Deanship of Graduate Studies
Al-Quds University

SAT Solver for Online Model Checking

Mona Musa Nazmi Qanadilo

M.Sc Thesis

Jerusalem – Palestine

1434 / 2013

Deanship of Graduate Studies
Al-Quds University

SAT Solver for Online Model Checking

Mona Musa Nazmi Qanadilo

M.Sc Thesis

Jerusalem – Palestine

1434 / 2013

SAT Solver for Online Model Checking

Prepared By:
Mona Musa Nazmi Qanadilo

B.Sc.: Computer Engineering, 2007, Palestine
An-Najah National University, Palestine

Supervisor: Dr. Sufyan Samara

A thesis submitted in partial fulfillment of requirements for the degree of Master of Electronics and Computer Engineering/ Department of Electronics and Computer Engineering/ Faculty of Engineering/ Graduate Studies
Al-Quds University

1434/2013

Al-Quds University
Deanship of Graduate Studies
Master of Electronics and Computer Engineering

Thesis Approval

SAT Solver for Online Model Checking

Prepared By: Mona Musa Qanadilo

Registration No: 20913395

Supervisor: Prof. Sufyan Samara

Master thesis submitted and accepted, Date: 5/1/2013

The name and signatures of the examining committee members are as follow:

1 – Head of Committee:	Dr. Sufyan Samara	Signature:.....
2 – Internal Examiner:	Dr. Labib Arafeh	Signature:.....
3 – External Examiner:	Dr. Ashraf Armoush	Signature:.....

Jerusalem- Palestine
1434/2013

Dedicated To

Prof. Sufyan Samara

Declaration:

I certify that this thesis submitted for the degree of Master, is the result of my own research, except where otherwise acknowledged, and that this study (or any part of the same) has not been submitted for a higher degree to any other university or institution.

Signed:.....

Mona Musa Nazmi Qanadilo

Date:.....

Abstract

The complexity of software and hardware design has increased significantly over the last years. Clearly, the need for reliable software systems and formal verification techniques is critical to ensure the overall product quality. Hence there is a growing need for more efficient and scalable verification solutions.

SATisfiability based verification techniques have proven to be efficient and scalable for verifying different hardware and software systems. This directed great interests and intense research to improve, utilize, and customize SAT techniques specific to a range of related problems.

In this thesis, we exploit the online model checking specific features. We propose and implement several techniques to optimize SAT solver for online model checking. We introduce an efficient decision strategy for SAT solver specific to online model checking. The objective is to speed up the verification time needed by the online model checker in order to predict potential errors before they have really happened. This is done by a new technique that accepts on-the-fly constraints and performs smart backtracking. Moreover, we take advantage of parallel feature and multiport memory available on FPGA chips. We present a new underlying architecture using two SAT solvers as verification engine for online model checking. We implement a quick prototype of the new underlying architecture for online model checking [1].

Our experimental results show an improved performance over previous techniques. As a result, our techniques achieved significant speed-up for online model checking, by reducing the solving time by 30% compared to other techniques, for some models.

Acknowledgements

First and foremost, all praise is due to ALLAH, the Almighty, and may his peace and blessings be upon the Prophet (peace be upon him), who said: " Should not I be a grateful servant." [1]
I would like to express my deepest gratitude to my supervisor, Prof. Sufyan Samara, for his quiet patience, encouragement, guidance and constant support during the development of this thesis.
I would also like to thank Dr. Yuhong Zhao for her suggestions, and helpful advices.
Many thanks go to Dr. Ashraf Armoush and Dr. Labib Arafeh, for their encouragement, support and helpful advices.
Last but not least, I am grateful to my parents and my family for their patience and *love*. Without them this work would never have come into existence.

Nablus Mona Qanadilo
January, 2013

[1] [Sahih al-Bukhari and Sahih Muslim]

Table of Contents

List of Figures

Chapter 1

Introduction

> *"It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."* [1]

Software and hardware design is getting more and more complex. It is mandatory to analyze and verify them, as errors can have serious consequences. The major challenge is to find an efficient and scalable verification techniques, that spend less effort and time as well as have a large and total coverage.

Formal methods has a great potential to be applied for the verification of complex software and hardware systems. They can be considered as "the applied mathematics for modeling and analyzing systems". Their aim is to establish system correctness with mathematical rigor. Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner [2].

Furthermore, to make these models more precise, mathematical statements can be translated into the language of logic. Propositional logic and its rules can be used to design computer circuits, to construct computer programs, to verify the correctness

[1] H. Barendregt. The quest for correctness.Images of SMC Research, pages 3958, 1996.

of programs, and to build expert systems [3]. Research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. Investigations have shown that formal verification procedures would have revealed the exposed defects in, e.g., the Ariane-5 missile, Mars Pathfinder, Intels Pentium II processor, and the Therac-25 therapy radiation machine [2].

Model checking, one of many formal verification methods, is an automated technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It aims to examine whether or not the execution tree satisfies a user-given property specification. It is a successful technology to verify requirements and design for a variety of systems, particularly in hardware systems and real-time embedded and safety-critical systems.

A major difficulty in applying model checking to a practical design is the state explosion problem, that explores all possible system states. This means that the states of the design to be checked grows rapidly (most of the time exponentially) as the size of the target design increased. Different model checkers use different search algorithms and allow for a variety of different types of behavior to be investigated (e.g. nondeterministic, probabilistic, real time).

Recently, model checking techniques using SATisfiability solving has given promising results[4]. In this thesis, we present several efficient techniques to accelerate online model checking, which might run ahead of or fall behind the execution of the system application. We focus on the following two aspects (1) customizing and optimizing SAT solvers, exploiting online model checking's features, and (2) implementing a new underlying architecture based on FPGA using two SAT solvers working in a pipelined

manner for online model checking.

Ideally, if we can make model checking running fast enough, it is able to predict the potential errors before they really happen [5]. This mainly depends on the complexity of the checking problem, the searching strategy and the underlying hardware architecture. How to speed up online model checking, so that we in any case still have more chance to detect errors before they really happen?

1.1 Objectives

Driven by the demand for more functionality, the complexity of software and hardware design continues to increase. Clearly, the need for reliable software systems and formal verification techniques is crucial in order to ensure of the overall product quality.

Several errors and mistakes are costly. Safety-critical systems such as nuclear power plants or flight control systems may be subject to unexpected runtime errors, which have a huge impact. In such complex systems, subtle errors are extremely difficult to reproduce in a laboratory environment. Therefore, there is a growing need to investigate and develop more robust online verification methods, which may be performed while the system is running.

Online checking technique is necessary complement to the traditional offline checking techniques. Online model checking is a new verification technique, presented in [5]. Errors at the model level might indicate potential errors at the implementation level. Given an abstract model and a concrete implementation of the system to be checked, online model checking [5] aims to ensure the safety of the current execution trace by

online checking a sequence of partial models derived from the current states observed from time to time while the system under test is running.

1.2 Contributions

Using SAT solver as verification engine has been proved to be efficient and scalable for checking both hardware and software systems [6].

Model Checking using satisfiability solving has a substantial success in several industrial fields, which leads to more research in verification methods based on SAT solvers [7].

SAT is related to Boolean SATisfiability Problem which is an important subclass of constraint satisfaction problems, and can formalize a wide range of application problems. It has a wide range of practical applications and is considered a promising technique in artificial intelligence, mathematical logic, and computing theory. An important application of Boolean SATisfiability Problem is Formal Verification, which widely exploits SAT solvers.

In this thesis, we present several techniques that are designed to customize and optimize SAT solvers for online model checking. First, we introduce an efficient decision strategy for SAT solvers that exploits online model checking specific features. The aim is to speed up the execution time of the online model checker so that it predicts potential errors before they really happen. Second, we present a new technique, which takes into account on-the-fly constraints that are observed from time to time while the system under test is running. Finally, these constraints are handled efficiently in a way that leads to smart backtracking. This reduces backtracking in general and

avoid backtracking which cancels the previously traversed search space.

We have implemented our SAT solver for online model checking based on the *Zchaff* [8] SAT solver. For the underling hardware architecture, we have used Field Programmable Gate Arrays which is commonly found in realtime and critical embedded systems. The use of FPGA allows for rapid implementation of parallel computing system. This will give even more speed-up to the SAT solver. We evaluate our heuristic decision strategy by comparing it with other decision strategies for SAT problems taken from the IBM Formal Verification Benchmark [9].

1.3 Thesis Structure

This thesis is structured as follows

Chapter 2 outlines some background knowledge necessary to the context of the subsequent chapters. It briefly reviews system verification techniques, then introduces online model checking and its basic idea. It also introduces the SAT problem, basic SAT algorithms, and the advanced features of modern SAT solvers. It also summarizes relevant related work which includes online model checking and SAT based model checking.

Chapter 3 presents our work and the techniques that are designed to customize and optimize SAT solvers for online model checking. Moreover, it presents the prototype architecture, used for SAT solver for online model checking.

Chapter 4 evaluates the methods and techniques developed in this thesis. It also

reports experimental results showing our decision heuristic is superiority to VSIDS based ones, using IBM model checking benchmark instances.

Chapter 5 presents the hardware system architecture, which implements SAT solver for online model checking on FPGA.

Chapter 6 summarizes and concludes the work done in this thesis. It also presents the future directions which include developing tools for decomposition sat solver to exploit the fine granularity and massive parallelism of FPGAs.

Chapter 2

Background and related work

Verification methods based on SAT solvers is currently enjoying as a promising solution [10]. Dramatic improvements in SAT solver technology have led to great interests and intense research in verification methods based on SAT solver such as online model checking and runtime verification.

