(T_2)$$

The Min-Max example

For the illustration of EUF elimination approach, we consider the MIN-MAX circuit described in [25, 11]. Figure 4.2 represents the MIN-MAX state machine which has two input variables, $X = \{r; x\}$ and three state variables $Y = \{c; m; M\}$, where r and c belongs to the Boolean sort B, a concrete sort with enumeration $\{0; 1\}$, and x, m, and M are of an abstract sort s. The outputs coincide with the state variables,

which means, all the state variables are observable and therefore no additional output variables is introduced.

The transition labels denote the conditions under which each transition can be taken and an assignment of values to the abstract next state variables n_m and n_M. The machine stores in m and M, respectively. m and M are the smallest and the greatest values respectively, presented at the input x since the last reset ($r = 1$). m is loaded by the maximal possible value max and M by the minimal possible value min, if the machine is reset. The min and max symbols are uninterpreted generic constants of sort s. An operator leq_Fun computes the smallest and greatest values, such that for any two values p and q of sort s, $leq\_Fun(p, q) = 1$ if and only if p is less than or equal to q. The transition relation can be described by a set of individual transition relations, one related with each next state variable. The directed formulas of these individual transition relations, for a particular custom symbol order, are stated below:

$$
\begin{aligned}
Tr\_c \quad &= \quad [((r = 0) \wedge (n\_c = 0)) \vee \\
&\qquad ((r = 1) \wedge (n\_c = 1))] \\
Tr\_m \quad &= \quad [((r = 0) \wedge (c = 0) \wedge (n\_m = m) \wedge (leq\_Fun(x, m) = 0)) \vee \\
&\qquad ((r = 0) \wedge (c = 0) \wedge (n\_m = x) \wedge (leq\_Fun(x, m) = 1)) \vee \\
&\qquad ((r = 0) \wedge (c = 1) \wedge (n\_m = x)) \vee \\
&\qquad ((r = 1) \wedge (n\_m = max))]
\end{aligned}
$$

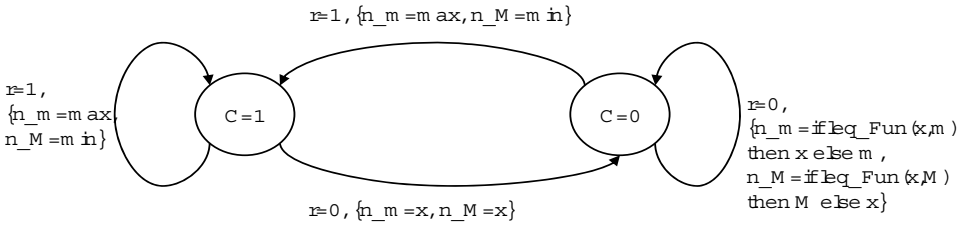

Figure 4.2: MIN-MAX State Machine

$$
\begin{aligned}
Tr\_M \quad = \quad & [((r=0) \wedge (c=0) \wedge (n\_M = x) \wedge (leq\_Fun(x, M) = 0)) \vee \\
& ((r=0) \wedge (c=0) \wedge (n\_M = M) \wedge (leq\_Fun(x, M) = 1)) \vee \\
& ((r=0) \wedge (c=1) \wedge (n\_M = x)) \vee \\
& ((r=1) \wedge (n\_M = min))]
\end{aligned}
$$

The conjunction of these individual transition relations represents the DF of the system transition relation Tr. We illustrate with this example, how we reduce this directed formula to propositional logic. We consider the Directed formula describing the transition relations. Directed formula representing the set of states can be reduced to propositional logic in the same way. We have two appearance of same function symbol with different arguments. We define two term variables $U1$ and $U2$ for these terms. But we are not imposing ITE chain here, because M and m are constants, so $(m = M)$ is always false. So we are introducing a new domain variable for both the cases:

$$U1 = leq\_fun(x, m) = uf1$$
$$U2 = leq\_fun(x, M) = uf2$$

We should introduce some constraint to establish a relation between $uf1$ and $uf2$. For this case:

$$uf1 \Rightarrow uf2$$
$$= \neg uf1 \vee uf2$$

We replace the UF application and introduce boolean variables for the rest of the terms. After replacing all the terms with boolean variables we get an equivalent boolean formula:

$$
\begin{aligned}
Tr\_c &= [((r0) \wedge (nc0)) \vee \\
&\quad ((r1) \wedge (nc1))] \\
Constraints &= r0 \oplus r1 \\
&\quad nc0 \oplus nc1 \\
Tr\_m &= [((r0) \wedge (c0) \wedge (nm\_eq\_m) \wedge (uf1\_eq\_0)) \vee \\
&\quad ((r0) \wedge (c0) \wedge (nm\_eq\_x) \wedge (uf1\_eq\_1)) \vee \\
&\quad ((r0) \wedge (c1) \wedge (nm\_eq\_x)) \vee \\
&\quad ((r1) \wedge (nm\_eq\_max))] \\
Constraints &= r0 \oplus r1 \\
&\quad nc0 \oplus nc1 \\
&\quad nm\_eq\_x \oplus nm\_eq\_m \\
&\quad uf1\_eq\_0 \oplus uf1\_eq\_1 \\
Tr\_M &= [((r0) \wedge (c0) \wedge (nM\_eq\_x) \wedge (uf2\_eq\_0)) \vee \\
&\quad ((r0) \wedge (c0) \wedge (nM\_eq\_M) \wedge (uf2\_eq\_1) \vee \\
&\quad ((r0) \wedge (c1) \wedge (nM\_eq\_x)) \vee \\
&\quad ((r1) \wedge (nM\_eq\_min))] \\
Constraints &= r0 \oplus r1 \\
&\quad nc0 \oplus nc1 \\
&\quad nM\_eq\_x \oplus nM\_eq\_M \\
&\quad uf2\_eq\_0 \oplus uf2\_eq\_1
\end{aligned}
$$