In this chapter, a background and related work is presented. In Section 2.1, we briefly review system verification and its technique, then we introduce online model checking and its basic idea. After that, we go through a brief review of the SAT problem, basic SAT algorithms, and the advanced features of modern SAT solvers, as presented in section 2.2. Finally, a summery for relevant work, which includes online model checking and SAT based model checking, is given in section 2.3.

2.1 System Verification

System verification is the process of checking that a system meets specifications and that it satisfies its intended purpose, i.e., verification is "check that we are building the thing right".

The growing need for more efficient and scalable verification solutions have fueled research in verification techniques. As a result, they are being applied to software and hardware systems in a more reliable way. There are several software and hardware verification techniques. This thesis deals with a verification technique called model checking that starts from a formal system specification. Before introducing this technique, we briefly review alternative software and hardware verification techniques.

Software Verification techniques The major software verification techniques are Peer reviewing and testing. They are used so much in practice [2].
A peer review is a way of checking the code, carried out by a team of software engineers. In this technique, the code is not executed, but reviewed and analyzed completely statically.

Software testing is a dynamic technique that actually runs the software and provides it with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. That is to say, testing can only show the presence of errors, not their absence.

Hardware verification techniques There are several techniques used to verify the designs of electronic circuits and the software for micro-controllers, such as emulation, simulation, and structural analysis.
Structural analysis comprises several specific techniques such as synthesis, timing analysis, and equivalence checking.
Emulation is a kind of testing. A software or hardware system (the emulator) is

configured, so that the emulated behavior closely resembles the behavior of the real system. As with software testing, emulation amounts to providing a set of inputs and comparing the outputs with expected ones.

Simulation, is applied on an abstract model of a particular system. Simulation is the most popular hardware verification technique and is used in various design stages, e.g., at register-transfer level, gate and transistor level [2].

2.1.1 Formal Verification and Model Checking

Formal verification is a systematic process, which use mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness. If you have a model of design and some properties that the design is intended to satisfy, you may explore all possible input values, exhaustively. So even the most serious errors that remain undiscovered using emulation, testing and simulation, can be detected. It aims to prove or disprove the correctness of an abstract mathematical model with respect to a certain formal specification or property.

Formal methods has a great potential to be applied for the verification of complex software and hardware systems. Besides, they are one of the highly recommended verification techniques for software development of safety critical systems [2].

For instance, modern e-commerce and communication systems have been verified, deadlocks have been detected in online airline reservation systems and several studies proved the efficiency of this technology, which led to more pervasive in industrial design verification flows [2]. Model checking, one of many formal verification methods, is an automated technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It has been successfully applied to

several information and communication technology systems and their applications. Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner.

It is a successful technology to verify requirements and design for a variety of systems, particularly in hardware systems and real-time embedded and safety-critical systems.

The main weakness of model checking that it suffers from the state-space explosion problem. This means that the states of the design to be checked grows rapidly (most of the time exponentially) as the size of the target design increased. To combat this problem, several research Anl00groups have explored efficient techniques such as, partial order reduction [13], compositional reasoning [3], abstraction technique [10], bounded model checking [4].

Traditional Model checking is defined as an explicit-state model checking. It can perform exhaustive verification in a highly automatic manner. Recent Model checking explores the search space in efficient ways, called Symbolic Model Checking. The first symbolic methods used Binary decision diagram (BDD), a data structure that is used to represent a Boolean function. The state space traversal is based on representations of states sets and transition relations as formulas.

Recently, a new type of model checking technique, bounded model checking (BMC) with satisfiability solving has given promising results [4]. BMC checks if there is a state reachable in k cycles, which satisfies counterexample of a property or not. Dramatic improvements in SAT solver technology over the last decade, make the use of BMC with satisfiability solver, more attractive in industry.

2.1.2 Online Model Checking

Online model checking is a technique to ensure the correctness of the current execution trace. This is done by online checking a sequence of partial models derived from the current states, which are observed from time to time while the system is running.

Simply speaking, online model checking checks the system model against the system specification while using concrete state information observed at runtime to reduce the state space to be explored. Given a state observed at runtime, it is not necessary to search the state space that is not reached from the monitored state [5].

Given a model and an implementation (source code) of a system application to be checked, online model checking aims to check (at the model level) whether the current execution trace (at the implementation level) might run into a predefined unsafe region (error states) or not. Therefore, the property to be checked is limited to Linear Temporal Logic (LTL) formula. For this purpose, online model checking works while the system under test is running.

The current state s_i of the system can be monitored in some way from time to time. The corresponding abstract $\widehat{s_i}$ (if any) can be used by online model checking to reduce the state space to be explored. In other words, online model checking aims to "look" into a near future in the model space, say, next k steps with respect to each abstract state $\widehat{s_i}$ as shown in Fig. 2.1. If no error is detected in the partial model, then the execution trace is safe at least within the next k steps.

It is easy to see that online model checking is not tightly coupled with the system execution. It is worth mentioning the difference between runtime verification and online model checking. Runtime verification focuses on the correctness of the sequence of states (or events) observed or derived at runtime. It has nothing to do with the

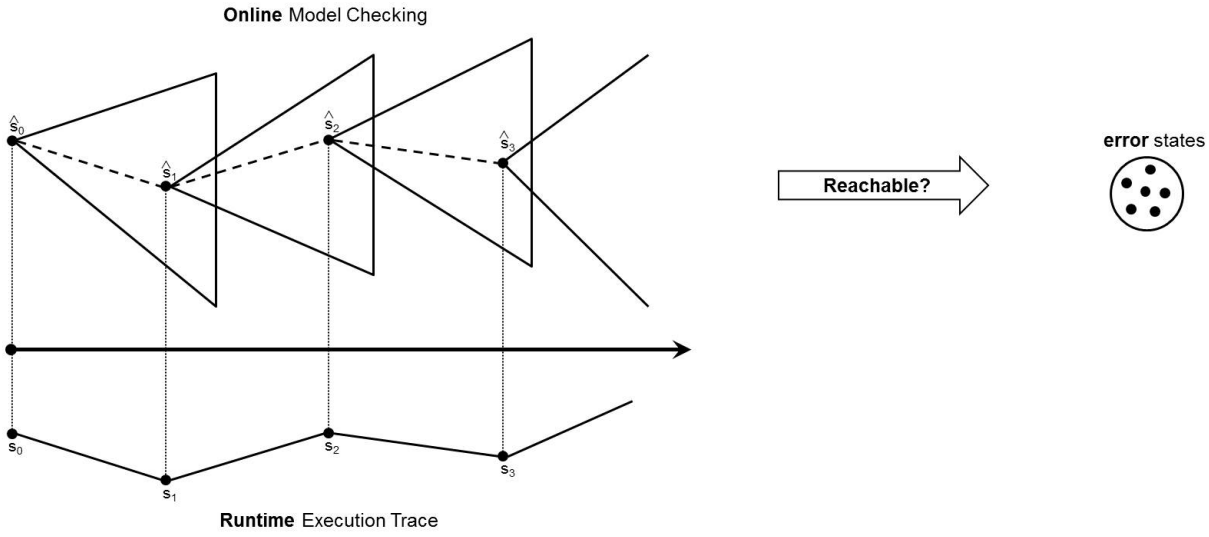

Figure 2.1: Online Model Checking

system model. Runtime verification can proceed further only after a new state (or event) is observed or derived. Therefore, it is hard to detect errors before they have already been occurred. For real-time systems this might be too late to initiate appropriate countermeasures [5]. While, online model checking checks a sequence of partial models derived from the current states, which are observed from time to time while the system is running.

If we can make online model checking run fast enough, then we have more chance to detect subtle errors before they've really happened.

We may use offline backward exploration to reduce the workload of online model checking, using fairness constraints to restrict the behaviour of the design. Each fairness condition specifies a set of states in the state space, and requires that in any acceptable behavior these states must be traversed infinitely. Assume initial unsafe region is F_0 (error states). This region covers those states that satisfy the fairness

condition and that have some loop through them in the meantime.

One can extend this region to be $F' = F_0 \vee F_1 \vee \cdots \vee F_n$ up to n steps from the given error states as shown in Fig. 2.2. Given enough time and memory, it is possible to explore backwards much deeper in the state space of the system model to be checked. Therefore, the workload of online forward exploration will not be so heavy.