4.2.3 CNF Conversion of Directed Formula

The *encoder* takes the Boolean encoded directed formula DF_{bool} as input and conjunct the encoding constrains and the negated property $\neg P$ with it. In this step the formula to be converted to CNF can be expressed by:

$$B_{DF} = DF_{bool} \wedge Constraints \wedge \neg P$$

Algorithm 4 CreateCNFFormula(DF)

1: Formula = MDG Direct Formula;
2: DF_{bool} = Replace UF's by introducing term variables;
3: Infer constraints between predicates;
4: Transform predicates to Boolean variables;
5: **for** each DNF_i in DF_{bool} **do**
6: $\quad CNF_{DNF_i}$ = ConvertoCNF(DNF_i)
7: **end for**
8: $CNF_{complete}$ = Conjunct all CNF_{DNF_i}
9: $ReturnCNF_{complete}$;

After CNF conversions we call this formula a correctness formula:

$$CorrectnessFormula = CNF(B_{DF})$$

Algorithm 4 shows the complete algorithm for the encoding and conversion. A B_{DF} can be a single DNF formula (representing the set of states) or conjunction of several individual DF, where each of these DF is in DNF format(representing transition relations):

$$DF_{complete} = \bigwedge_i DF \tag{4.1}$$

where i is the number of transitions the and DF_i is a DNF. So, it is enough to get the equivalent CNF for each DF_i and conjunct them because conjunction of CNF is also a CNF.

$$DF_{CNF} = \bigwedge_i CNF_{DF} \tag{4.2}$$

Linear algorithm for computing $CNF(DF)$ is well known as Tseitin [73]. In Tseitin a new variables for every logical gate is introduced. Thus variables impose a constraint that preserve the function of that gate. Given a DNF formula

$$(a \wedge b) \vee (c \wedge d) \tag{4.3}$$

with Tseitin encoding, a new variable for each subexpression is introduced. In this example, let us assign the variable x to the first 'and' gate (representing the subexpression $a \wedge b$), y for the second 'and' gate (representing the subexpression $c \wedge d$). We also introduce a new variable z to represent the top most operator. For

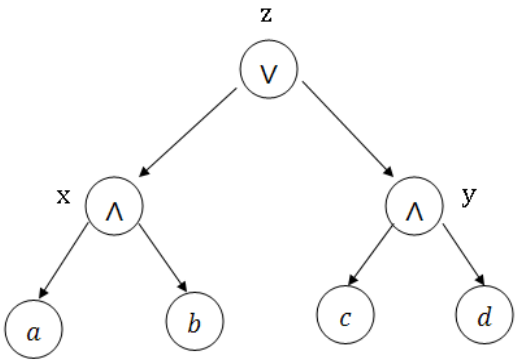

Figure 4.3: Tesitin encoding to convert a propositional formula to CNF linearly

DF, the top most operator which is always an 'OR' gate connected with several 'AND' gates. Figure 4.3 illustrates the parse tree of our formula. We need to satisfy the two equivalences:

$$\begin{aligned} x &\Longleftrightarrow a \wedge b \\ y &\Longleftrightarrow c \wedge d \end{aligned} \tag{4.4}$$

The overall CNF formula is the conjunction of the two equivalences written in CNF as:

$$(\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b \vee x) \wedge$$
$$(\neg y \vee c) \wedge (\neg y \vee d) \wedge (\neg c \vee \neg d \vee y)$$

The unit clause (z) which represents the top most operator. Instead of (z) we use $(x \vee y)$, which represents the same. The converter keeps track by mapping the Tseitin variable for each logic gates. In the example, *Equation* (4.4) represents this mapping. Such mapping will be fed to the goal generator in the next step for verification of the conversion.

4.2.4 Verification of the Conversion

Application and improvement of different linear algorithms for CNF conversion has been a major research interest for researchers [72, 16, 32, 23]. However, we could

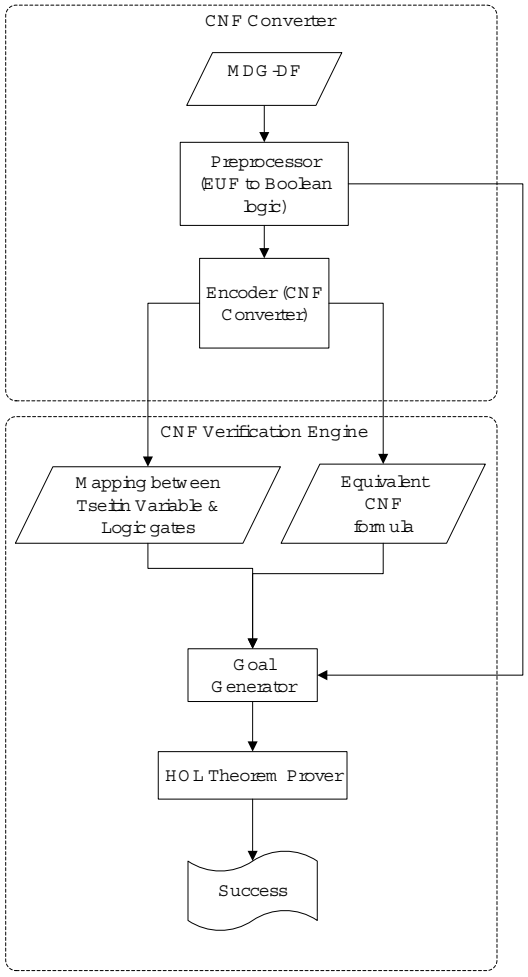

Figure 4.4: Overview of the DF to CNF conversion-verification methodology

not find any automated methodology to formally verify the conversion approach. This motivated us to give some extra effort and integrate a small tool that formally verifies our CNF conversion.

Our automated conversion-verification methodology in Figure 4.4 using HOL theorem prover is an extra contribution to the methodology to demonstrates the correctness of the CNF conversion automatically. The obtained CNF formula is compared formally to the original DF using the HOL theorem prover. This will enhance confidence in the whole verification process. The verification part of the methodology contains a goal generator and HOL Theorem prover. The goal generator generates the goal to be proved by the HOL theorem prover. At the end, HOL provides a decision based on the inputs.

Goal Generator

The goal generator takes the CNF formula, Tseitin variable for each logic gate mapping generated by the converter and the Boolean encoded DF as input. Given the Tseitin variable for each logic gate mapping, the assumptions are computed by the goal generator. The assumptions for the previous conversion example(Figure 4.3) can be written as:

$$\begin{aligned} x &= a \wedge b \\ y &= c \wedge d \end{aligned} \tag{4.5}$$

At the end, the goal generator ends up with generating a complete goal to prove in HOL:

$$Assumptions \Longrightarrow EncodedDF \Longleftrightarrow CNFFormula$$

Call to the HOL theorem prover

As we mentioned earlier, we used HOL theorem prover to prove the goal. After generating the goal, the goal generator places a call to the HOL theorem prover. Given the input goal, the proof is conducted by applying rewriting rules. Note that

the goal is generated in such a way that only one `Tactic` is enough to decide the goal.

4.2.5 Specification of Invariant Property and Correctness Formula

In our proposed methodology, we check the safety of a design by checking invariant property. The specification is in a commonly encountered generic form of safety properties, $M \vDash P_{init} \Rightarrow AGP_s$, where P_{init} and P_s are *instantaneous formulas* not containing temporal operators. A safety property of this form is called invariant and has the intuitive interpretation that every computation of M, which starts in a state satisfying P_{init} also satisfies P_s at all reachable states. For example, heating should be turned of when the door of a microwave-oven is open. This invariant property can be expressed in *CTL* logic as follows:

$$AG(!(door = open)\&(heating = on))$$

In order to build a correctness formula we consider $EF(\neg P)$; the negation of the property. The encoder described in 4.2.3 conjuncts the negated property with the encoding constraints and boolean encoded directed formula. The CNF representation of this formula is a correctness formula:

$$Correctness formula = CNF(Directed\ formula\ representing\ all\ the\ reachable\ states \wedge EncodingConstraints\ \wedge\ Negated\ invariant\ property).$$

We use SAT solver to prove the correctness formula *UNSAT* i.e. `contradictory`. For the microwave-oven example, we use the SAT solver to prove that there is a state where $(door = open)$ and $(heating = on)$. If no such path exist, where such state occurs, SAT solver will give an *UNSAT* decision.

4.2.6 Using SAT as a Verification Engine

The use of SAT-solvers in various applications is on the march. As insight on how to efficiently encode problems into SAT is increasing, a growing number of problem domains are successfully being tackled by SAT-solvers. This is particularly true for the electronic design automation (EDA) industry [17, 52]. The success is further magnified by current state-of-the-art solvers being adapted to meet the specific characteristics of these problem domains [14, 34].

The method of representing an instance of a search problem as a propositional formula so that a satisfying assignment represents a solution, and then running a SAT solver to find such an assignment if there is one, has been found to be a practical and effective method for solving a number of problems. It has been used successfully in the electronic design automation (EDA) industry for a variety of tasks including microprocessor verification [18] and automated test generation [53] among many others [58]. Perhaps most notably, SAT-based bounded model checking [71] has become a widely used verification method, and these methods are being extended to un-bounded model checking [60]. SAT solvers are used as computational engines in a variety of model checking tools such as NuSMV[2]. SAT solvers, or modifications of them, are used as the engines for tools using more expressive logics, including for problems that we expect are not in NP, such as answer set programming [42, 56], quantified boolean formulas and modal logics [43], and even restricted first order theorem proving [1].

The successful SAT-MDG integration as a *reduction engine* [9] motivated us to for a new methodology, using the SAT solver as a *verification engine* for MDG model. Given a correctness formula, a SAT solver can be used to search for a path such that the property holds true at all the nodes in that path. If at least one such path exists, then the formula is *satisfiable*, indicating that property is true for the given model. Absence of a feasible path indicates a violation of the property. We use *MiniSAT 2.0*[33] as an efficient SAT solver. As our approach is to prove the

correctness formula as a tautology, so, a *satisfiable* decision by the solver indicates violation of the property and gives a counter example, whereas an *unsat* decision validates the property. If *satisfiable*, the assignments constitutes a counter example to the original(un-negated) formula. Optionally, the satisfiable assignments can be substituted in the negation of the formula and a theorem that the counter example implies the negated formula can be derived. To explain more in detail, in the next chapter, we will present an illustrative example of SAT based invariant checking.

In this chapter we presented the overview of our proposed methodology. Also we provided step by step description of our the methodologies with examples. In next chapter, we present implementation and experimental result of the methodologies we described above.

Chapter 5

Implementation and Case Study

In this chapter, we discuss the implementation details and experimental results of our proposed methodology, to integrate SAT as a *verification engine* with MDG. Unlike other researchers, we implement not just the conversion algorithm to convert MDG Direct formula (DF) to CNF(which will be fed to the SAT solver), but also implement an automated verification technique to formally verify the conversion. We present and analyze the experimental results that we obtained for both of the methodology, supporting the correctness, soundness and performance of them.