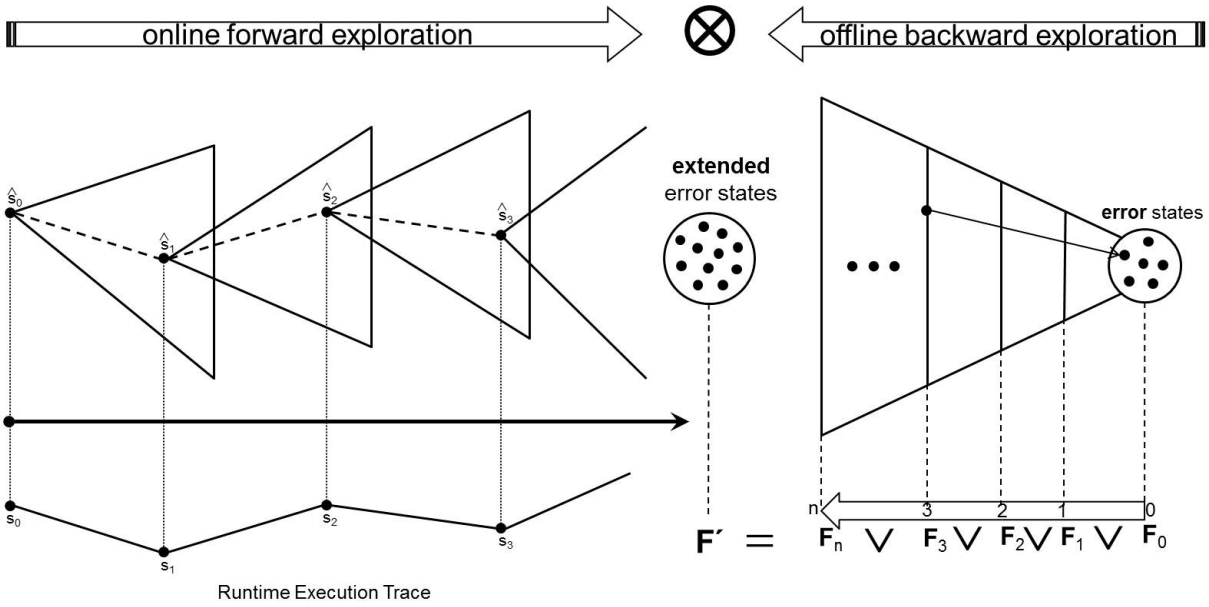

Figure 2.2: Speed up Online Model Checking

As a result, online model checking degenerates into online reachability checking. Many existing efficient solutions to model checking can be directly applied to offline backward exploration.

Modern SAT solver can handle large SAT problems with hundreds of thousands of variables very quickly. Online reachability analysis is a simple form of Bounded Model

Checking (BMC) [6]. By doing BMC at runtime it is quite possible to find deep corner bugs (if any) in the model space. In this thesis, We try to do online reachability checking using SAT solver verification engine.

In [5], the conversion of online model checking problem into a propositional satisfiability (SAT) problem has already generated off-line, as shown in Fig. 2.3.

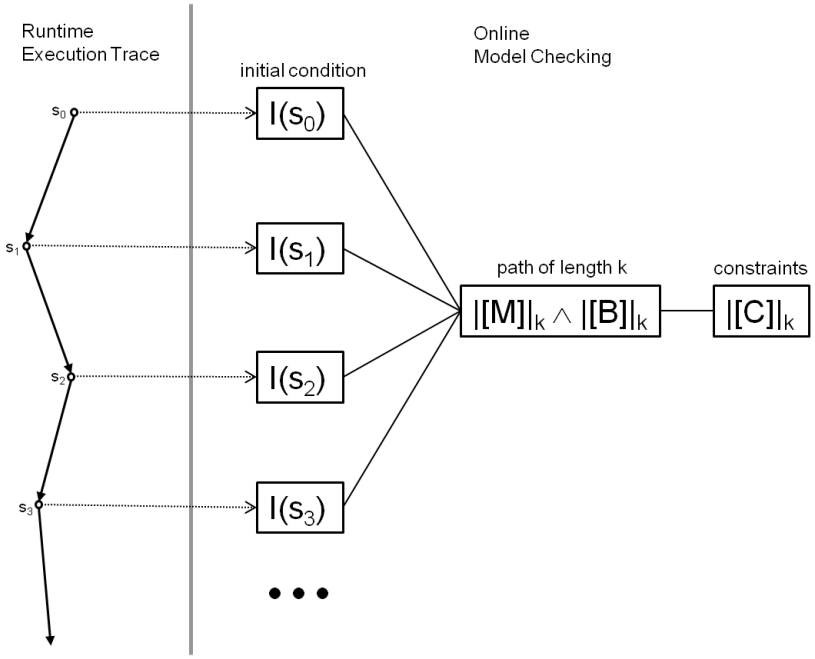

Figure 2.3: Online Symbolic Model Checking

Given the system model M and the *Büchi* automaton B derived from the negation of the LTL formula f, we need to encode online reachability problem into propositional satisfiability (SAT) problem offline in advance. For each abstract state $\widehat{s_i}$ derived from the concrete state s_i observed at runtime, the *Boolean* expression $\mathcal{I}(\widehat{s_i})$ represents the initial condition of the i'th checking round. $|[M]|_k$ is the *Boolean* expression of the paths of length k in M. Similarly, $|[B]|_k$ is the *Boolean* expression of the paths

of length k in B. The *Boolean* expression $|[C]|_k$ represents the the constraint on the product path $|[M]|_k \times |[B]|_k$. Notice that $|[C]|_k$ is derived from the extended unsafe condition F' obtained by offline backward exploration. As a result, the online reachability problem at the i'th checking round is denoted as $|[M, \neg f]|_k^i = \mathcal{I}(\widehat{s}_i) \wedge |[M]|_k \wedge |[B]|_k \wedge |[C]|_k$ as shown in Fig. 2.3.

For simplicity, the length k of the path for online reachability checking at each checking round is fixed to be a predefined small constant. By introducing offline backward exploration, we do not need to set k to be a larger value for online forward exploration. In case of a relatively smaller k, [6] concludes that "SAT based BMC is typically faster in finding bugs compared to BDDs".

Now SAT solver is the key to the performance of the online reachability checking. Let $CNF(|[M, \neg f]|_k^i)$ be the Conjunctive Normal Form of the *Boolean* expression $|[M, \neg f]|_k^i$. For each checking round i, the SAT solver need to answer within the given time limit, whether $CNF(|[M, \neg f]|_k^i)$ is satisfiable or not? If satisfiable, the generated solution indicates an error path (counterexample).

2.2 Boolean Satisfiability (SAT) Problem

2.2.1 Formal Definition of the SAT Problem

SAT is a problem of deciding whether there exists a variable assignment that makes the formula φ evaluate to true. If this assignment exists, then the formula φ is called satisfiable. Otherwise φ is said to be unsatisfiable. The SAT problem is known to be NP complete (it cannot be solved in polynomial time in any known way) [11].

However, much research is done on SAT solvers [10, 12]. This resulted in dramatic improvements in SAT solver technology.

Most SAT solvers use Conjunctive Normal Form (CNF) representation, which is a conjunction of a number of clauses. Each clause is a disjunction of a number of literals. A literal represents either a Boolean variable or its negation. For example, consider the following formula.

$$\varphi = (\bar{x_1} + x_2)(\bar{x_2} + \bar{x_3})(x_1 + \bar{x_2} + x_3)(x_1 + \bar{x_3}) \tag{2.2.1}$$

In the above formula, we have four clauses, three clauses have two literals and one clause has three literals. This formula is satisfied when {x1, x2, x3} = {1, 1, 0}. Note that a CNF formula is satisfied if and only if each clause is satisfied.

2.2.2 Basic SAT Algorithm

Modern SAT solvers are based on the search procedure proposed by Davis-Putnam-Logemann-Loveland (DPLL). These solvers perform a branching search with backtracking. Algorithm 1 shows the basic framework of the DPLL algorithm, adopted from [13].

The algorithm 1 contains 4 main components: Decision Heuristics, Deduction, Conflict Driven Learning, and Backtracking.

The operation of decision heuristics is to select a variable from all free variables and to assign a value for it.

Deduction returns with a set of the necessary variable assignments that can be deduced from the existing variable assignments by the unit literal rule.

Conflict Driven Learning and Backtracking is the procedure to find a reason for a conflict. This reason is decoded as a clause, which may be added to the original

Algorithm 1 *The iterative form of DPLL algorithm*

Require: Model: M.
Ensure: SAT, UNSAT.

```
1: if preprocess() = CONFLICT then
2:     return UNSAT
3: end if
4: while TRUE do
5:     if not decide-next-branch() then
6:         return SAT
7:     end if
8:     while deduce() = CONFLICT do
9:         blevel = analyze-conflict()
10:        if blevel = 0 then
11:            return UNSAT
12:        end if
13:        backtrack (blevel)
14:    end while
15: end while
```

clause database. When a conflict occurs, the solver needs to backtrack and undo decisions.

SAT solvers based on DPLL differ in the implementation of above components.

2.2.3 Variant DPLL SAT Solvers

Modern variants of the DPLL algorithm like Chaff, zChaff [8], BerkMin [14], Minisat [15], Relsat [16] and GRASP [13] received vast attention in the last few years. The continuously improving results have been provided by work on good heuristics and modern techniques such as dynamic selection of the next decision variable [8], non-chronological backtracking [17], conflict driven learning [17], and restart [8].

In our work, we adopt zChaff as a verification solution. zChaff was initially released

in 2001 by Princeton University as a new implementation of the Chaff algorithm. It was awarded as best complete solver for both industrial and handmade categories at SAT 2002 Competition [18].

Its main features include: Two Literal Watching scheme for Boolean constraint propagation, Variable State Independent Decaying Sum (VSIDS) scores for decision making and locality centric decision strategy, non-chronological backtracking with multiple conflict analysis, adoption of a rapid restart policy, and an aggressive clause database management [19].

2.3 Related Work

SAT-based model checking is currently enjoying a substantial success in several industrial fields. Dramatic improvements in SAT solver technology over the last decade have fueled research in verification methods based on SAT solvers. [4] presents a methodology for applying Bounded Model Checking Using Satisfiability Solving in industry for invariance checking. [20] presents tuning the static order decision heuristics over GRASP for SAT procedures in the context of bounded model checking of industrial designs. [21] improves SAT performance for Bounded Model Checking by tuning the VSIDS decision heuristic. There are other related works based on applying SAT for verification techniques as reported in several academic and industrial case studies [22, 23] , other efforts in bounded model checking based on SAT have been applied in both hardware verification [24] and specification logics [25]. In addition, SAT have been used in the formal verification of railway control systems [26].

All previous works belong to off-line model checking, since they can't be done at runtime. Furthermore, Online model checking checks the system model against the

system specification while using concrete state information observed at runtime to reduce the state space to be explored. This work is the first, which customize SAT solver's techniques for online model checking exploiting its own features.

Several research groups have explored efficient techniques to combat model checking problem: partial order reduction [27], compositional reasoning [28], abstraction technique [29], bounded model checking [6], to name just a few. Most of these research aims to prune the search space at the cost of making model checking process to some extent complicated. Any way they all belong to off-line model checking, since they can't be done at runtime.

Chapter 3

SAT Solver For Online Model Checking

We customize and optimize SAT solvers for online model checking. These include an efficient way to minimize backtracking and an efficient decision strategy specific to online model checking. This allows the prediction of potential errors before they are really happened. Moreover, our techniques take into account on-the-fly constraints, which are observed from time to time while the system under test is running.

3.1 Decision Heuristic for Online Model Checking

Decision heuristic is the process of selecting a variable (literal) from the unassigned variables and assigns it a value (either 0 or 1). This process has a major effect on the deduction process efficiency. Selecting good variable may deduce a large set of variable values. These may be added to the original variable assignment as a new current set of assignments, which would result in exploring the search space in less time. Converting a propositional logic expression φ into CNF format $CNF(\varphi)$ usually

needs to introduce many auxiliary variables, which will lead to a large formula with excessive variables. Relative to the auxiliary variables, we say the variables in φ, are dominant variables. Experiments show that assigning a variable from the dominant variables usually can deduce a large set of variable assignments compared to when assigning an auxiliary variable. That is, dominant variables have more influence than auxiliary variables [21].

Giving dominant variables priority over auxiliary variables in the assignment process, we observed a noticeable improvement in performance of the SAT solver. This is done by obtaining runtime information about the system concrete states. This information is used to minimize the state space to be explored. Concrete states represent dominant variables with fixed values and fixed order. Giving dominant variables priority over any other variable in the assignment process has a noticeable effect on the SAT solver performance. The solver improved performance is retained or it may even increase when the given priority in one checking round is kept for the next round. This is more noticeable especially if the concrete states are related to each other.

zChaff is a general SAT solver, which can not differentiate between dominant and auxiliary variables directly from the given CNF formula. It employs Variable State Independent Decaying Sum (VSIDS) [8] as its decision strategy, which is summarized here as follows:

1) Each variable in each polarity has a counter, which is initialized to number of occurrences.

2) When a clause is added to the database, the counter associated with each literal in the clause is incremented by the same constant.

3) The (unassigned) variable whose polarity with the highest counter is chosen at each decision phase.

4) Ties are broken randomly by default, which is also configurable.

5) Periodically, all the counters are divided by a constant.

In addition to the CNF formula, we also provide zChaff with the information about the variables in the CNF formula, so that the SAT solver can distinguish dominant variables from auxiliary variables. We give priority to dominant variables over auxiliary variables. As a result, the second item of zChaff's decision heuristic is modified in the following way:

2) When a clause is added to the database, the counter associated with each literal in the clause is incremented by 2 in case of dominant variable, or by 1 in case of auxiliary variable.

For instance, Fig. 3.1 illustrates our decision strategy. Provided that the variable x_7 is dominant variable, if the clause $x_7 + x_{10} + x_{12}{}'$ is added to the clause database, the new score of x_7 is 5 (instead of 4) according to our decision strategy. In this way, dominant variable will have more chance of being selected than auxiliary variable. The experimental results in Section 4 (Fig. 4.1 and Fig. 4.2) show the performance improvement using this decision strategy.

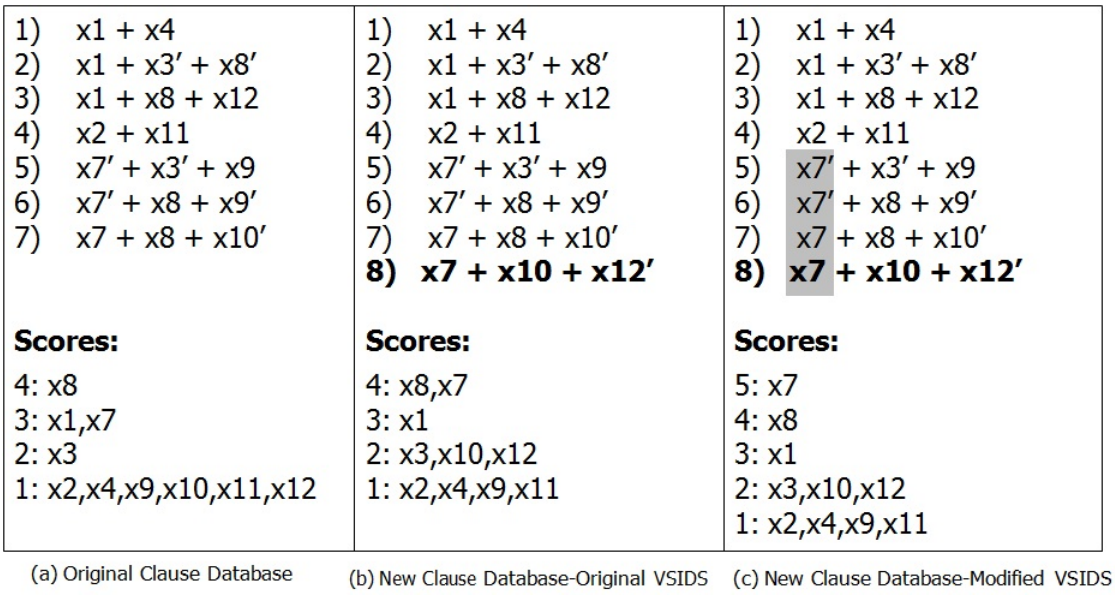

Figure 3.1: Decision Strategy in Favor of Dominant Variables

3.2 On-the-fly constraints and Non-Chronological Backtracking

It is easy to see in Fig. 2.3 that the only difference among the SAT problems of different checking rounds is the initial condition $\mathcal{I}(\widehat{s_i})$ derived from the concrete state s_i observed at runtime.

Without loss of generality, let $V = \{v_1, v_2, \cdots, v_n\}$ be the state variables of the system model. Of course, they are dominant variables. The initial condition indicates the valuation of these state variables. For example, $\mathcal{I}(\widehat{s_i}) = v_1 \wedge \neg v_2 \wedge \cdots \wedge \neg v_n$ means that at the i'th checking round initially we have $v_1 = 1, v_2 = 0, \cdots, v_n = 0$. Notice that each conflict clause learned by the SAT solver is an implication of some clauses of the SAT problem, therefore, it is redundant and has nothing to do with the valuation

of the variables. This means, the conflict clauses learned at previous checking rounds can be directly used at the later checking rounds to reduce the searching space.

From one checking round to the next checking round we do not need to *reset* the SAT solver, which will remove all the learned clauses from the clause database. We'll instead *restart* the SAT solver in an efficient manner. The restart operation in zChaff will simply undo those assignments of variables with decision level greater than 0. Let the state variables $v_1, v_2, \cdots, v_n$ at time frame 0 be ordered in this way. We observe that the initial states $\mathcal{I}(\widehat{s_i})$ and $\mathcal{I}(\widehat{s_{i+1}})$ of any two consecutive checking rounds usually have common part, i.e., the valuation of some variables keep unchanged. Therefore, we can backtrack to the first variable v_j (at decision level j) whose valuation is changed at $\mathcal{I}(\widehat{s_{i+1}})$. Of course, if v_1 is such a variable, then we have to backtrack to v_1 at decision level 1 in this case. However, as long as $j > 1$, we can reuse the deduction results done for $v_1, v_2, \cdots, v_{j-1}$ at the next checking round. In particular, if the initial states $\mathcal{I}(\widehat{s_i})$ and $\mathcal{I}(\widehat{s_{i+1}})$ happen to be the same, then we can simply continue solving as usual.

The example in Fig. 3.2 illustrates our backtracking from one checking round to the next checking round, provided that x_2 and x_7 are the encoding of the variables v_1 and v_2 at time frame 0 respectively. It is worth mentioning that the variables assigned at decision level 0 are deduced at preprocessing phase of the SAT solver, which is independent of any other variable assignments. Therefore, these variable assignments keep always unchanged for each checking round.