5.1 Conversion-Verification of Directed Formula

5.1.1 Experiment Description

We implement our methodology in C++ and run it on several different sized directed formulas, each of them containing different number of clauses and variables. For the experiment we considered DF with minimum 100 clauses to maximum 1000 clauses. Those clauses contain from 38 upto 168 different variable. The experiments are performed under Fedora Core 9 on an Intel Xeon 3.4 GHz processor with 3 GB of RAM. We summarize the results of conversion and verification runtime in the next

subsection.

5.1.2 Experimental Results

Table 5.1 shows the experimental results. Our program produced a delay of less than 0.01 second for the DF with less than 100 clauses. Hence, we increased both the number of clauses and the number of variables with some bigger sized DFs to check the performance. We observed, a very fast response time of 0.02 second with a larger DF with 300 clauses of 78 variables. Conversion time increased to 0.04 second for the DF with 500 clause and 118 variables. The largest DF we tested with our methodology is 1000 clauses with 168 variables. Out program took only 0.1 second to compute the CNF of that DF. Figure 5.1 shows a nearly linear behavior for our implementation. The slight deviations from linearity is ignored(those are caused by the interruption internal processes of the operating system).

On the other hand, we noticed, the verification time in HOL increases with the size of directed formula. HOL took a few seconds for the verification of smaller sized DFs, but suffers for bigger sized DFs while taking a longer time to prove. As we mentioned earlier, the way we constructed the goal requires only one Tactic (`DECIDE_TAC`) for proving the goal, which is a positive side for the methodology. For a DF with 100 clauses, our conversion produced a less than 0.00 second delay, where as HOL took about 4.010 seconds to verify the conversion. HOL took about 14.901 and 28.021 seconds to prove the conversion of DFs with 300 and 500 clauses, respectively,

Table 5.1: CNF conversion time

DF size	**No. of variables**	**Conversion time**	**Verification time**
100	38	less than 0.01	4.010
200	58	0.01	8.231
300	78	0.02	14.908
400	98	0.03	19.042
500	118	0.04	28.021
700	148	0.06	53.098
1000	168	0.10	93.118

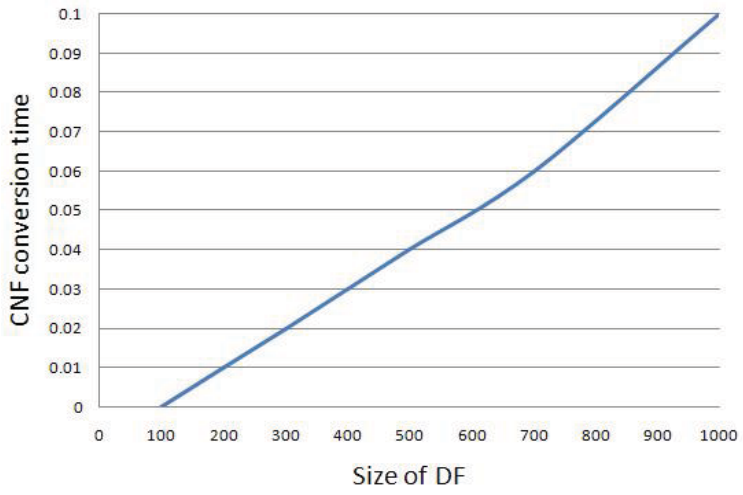

Figure 5.1: DF size vs. CNF conversion time

which is more than the conversion time. The verification time increases sharply for the DF with 1000 clauses of 168 variables. But for all cases, HOL successfully verified the conversion.

5.2 Integrating SAT with MDG for Invariant Checking

The main goal of our work is to integrate SAT with MDG for a new invariant checking methodology. In the following subsection, we present a small *abstract counter* example to illustrate the approach. Also, we present a case study to show the performance of our approach.

Abstract Counter Example

In the abstract counter example, we show how we verify the invariant on a design. Abstract counter example was introduced in [63]. Figure 5.2 shows the state transition graph of the counter. There are five control states: *c_fetch, c_load, c_inc1, c_inc2, c_dec*. Depending on the input, the counter *pc* gets a new value, increased

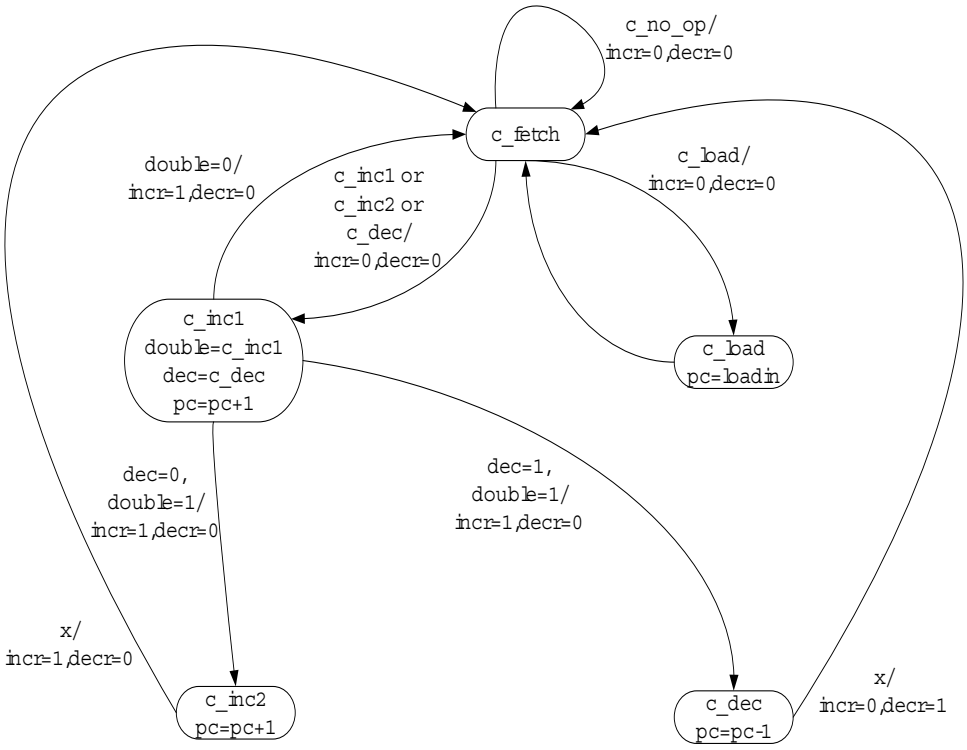

Figure 5.2: An abstract counter

by one, decreased by one or keeps the previous value.