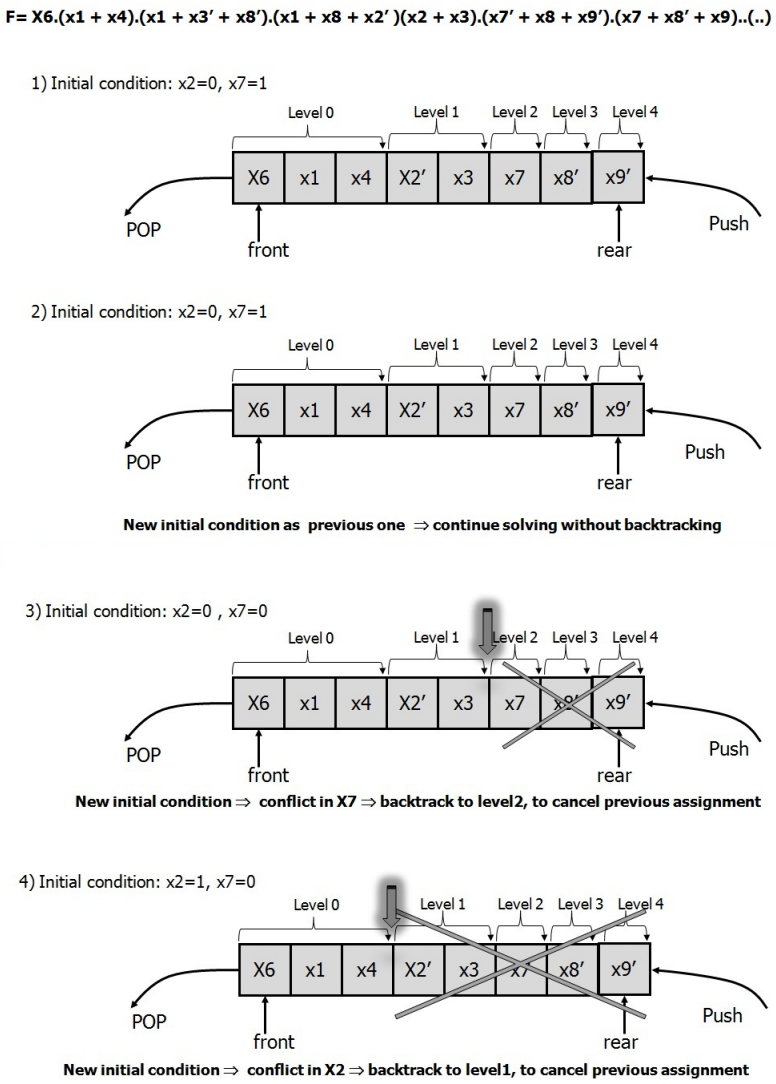

Figure 3.2: Efficient Backtrack

3.3 SAT solver's Output for Online MC

Provided that T is the time limit allocated to online model checking for each checking round, after the initialization at the very beginning, the SAT solver needs to *restart* itself every T time units with a new initial condition $\mathcal{I}(\widehat{s}_i)$. At each checking round,

the SAT solver needs to search a solution starting from the given initial condition in the model space derived from $|[M]|_k \wedge |[B]|_k \wedge |[C]|_k$. This solution (if any), i.e., a path of length k, indicates some error in the system model. Therefore, for the question: is there some error in the system model? The SAT solver will answer "Yes" if a solution is found; "No" if no solution is found; or "Unknown" if the searching process is terminated due to the time constraint. In case of "Unknown", some statistics can be output if required:

- Current Decision Level and Highest Decision Level. The solver reaches high decision levels when it makes a large number of decisions without conflicts, or when the conflicts do not set it back too much. The average of decision level may measure the efficiency of the selected decision heuristic.
- Number of Implications. This parameter may give us probabilities of the model being satisfied. Storing runtime information and comparing it with each other, help to estimate how large state space, is explored.
- Number of Conflicts. This parameter may indicate a number of backtrackings.

Algorithm 2 shows how to include above techniques to implement our SAT solver for online model checking. At the beginning of the algorithm the solver will perform some preprocessing on the instance, performed by function preprocess(), Line 2. This function finds out if the formula's satisfiability can be trivially determined or if some assignments may be made without any branch. If preprocessing cannot determine the outcome of the formula's satisfiability, the main loop begins, Line 6. Getting a state from monitored states -the initial conditions, if this is a first state, then there

is no need to reset (backtrack); no conflict constraints have been added yet.

Otherwise, the algorithm search for lowest decision level for all conflict variables then backtrack to undo wrong assignments up to that level, Lines 11 to 22, see section 3.2, for more details. Then the algorithm assigns (implies) the specific value for every dominant variable in current state, associating every variable with a decision level, Lines 26 to 35.

Next, the function deduce() is called to determine variable assignments that may be deduced from current assigned dominant variable, Line 31. All variables that are assigned as a consequence of the deduction are assumed to have the same decision level as the current dominant variable.

After that, another main loop begins, Line 37. A branch is made on a free variable by assigning it a value and the function deduce() is called, Lines 38 to 41. After the deduction, if all the clauses are satisfied, then the instance is satisfiable, Line 39. Otherwise; if there exists a conflicting clause, then the current branch cannot lead to a satisfying assignment, so the solver will backtrack, Lines 42 to 47. If the instance is neither satisfied nor conflicting under the current variable assignment, the solver will choose another variable to branch and repeat the process.

Backtracking to a decision level less than or equal to the highest level of previous assigned dominant variable, means that the formula is unsatisfiable and there is no error in the model, Lines 43 to 45.

Algorithm 2 *Sat Solver for Online Model Checking*

Require: Abstract Model: M, Dominant Variables: D, Execution trace: E.
Ensure: SAT, UNSAT + Statistics, TIMEOUT.

```
1. decisionLevel ← 0 // decisionLevel: denote the level of assigning the variable.
2. if preprocess() = CONFLICT then
3.  return UNSAT
4. end if
5. firstState ← 1 //firstState: denote if this is first state or not
6. while TRUE do
7.  E[x] ← get a state from mailbox or buffer
8.  n ← literal count in current state, E[x]
9.  t ← timeout limit of a current state
10.  if firstState ≠ 1 then
11.   for j = 0 to n do
12.    literal ← E[x].Literal(j)
13.    if Variable(literal).Value = UNKNOWN then
14.     continue;
15.    else if Variable(literal).Value = 1 & Sign(literal)=0 then
16.     continue;
17.    else if Variable(literal).Value = 0 & Sign(literal)=1 then
18.     continue;
19.    else if Variable(literal).DecisionLevel ≤ decisionLevel then
20.     decisionLevel ← Variable(literal).DecisionLevel
21.    end if
22.   end for
23.   reset(decisionLevel)
24.  end if
25.  firstState ← 0
26.  for j = 0 to n do
27.   decisionLevel ← decisionLevel + 1
28.   literal ← E[x].Literal(j)
29.   if Variable(literal).Value = UNKNOWN then
30.    ImplicationQueue ← literal
31.    if deduce() = CONFLICT then
32.     return UNSAT
33.    end if
34.   end if
35.  end for
36.  L ← decisionLevel
37.  while TRUE do
38.   if not decide-next-branch() then
39.    return SAT
40.   else
41.    while deduce() = CONFLICT do
42.     blevel = analyze-conflict()
43.     if blevel ≤ L + 1 then
44.      backtrack(1)
45.      return UNSAT
46.     end if
47.     backtrack (blevel)
48.    end while
49.   end if
50.  end while
51. end while
```

Chapter 4

Evaluation Methods

This chapter presents the evaluations of the methods and techniques presented in chapter 3. In section 4.1, we evaluate our decision heuristic strategy. The evaluation is done using instances from IBM Formal Verification Benchmark and other instances, are generated for online model checking.

4.1 Evaluating our decision heuristic strategy

4.1.1 Evaluation using standard benchmark

We used IBM Formal Verification Benchmark, described in table 4.1, to evaluate our decision heuristic strategy. All the instances are satisfiable and taken from real industrial hardware designs. They are generated using Bounded Model Checker Software which takes a model, described in SMV[1] language and a bound k.
It is worth mentioning that all 13 instances are for different models. Column 3 and 4 in table 4.1, indicate the number of variables and clauses for each instance, respectively.

[1][SMV is a popular description language in model checking]

Index	SAT Problem	Variables	Clauses
1	bmc-ibm-6	51639	368352
2	bmc-ibm-10	59056	323700
3	bmc-galileo-9	63624	326999
4	bmc-galileo-8	58074	294821
5	bmc-ibm-12	39598	194778
6	bmc-ibm-11	32109	150027
7	bmc-ibm-4	28161	139716
8	bmc-ibm-3	14930	72106
9	bmc-ibm-13	13215	65728
10	bmc-ibm-1	9685	55870
11	bmc-ibm-5	9396	41207
12	bmc-ibm-7	8710	39774
13	bmc-ibm-2	2810	11683

Table 4.1: IBM Formal Verification Benchmark

Since we could not get the original models of the benchmark, we have no way to distinguish dominant variables from auxiliary variables in the CNF representations, therefore, the dominant variables are decided randomly.

We compared our decision heuristic with other decision heuristics: static order and the original VSIDS, taking into account two performance metrics: (i) the runtime as the speed of SAT solver, and (ii) the number of occurred conflicts.

		No of Conflicts			Runtime (s)		
Index	SAT Problem	Orig.VSIDS	Static	Modified VSIDS	Orig.VSIDS	Static	Modified VSIDS
1	bmc-ibm-6	2888	3256	2263	5.789	6.011	4.496
2	bmc-ibm-10	11563	13722	8979	38.074	41.237	27.998
3	bmc-galileo-9	3060	4604	3310	7.128	9.505	7.218
4	bmc-galileo-8	2625	5500	2580	8.159	16.342	7.112
5	bmc-ibm-12	15340	21593	10122	76.917	105.426	47.583
6	bmc-ibm-11	9756	11143	8295	29.724	27.79	24.063
7	bmc-ibm-4	2260	2123	2070	4.099	3.128	2.965
8	bmc-ibm-3	52	1274	52	0.317	1.012	0.272
9	bmc-ibm-13	4044	15536	6443	5.709	25.329	9.205
10	bmc-ibm-1	2316	2379	2533	2.783	2.562	2.841
11	bmc-ibm-5	116	145	93	0.337	0.314	0.249
12	bmc-ibm-7	40	49	40	0.25	0.209	0.182
13	bmc-ibm-2	26	33	25	0.081	0.071	0.066

Table 4.2: Experimental Results for IBM Formal Verification Benchmark

Table 4.2, presents our experimental results. We run zChaff three times for each instance, to compute speedup and conflict occurrences for 3 different decision heuristics. For ten instances, the number of conflicts using our decision heuristics is less compared to other two decision strategies. This would give us a speed up for the SAT solver.