To apply our methodology, we first describe the behavior of the counter using the MDG-HDL language. The counter PC is of abstract sort. The control state is initialized to c_fecth and the initial value of pc is a free variable called $init\_pc$ (i.e. the initial value is generalized to any value). The output variables $incr$ and $decr$ is initialized to '0'. The MDG tool computes all the reachable states in three transition steps. Conjunction of these set of states, for each transition, generates the complete set of reachable states for the design. Preprocessor applies the boolean encoding and generates a boolean formula B_{DF} along with the encoding constraints. The Encoder applies the CNF conversion and generates the correctness formula to be verified with a SAT solver. We consider the following property to be verified:
Property 1: Starting from the c_fetch state, there is no such state in the future, where pc is incremented and decremented at the same time. The property can be expressed as an invariant as following:

$$AG\ !(incr = 1\ \&\ decr = 1)$$

We check this invariant using both our SAT-MDG methodology and MDG tool. Table 5.2 summarizes the result. The SAT-MDG approach including boolean encoding of directed formula, CNF conversion with correctness formula generation and using SAT solver for decision took about 0.0422 seconds in total. On the other hand using MDG as a stand alone tool for the invariant checking took about 0.220 seconds. Even though we use the MDG tool to compute the reachable states, the negligible verification time (in comparison to MDG tool) of MDG-SAT technique gives us a clear indication about the efficiency of our proposed methodology.

Table 5.2: Invariant Checking Results with MDG tool and MDG-SAT approach

Property	**MDG-SAT approach**				**MDG tool**
	Preprocessing Time	Encoding Time	Decision Time	Total Time	
Property 1	0.02	0.02	0.002295	0.0422	0.220

5.2.1 Case Study: Island Tunnel Controller (ITC)

System Description

The results from abstract counter example motivated us to check the efficiency with larger design. The SAT-MDG reduction technique is demonstrated on the example of the Island Tunnel Controller (ITC) [9], which was originally introduced by Fisler and Johnson [39]. We illustrate our SAT-MDG verification methodology on the same example. Based on the information collected by sensors installed at both ends of the tunnel, the ITC controls the traffic lights at both ends of a tunnel : there is one lane tunnel connecting the mainland to an island. As shown in Figure 5.3, at each end of the tunnel, there is a traffic light. Four sensors are used to detect the presence of cars: one at tunnel entrance on the island side *(ie)*, one at tunnel exit on the island side *(ix)*, one at tunnel entrance on the mainland side *(me)*, and one at tunnel exit on the mainland side *(mx)*. The following constraint is imposed in [39] : maximum sixteen cars may be on the island at any time. The assumptions includes all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, that cars do not leave the tunnel entrance without traveling through the tunnel, and sufficient distance is maintained between two cars so that the sensors can distinguish the cars.

As depicted in Figure 5.4, the ITC specification is composed of three communication controllers and two counters. The communication controllers are: The Island Light Controller *(ILC)*, the Tunnel Controller *(TC)*, the *Mainland Light Controller (MLC)*. The two counters are: the Island Counter and the Tunnel Counter

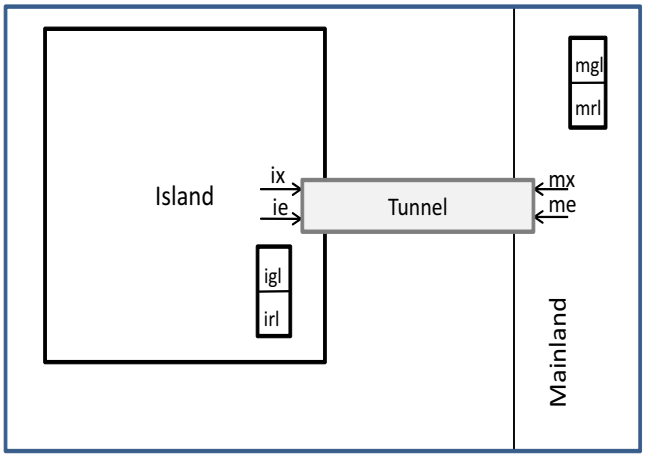

Figure 5.3: The Island Controller

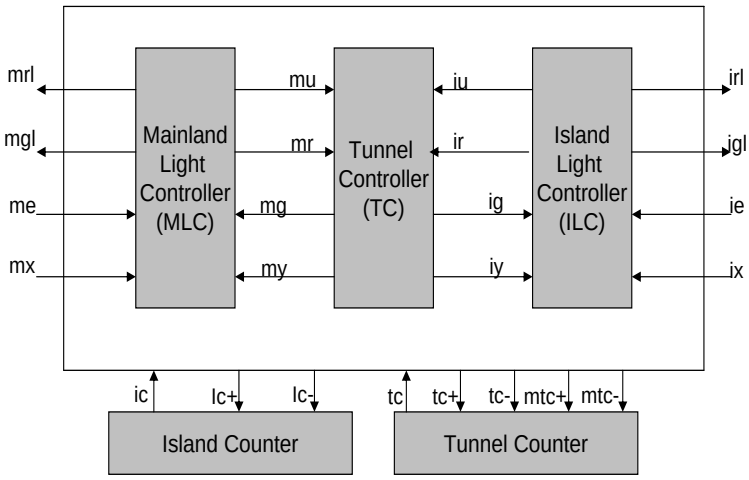

Figure 5.4: The Island Controller

(refer to [39] for the state transition diagrams of each component). The Island Light Controller *(ILC)* has four states: *green, entering, red and exiting.* The green and red lights on the island side are controlled by the outputs *igl* and *irl* respectively; *iu* denotes that the cars from the island side are currently occupying the tunnel, and *ir* denotes that *ILC* is requesting the tunnel. As shown in Figure 5.4, the input *iy* requests the *ILC* to release the control of the tunnel, and *ig* grants control of the tunnel from the island side. For the *Mainland Light Controller (MLC)*, a similar set of signals is defined. The requests for access issued by *ILC* and *MLC* is processed by *The Tunnel Counter (TC).* The number of cars currently on the island and in the tunnel is monitored by the *The Island Counter* and the *Tunnel Counter*, respectively. In the case of tunnel controller, the counter *tc* is increased by 1 depending on *tc+* or decremented by 1 depending on *tc-* unless it is already 0. The Island Counter functions in a similar way, except that increment and decrement depend on *ic+* and *ic-*, respectively: one for the island lights, one for the mainland lights, and one tunnel controller to process the access requests issued by the other two controllers.

5.2.2 Implementation Description

Property checking is handy to verify that a specification meets the certain requirements. In [77, 81, 9],verification of Island Tunnel Controller (ITC) thorough Model checking was performed. Three different invariant properties were verified for this circuit in [61, 81]. We list below those three properties with their corresponding *CTL* formula:

- **Property 1:** Cars never travel both directions in the tunnel at the same time. $AG(!((igl = 1)\&(mgl = 1)))$
- **Property 2:** The tunnel counter is never signaled to increment simultaneously by the ILC and the MLC.

$AG(!((itc+ = 1)\&(mtc+ = 1)))$

- **Property 3:** The island counter is never signaled to increment and decrement simultaneously. controller requests.
  $AG(!((ic- = 1)\&(ic+ = 1)))$

To check the correctness and efficiency of our proposed methodology, we modified the property in a way so that it fails in both: invariant checking by MDG as a stand alone tool and SAT-MDG approach :

- **Property 1:** $AG(!((igl = 1)\&(mgl = 0)))$
- **Property 2:** $AG(!((itc+ = 1)\&(mtc+ = 0)))$
- **Property 3:** $AG(!((ic- = 1)\&(ic+ = 0)))$

The MDG tool computes the reachable states for the given MDG-HDL model of ITC tunnel controller. Although, to ensure the correctness of the design, all the reachable states should be considered and conjuncted to build the complete $S_{reachable}$, we consider the first five reachable states and the initial state and was able to identify the violation of property (the granularity can be adjusted to identify the first violation of property). The reason behind that is the implementation of interface parser to collect the reachable states from the MDG tool and to fed the preprocessor is not yet available. This is a straight forward implementation issue which can be implemented easily as future work for the completeness of the methodology. So we decided not to spend much time on it and concentrate on the other important parts of the methodology. Also as we use MDG tool for reachability analysis so we don't include the reachable state computation time in the MDG-SAT experiential results.

To evaluate three different properties, we generate three different *correctness formula*. An UNSAT decision from SAT solver validates the property whereas a

SAT decision indicates the violation of the property. For the experiments, solaris 5.10 workstation was used containing a quad-core processor running at 2.5GHz and having 6 GB of physical memory.

5.2.3 Experiment Results

Table 5.3 summarizes the results of our MDG-SAT approach. Preprocessor imposes the boolean encoding on it with adequate constraints. For *Property 1* the preprocessor took only 0.5 seconds and similar time was taken for the other two properties. *Correctness formula* is generated by the encoder. Encoder conjuncts the constraints and the negated property with the directed formula representing the reachable states. Later on, the encoder generates a correctness formula, i.e an equivalent CNF representation. In our experiment correctness formula generation for all the properties took same time, 0.06 seconds because of the similar size of the property. We check the satisfiability of the correctness formula using MiniSAT 2.0. MiniSAT took 0.00361 seconds to fail *Property 1*. *Property 2* and *Property 3* took 0.00538 seconds and 0.00539 to fail the property.

We verify those properties with MDG tool and summarize the results in Table 5.4. As we use the reachability analysis feature of MDG tool and unable to extract the time to compute reachable individual set of states, so we don't compare the results but the table clearly shows the efficiency of MDG-SAT approach. MDG-tool failed Property-1 in 0.95 seconds where as MDG-SAT approach took only 0.11361 seconds to do the same. For Property-2 and Property-3, MDG-tool took

Table 5.3: Total time for SAT-MDG approach

Benchmark Properties	**Preprocessing Time**	**Encoding Time**	**Decision Time**	**Total Time**
P1	0.05	0.06	0.00361	0.11361
P2	0.04	0.06	0.00538	0.10538
P3	0.04	0.06	0.00539	0.10539

Table 5.4: Invariant checking time: SAT-MDG and MDG tool

Benchmark Properties	**MDG Time**	**MDG-SAT Time**
P1	0.81	0.11361
P2	0.920	0.10538
P3	0.910	0.10539

0.92 and 0.91 seconds. On the other hand, MDG-SAT approach took only 0.10538 and 0.10539 second to fail those same properties. The results show the efficient and effectiveness of our proposed approach the prospect as a new tool. Implementation of MDG reachability analysis algorithm in any language will give us a completely new tool combining both SAT and MDG. Use of MDG to represent the circuit provides a higher level of abstraction. Also, the use of SAT solver as a fast and efficient verification engine and its fast search algorithm to find the states violating the properties will facilitate the tool.