Fig. 4.1 and 4.2, show the preeminence of our decision heuristic over the other two strategies. Our experimental results show that our techniques can deliver substantial performance improvement results; they benefit a reduction in solving time, especially for large models (large CNF formula).

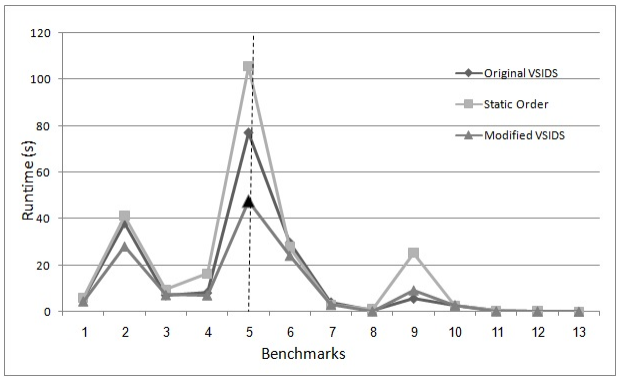

Figure 4.1: Total run time comparisons between 3 different decision heuristics, using IBM Formal Verification Benchmark

Fig. 4.1 shows the total runtime comparison between Static Order, Original VSIDS and Modified VSIDS. In this graph, vertical line represents the SAT solver's runtime. While horizontal line represents the index of IBM benchmark, described in table 4.1. For each instance, we run SAT solver three times, for different decision heuristics. Note that for instances such as 'bmc-ibm-10' and 'bmc-ibm-12', our technique has an improved result compared with other decision heuristics. For example, the runtime of SAT solver using our decision heuristic, for instance with index 5 'bmc-ibm-12', is 47.583 seconds. This would give us a speed up of 40% and 50% compared to original VSIDS and static order, respectively. We previously mentioned that all 13 instances are for different models. So, we can't recognize any relation between the instances.

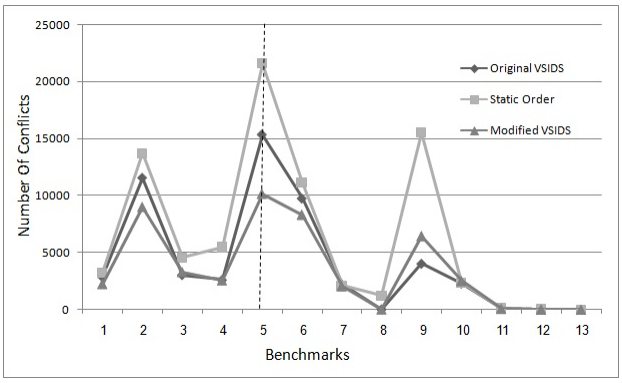

Figure 4.2: Conflict comparison between 3 different decision heuristics, using IBM Formal Verification Benchmark

The same improved result is obtained for the second performance metric- number of conflicts. Fig. 4.2 shows the number of conflicts occurred for every IBM benchmark. It is easy to see that our modified VSIDS decision strategy has better performance than the other two decision heuristics. For example, the number of conflicts of SAT solver using our decision heuristic, for instance with index 5 'bmc-ibm-12', is 10122. This would give us a speed up of 40% and 50% compared to original VSIDS and static order, respectively. Clearly, there is a relation between fig. 4.1 and fig. 4.2. As the number of conflicts increases, the SAT solver's runtime is increased. Hereby, the shape of two graphs are nearly the same.

4.1.2 Evaluation using our online model checking's benchmark

In this section, we evaluated our whole work through instances for online model checking. As for the IBM benchmark, we have neither the original models nor the implementations of these models, therefore we can not use this benchmark to do online model checking.

To test the performance of the SAT solver for online model checking, we take the MSI protocol with transient states" from the NuSMV software package [30], as our case study.

This protocol specifies that "There are three processors, each with one level of cache that stores 1-bit of data and has a 1-bit tag. The caches are write-back, write-allocate. The bus arbitration is round-robin. There is a memory with two 1-bit locations."

From the NuSMV specification of the protocol we can generate the CNF representations of transition relation (without initial condition) and the initial condition respectively. In addition, we can obtain dominant variables as by-product.

By unrolling the transition relation up to k steps we get the CNF representation $|[M]|_k$ of the path of length k of the protocol model. The property to be checked is an invariant, from which we generate the CNF representation of the constraint $|[C]|_k$ on the path of length k. The runtime state information is simply generated by "executing" the model itself.

Therefore, we have $s_i = \widehat{s_i}$. The SAT problem for online model checking is $\mathcal{I}(\widehat{s_i}) \wedge |[M]|_k \wedge |[C]|_k$.

k	Total Variables	Dominant Variables	Total Clauses
10	1962	495	6613
15	2902	720	9823
20	3842	945	13033
25	4782	1170	16243
30	5722	1395	19453
35	6662	1620	22663
40	7602	1845	25873
45	8542	2070	29083
50	9482	2295	32293

Table 4.3: MSI Model with Different Length of Path

We generated 9 instances for k={10, 15, 20, .., 50 }, described in table 4.3. All the instances are unsatisfiable. As K increases so does the SAT problem becomes larger. As we did in section 4.1.1, we take into account the same performance metrics for the 3 different decision strategies. We evaluate our work using the initial current state of the monitored states. The results show an improvement of our decision heuristic in

	No of Conflicts			Runtime (s)		
SAT Problem	Orig.VSIDS	Static	Modified VSIDS	Orig.VSIDS	Static	Modified VSIDS
msi_k10_r20	634	772	631	0.146	0.148	0.158
msi_k15_r20	2491	2337	2039	0.59	0.512	0.51
msi_k20_r20	5150	4912	5917	1.662	1.606	1.886
msi_k25_r20	9834	9586	11658	3.914	3.694	5.586
msi_k30_r20	16977	20154	16722	8.442	11.228	9.411
msi_k35_r20	22812	28872	13288	14.443	17.244	6.594
msi_k40_r20	33329	42129	34511	26.725	36.645	29.784
msi_k45_r20	43878	59901	42488	45.777	73.631	43.531
msi_k50_r20	51072	54945	42020	56.986	68.077	41.554

Table 4.4: Experimental Results for our online model checking's Benchmark

total runtime compared with the other two decision heuristics. The results are shown in the table 4.4.

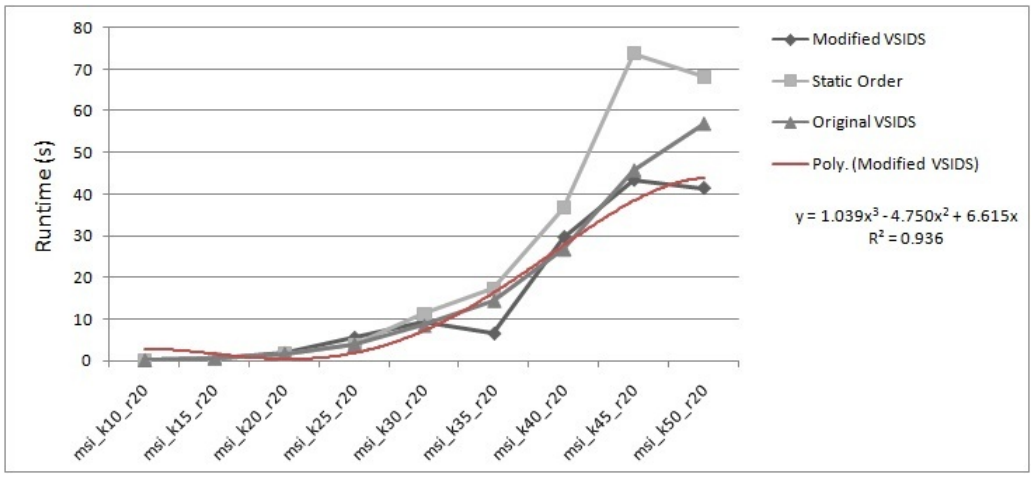

Figure 4.3: Runtime comparison between 3 different decision heuristics, using instances for online model checking

Fig. 4.3 shows the total runtime comparison between the three decision heuristics. Horizontal line represents the name of SAT problems, described in table 4.3. Clearly, as K increases, the runtime of SAT solvers is increased. That's related to the increase in SAT problem (CNF formula). The best fit line for the runtime of SAT solver with modified decision heuristic is found. The relation between K and the runtime is ploynomial of degree three. It is obvious, for large K (e.g. k=50), our SAT solver with modified decision heuristic outperforms the others.

Fig. 4.4 shows the comparison in number of conflicts occurred for different decision heuristics. For every instance, we run SAT solver three times. Each time is for different decision heuristics. Again, the shape of the graphs in fig. 4.3 and fig. 4.4 are the same. That reflects the relation between the runtime and number of conflicts occurred. Hereby, the relation between K and the number of conflics is polynomial of degree three.