In this chapter, we presented the results of our proposed conversion-verification methodology and SAT based invariant checking methodology. From the results we observed that our SAT-MDG methodology is very efficient and the fast verification time gives us the indication about the prospect of a new tool development using our methodology . Also, the experimental results of conversion-verification methodology showed the correctness and soundness of our approach while increasing the confidence in whole verification approach.

Chapter 6

Conclusion and Future work

MDG based model checking is an improvement over traditional BDD-based model-checking [13]. The design is represented in a higher level of abstraction, simplifying the data path operations. As a result, users can effectively deal with the state explosion problem. Integrating SAT with MDG is a new concept to enhance the performance of safety checking and our proposed methodology for the new tool shows the efficiency through the experimental results.

In our work, we propose the integration of a SAT solver with MDG as a verification engine. We presented a conversion-verification methodology for the CNF conversion of MDG DF with verification of this conversion. This enhances the confidence in whole verification approach. Our automated verification technique for the CNF conversion is a new contribution to this field of research. Researchers working with CNF conversions inspired by Tseitin algorithm, or slight modification/enhancement of Tseitin algorithm can easily apply this automated technique to formally verify their conversion. In chapter 5, experimental results with different sized formula showed the correctness of our approach.

We presented our main contribution, the proposed methodology to integrate the SAT solver with MDG for verification of safety properties in Chapter 4. The invariant checking results of *abstract counter* example presented in chapter 5 indicates

the improvement of performance. Later on, the *Island Tunnel Counter(ITC)* case study shows the efficiency of our invariant checking approach. We claim, our implementation of the methodology, is a prototype for a new tool for efficient invariant checking.

Using SAT solver as a verification engine with MDG has a wide range of research area. Our future work includes to apply this conversion-verification technique with algorithms other than Tseitin. The experimental result showed that with increasing the number of DFs, HOL suffers to prove the goal with larger runtime. This gives us more area to improve the performance. However, different goal generation techniques with and without quantifiers can also be a good research topic. Also, using different algorithms on MDG DF for CNF conversion and comparing different SAT solver's performance for invariant checking can also be interesting. As a continuation of this research work of integrating SAT with MDG, some new and interesting case studies will also be handy. So, future directions will concentrate on experimenting the methodology with industrial circuits and comparing the results with other industrial model checkers.

Bibliography

[1] Grande (ground and decide). http://www.cs.miami.edu/~tptp/ATPSystems/GrAnDe/.

[2] Nusmv home page. http://nusmv.fbk.eu.

[3] *Satisfiability Suggested Format*. http://people.sc.fsu.edu/ burkardt/pdf/dimacs_cnf.pdf.

[4] Smv home page. http://www.cs.cmu.edu/~modelcheck/smv.html.

[5] Spin home page. http://spinroot.com/spin/whatispin.html.

[6] Synopsys formality home page. http://www.synopsys.com/products/verification/verification.html.

[7] Vis home page. http://vlsi.colorado.edu/~vis/whatis.html.

[8] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS*, pages 411–425, 2000.

[9] S. Abed, O. Ait Mohamed, Z. Yang, and G. Al Sammane. Integrating SAT with Multiway Decision Graphs for Efficient Model Checking. In *Proc. of IEEE ICM'07*, pages 129–132, Egypt, 2007. IEEE Press.

[10] Sa'ed Abed, Otmane Aït Mohamed, and Ghiath Al Sammane. Reachability analysis using multiway decision graphs in the hol theorem prover. In *SAC*, pages 333–338, 2008.

[11] Sa'ed Rasmi H. Abed. *The Verification of MDG Algorithms in the HOL Theorem Prove*. PhD thesis, Concordia University, Canada, 2008.

[12] G. Ackermann. Solvable cases of the decission problem. *North-Holand, Amsterdam*, 1954.

[13] O. Ait-Mohamed, X. Song, and E. Cerny. On the non-termination of MDG-based abstract state enumeration. *Theoretical Computer Science*, 300:161–179, 2003.

[14] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. Generic ilp versus specialized 0-1 ilp: an update. pages 450 – 457, nov. 2002.

[15] Alessandro Armando and Luca Compagna. SAT-based model-checking for security protocols analysis. *Int. J. Inf. Sec.*, 7(1):3–32, 2008.

[16] A. Biere. *AIGER Format and Toolbox*. http://fmv.jku.at/aiger.

[17] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM.

[18] German Bryant and Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic, Vol.2*, 2001.

[19] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[20] R. Bryant, D. Kroening, J. Ouaknine, S. Seshia, O. Strichman, and B. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *STTT*, 11(2):95–104, 2009.

[21] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Transactions on Computers*, C-35(8):677–691, 1986.

[22] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[23] B. Chambers, P. Manolios, and D. Vroon. Faster sat solving with better cnf generation. DATE, 2009.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Language and Systems*, 8(2):244–263, 1986.

[25] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway Decision Graphs for automated hardware verification. In *Formal Methods in System Design*, volume 10, pages 7–46, February 1997.

[26] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[27] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[28] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[29] David Déharbe and Anamaria Martins Moreira. Using induction and bdds to model check invariants. In *Proceedings of the IFIP WG 10.5 International*

Conference on Correct Hardware Design and Verification Methods, pages 203–213, London, UK, UK, 1997. Chapman & Hall, Ltd.

[30] Cerny E., Corella F., Langevin M., Song X., Tahar S., and Zhou Z. *Automated Verification with Abstract State Machines using Multiway Decision Graphs*. Formal Hardware Verification: Methods and Systems in Comparison. Springer Verlag, 1997.

[31] Emerson E.A. *Temporal and Modal Logic*. Handbook of Theoretical Computer Science. Elsevier Science Publisher, 1987.

[32] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up sat. In *SAT*, pages 272–286, 2007.

[33] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[34] Niklas Een and Niklas Srensson. Temporal induction by incremental sat solving, 2003.

[35] Clarke E.M. and Emerson E.A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131, pages 52–71. Springer Verlag, 1981.

[36] D.E.Long E.M..Clarke, O.Grumberg. Model checking. In *Nato ASI, Volume 152, Springer-Verlag, 1996*.

[37] Corella F., Langevin M., Cerny E., Zhou Z., and Song X. State enumeration with abstract descriptions of state machines. In *Proc. IFIP WG 10.5*, 1995.

[38] X.Song M.Langevin F.Corella, Z.Zhou and Eduard Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.

[39] K. Fisler and S.D. Johnson. Integrating design and verification environments through a logic supporting hardware diagrams. pages 669 –674, aug. 1995.

[40] M. Ganai and Aarti Gupta. Completeness in SMT-based BMC for software programs. In *DATE*, pages 831–836, 2008.

[41] Malay K. Ganai and Adnan Aziz. Improved SAT-based bounded reachability analysis. In *VLSI Design*, pages 729–734, 2002.

[42] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Sat-based answer set programming. In *AAAI'04: Proceedings of the 19th national conference on Artifical intelligence*, pages 61–66. AAAI Press, 2004.

[43] Enrico Giunchiglia, Armando Tacchella, and Fausto Giunchiglia. Sat-based decision procedures for classical modal logics. *J. Autom. Reason.*, 28(2):143–171, 2002.

[44] M. Gordon and T. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, NY, USA, 1993.

[45] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979.

[46] Aarti Gupta, Malay K. Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. Learning from bdds in SAT-based bounded model checking. In *DAC*, pages 824–829, 2003.

[47] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In *FMCAD*, pages 354–371, 2000.

[48] Kamran Hussain. Abstract Property Language for MDG Model-checking Tool. Master's thesis, Concordia University, Canada, 2007.

[49] Michael Huth. Model checking modal transition systems using kripke structures. In *VMCAI*, pages 302–316, 2002.

[50] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[51] T. Kropf. Introduction to formal hardware verification. In *Springer; ISBN-13: 978-3540654452*.

[52] T. Larrabee. Test pattern generation using boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(1):4 –15, jan. 1992.

[53] Tracy Larrabee. Efficient generation of test patterns using boolean satisfiability. In *IEEE Trans. on CAD*, pages 795–801, 1990.

[54] L.C.Paulson. *ML for the working programmers*. New York, USA. Cambridge University Press, 1991.

[55] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM Press.

[56] Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. In *Artificial Intelligence*, pages 112–117, 2002.

[57] Gordon M. and Melham T. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, U.k., 1993.

[58] Joao P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Design Automation Conf*, pages 675–680. ACM Press, 2000.

[59] Joao P. Marques-Silva, Joo P. Marques Silva, Karem A. Sakallah, and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *in Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, 1996.

[60] K. L. Mcmillan. Applying sat methods in unbounded symbolic model checking. pages 250–264. Springer-Verlag, 2002.

[61] Otmane Aït Mohamed, Xiaoyu Song, Eduard Cerny, Sofiène Tahar, and Zijian Zhou. Mdg-based state enumeration by retiming and circuit transformation. *Journal of Circuits, Systems, and Computers*, 13(5):1111–1132, 2004.

[62] Matthew W. Moskewicz and Conor F. Madigan. Chaff: Engineering an efficient sat solver. pages 530–535, 2001.

[63] Friedrich Otto and Paliath Narendran. Codes modulo finite monadic string-rewriting systems. *Theor. Comput. Sci.*, 134(1):175–188, 1994.

[64] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5, 1989.

[65] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, Jerusalem, Israel, Israel, 1997. Weizmann Science Press of Israel.

[66] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[67] S. German R. Bryant and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Log.*, 2(1):93–134, 2001.

[68] Owre S., Rushby J.M., and Shankar N. Pvs: a prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, 1992.

[69] A. Goel S. Lahiri, R. Bryant and M. Talupur. Revisiting positive equality,tools and algorithms for the construction and analysis of systems. *LNCS 2988, Springer-Verlag*, pages 1–15, March, 2004.

[70] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.

[71] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *CHARME*, pages 58–70, 2001.

[72] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/ alanmi/abc.

[73] G. Tseitin. On the complexity ofderivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[74] M. Velev. Using automatic case splits and efficient cnf translation to guide a sat-solver when formally verifying out-of-order processors. pages 242–254. Artificial Intelligence and Mathematics, 2004.

[75] M. Velev. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. pages 310–315. ASP-DAC, January 2004.

[76] Miroslav N. Velev and Randal E. Bryant. Tlsim and evc: a term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories. *IJES*, 1(1/2):134–149, 2005.

[77] Ying Xu, Xiaoyu Song, Eduard Cerny, and Otmane Aït Mohamed. Model checking for a first-order temporal logic using multiway decision graphs (mdgs). *Comput. J.*, 47(1):71–84, 2004.

[78] Xu Y. Mdg model checker user's manual. Technical report, 1999.

[79] Zhou Z. Mdg tools v(1.0) user's manual. Technical report, 1996.

[80] Hantao Zhang. Sato: an efficient propositional prover. In *In Proceedings of the International Conference on Automated Deduction*, pages 272–275. Springer-Verlag, 1997.

[81] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs.

[82] Zijian Zhou, Xiaoyu Song, Francisco Corella, Eduard Cerny, and Michel Langevin. Description and verification of rtl designs using multiway decision graphs, 1995.