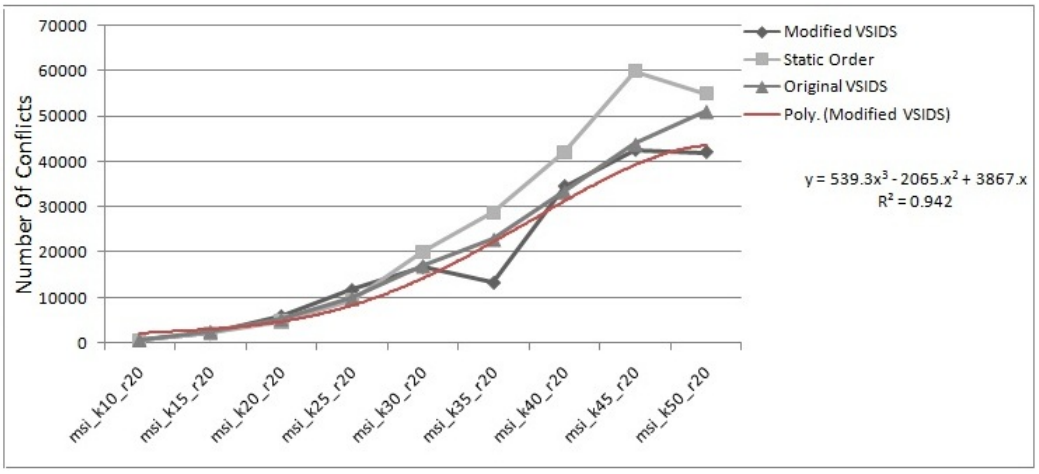

Figure 4.4: Conflict comparison between 3 different decision heuristics, using instances for online model checking

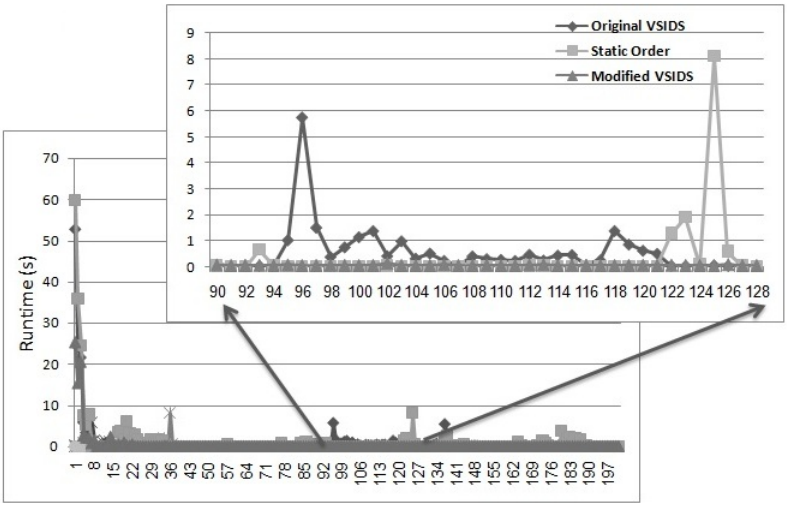

Figure 4.5: Runtime Performance for 201 round checks

Using learnt clauses between different checking rounds reduces the runtime of SAT solver significantly. Fig. 4.5 shows the performance of the 3 SAT solvers for 201 states; each state is the state monitored at runtime. We enlarge the graph for those states between 90 and 128. To be able to compare between the three decision strategies. One may observe that the runtime for modified VSIDS is less and stable compared to other decision heuristics. It is worth mentioning that the runtime for 3 SAT solvers decreases for next and later checking rounds, which is related to the Conflict Driven Learning technique. Keeping learned clauses among different checking rounds enable us to explore state space in less time. All that make the idea of implementing online model checking is more applicable.

Chapter 5

Underlying Hardware Architecture

The prototype underling hardware architecture was built using Field Programmable Gate Arrays (FPGA) which is commonly found in realtime and critical embedded systems. The use of FPGA allows rapid implementation of parallel computing system. This gives even more speed-up to the SAT solver.

In this chapter, we present our underlying hardware architecture and briefly introduce a description for all the products and peripherals, which are used. Then we customize two hardware architectures, one to evaluate our decision heuristic in FPGA and the other to integrate SAT solver and online model checking System, see sections 5.2 and 5.3, respectively.

As an underlying platform, the Xilinx Virtex 5- XUPV5-LX110T Development System kit [31] was used. It incorporates one PowerPC405 processor with the ability to implement soft processor cores, utilizing general-purpose FPGA logic cells.

5.1 FPGA and Underlying Hardware Architecture

With the rapid advancement of the electronic techniques, nowadays many realtime and critical embedded systems are equipped with Field Programmable Gate Arrays (FPGA). Compared to microprocessor, the use of FPGAs for heavy computational tasks is much faster and consumes less power due to the parallel computing feature.

However, in many complex applications, it is convenient to have a processor and FPGA on the same chip. Incorporating the processors and FPGA enables us to implement both serial and parallel algorithm to address the challenges, which we may face in designing today's embedded systems, which must meet ever-growing demands to perform highly complex functions.

There are two types of processor in FPGA: hard and soft cores. Hardcore processor is a dedicated part of the integrated circuit, whereas softcore processor is implemented utilizing general-purpose FPGA logic cells. Many FPGA fabrics, such as Xilinx Virtex family and Atmel FPSLIC family, are mounted with one or more processor cores (e.g. PowerPC or AVR) on a single chip. Others use softcore processors (e.g. NIOS or MicroBlaze) [32].

In our underlying hardware architectures, we used Microblaze, 32-bit soft processor. Microblaze is a part of the Xilinix Embedded Processor Development Kit (EDK), the development package for building MicroBlaze (and PowerPC) embedded processor systems in Xilinx FPGAs.

The MicroBlaze is 32-bit embedded processor (soft core), a reduced instruction set computer (RISC) based engine with 32 register by 32 bit LUT RAM-based Register

file, with separate instruction for data and memory access. It supports both on-chip Block RAM and/ or external memory. MicroBlaze's primary I/O bus, the CoreConnect PLB bus (Processor Local Bus). To access a local-memory (FPGA BRAM), MicroBlaze uses a dedicated LMB bus, which reduces loading on the other buses.

Fig. 5.1 shows a functional block diagram of the MicroBlaze core. To get more details about Microblaze and its features, the reader could read 'MicroBlaze Processor Reference Guide'.

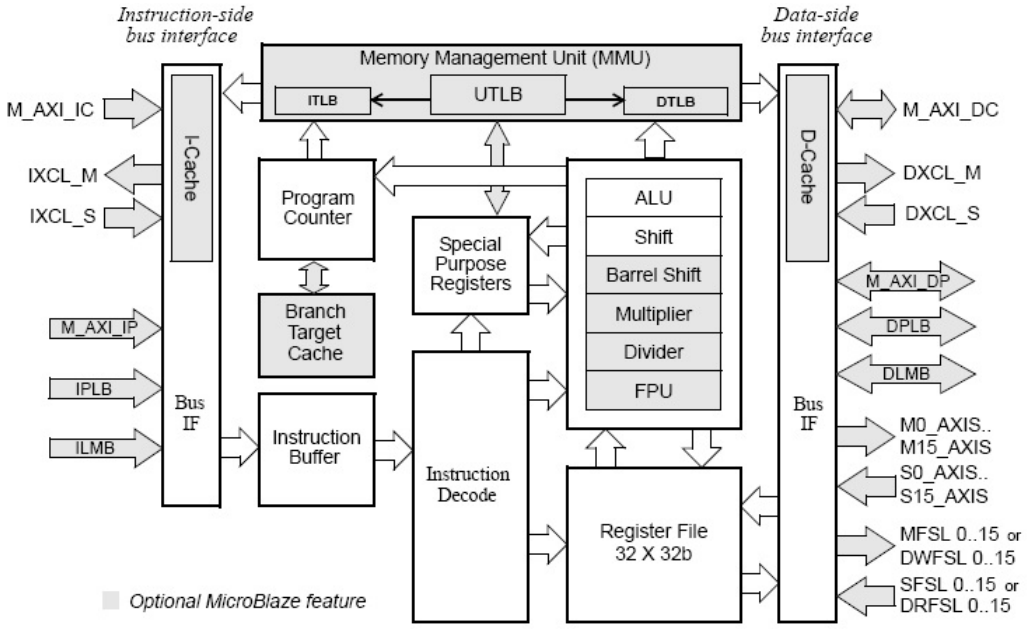

Figure 5.1: MicroBlaze Core Block Diagram

For our underlying hardware architecture, we build four Microblaze processors. Each processor is connected to different peripherals as shown in fig. 5.2. The original block diagram view for our hardware architecture is shown in fig. 5.3, generated using EDK.

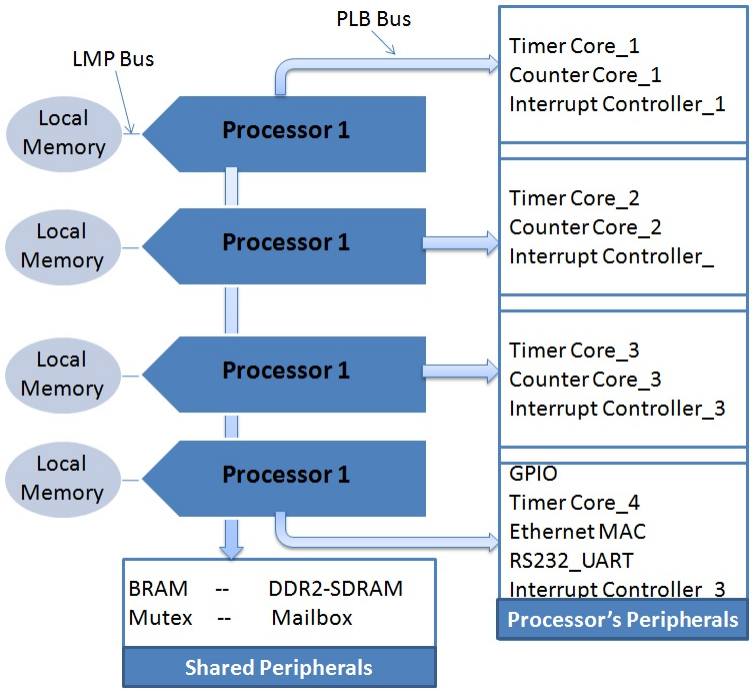

Figure 5.2: Our Underlying Hardware Block Diagram

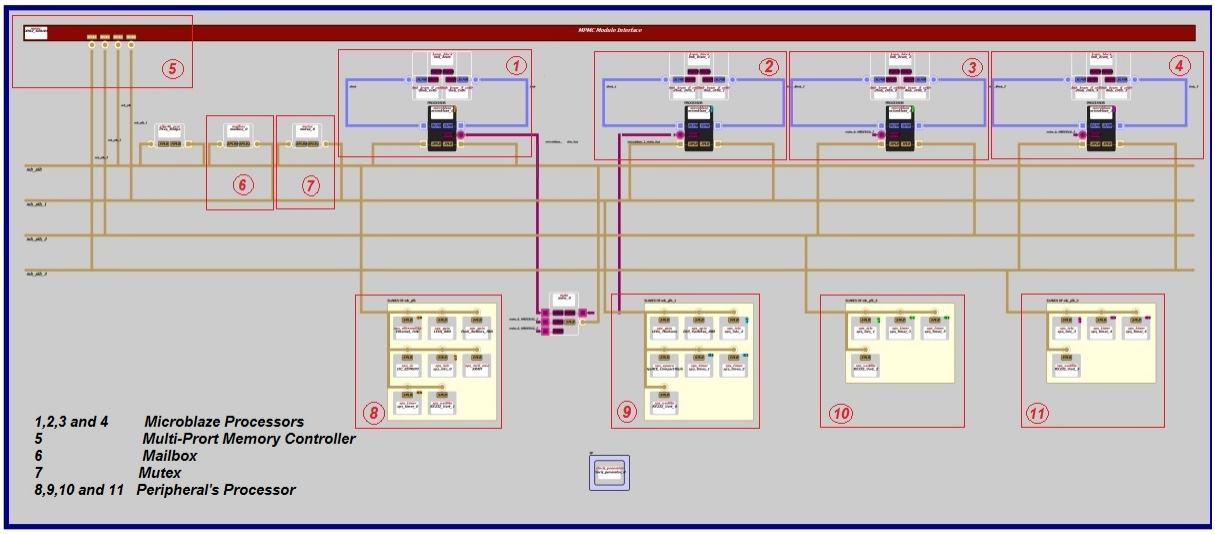

Figure 5.3: EDK Block Diagram View for our Hardware architecture

Fig. 5.4 shows a block diagram of one processor. Each processor has different ports to be connected to local or external memory. We design each processor such that it has a local-memory (BRAM) with two different local memory bus (LMB) interface controllers, for instruction and data memory. The white blocks in fig. 5.4, related to local memory and LMB interface controllers. Moreover, each processor has its own PLB bus and debug bus. A MicroBlaze Debug Module (MDM) is provided, to enable JTAG-based debugging of four MicroBlaze processors.

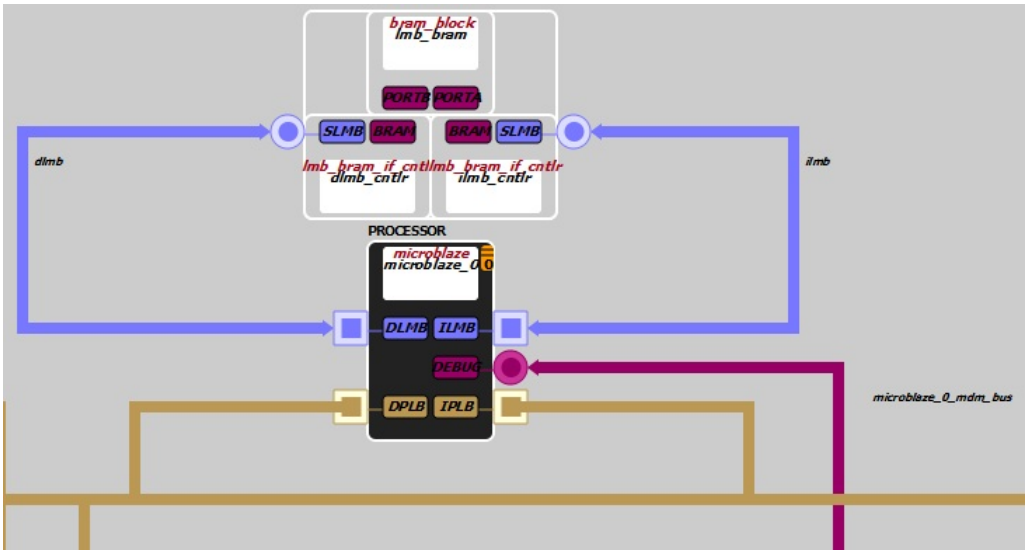

Figure 5.4: Block Diagram of a Processor

The peripherals are connected to one or more processors using PLB bus. Every processor has two timer cores and one interrupt controller core. These peripherals are used to compute the execution time of SAT algorithm. The timer core is 32 bit, used as a counter. Each timer is configured to generate a single interrupt at the expiration of an interval. The clock of timer module is the same as a PLB bus clock.

Other peripherals has more than one bus connection such as mailbox, mutex and multi-port memory controller core (MPMC). Fig. 5.5 shows the connections of these peripherals to PLB buses.

In a multiprocessor environment, our case, the processors need to communicate data with each other. The easiest method is to set up inter-processor communication through a mailbox. Mailbox features a bi-directional communication channel between two processors. Our hardware architecture provide a mailbox between two processors,

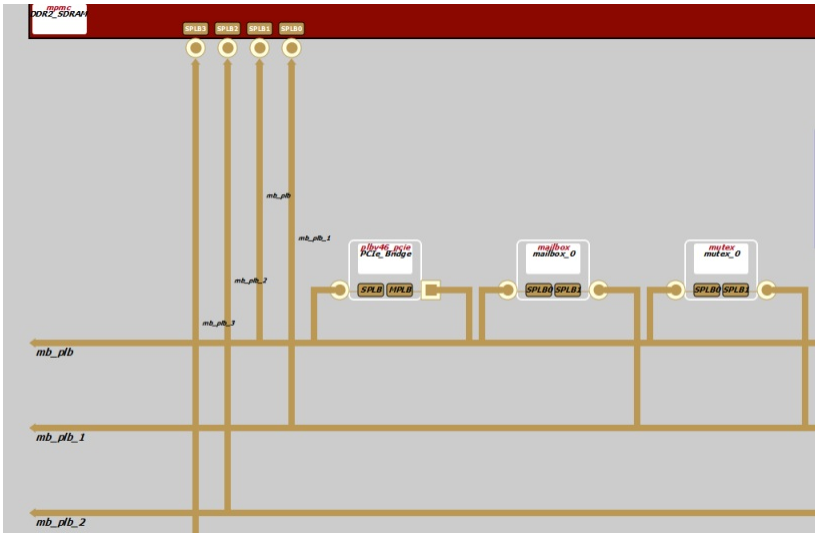

Figure 5.5: Shared Peripherals

to synchronize the communication between two SAT solvers, which are used as a verification engine for online model checking, section 5.3.

We take advantage of parallel feature and multi-port memory available on FPGA chips. We used multi-port memory controller core (MPMC), to provide access to DDR2-SDRAM memory for one to eight ports. This memory is loaded with the system model (CNF formula) and the dominant variables. Hence, DDR2-SDRAM is considered as a shared memory between different processors. The synchronization is done using MPMC, which supports fixed, round robin, and custom arbitration algorithms. Round robin arbitration algorithm is used in our architecture, which gives each port equal priority.

Among the four processor, we used one as a controller for our architecture. It has an Ethernet MAC core, accepts the system model and the dominant variables

through ethernet communication and loads them into DDR2-SDRAM. Moreover, this processor waits for the result from other processors, then it sends them again to the concrete system (application) through ethernet. In addition, this processor is connected to RS232-UART core, so the results could be sent serially.

It is worth mentioning that Xilinix Embedded Processor Development Kit (EDK) consists of two development environment, Xilinix Platform Studio (XPS) and Xilinx Software Development Kit (SDK). XPS is used to create a hardware design. It supports graphical design view, to integrate and configure intellectual property (IP) cores from Xilinix Embedded IP catalog. SDK is a graphical embedded software development environment, is used to complete the design by writing a software application to run on the Microblaze processor. For our first architecture, we have four software applications. One software application configures an ethernet communication of FPGA board. Then it receives the system model and other specifications, to load them on DDR2-SDRAM. Others run SAT solver algorithm. SDK includes GNU C/C++ compiler and debugger, Xilinx Microprocessor Debug (XMD) target server and Data2MEM utility for bitstream loading and updating. We used Data2MEM utility, to load system model and dominant variables as a second choice. This way enables loading data in a few seconds.

5.2 Evaluation's Hardware Architecture

First, we implemented a hardware architecture to evaluate our decision heuristic and test its efficiency compared with other decision heuristics, original VSIDS and static order. We built three soft processor cores in a single FPGA kit, to run three SAT

solvers, each one with a decision heuristic, are made to run in parallel.

Fig. 5.6, shows our architecture. Multi port memory DDR2-SDRAM is used to

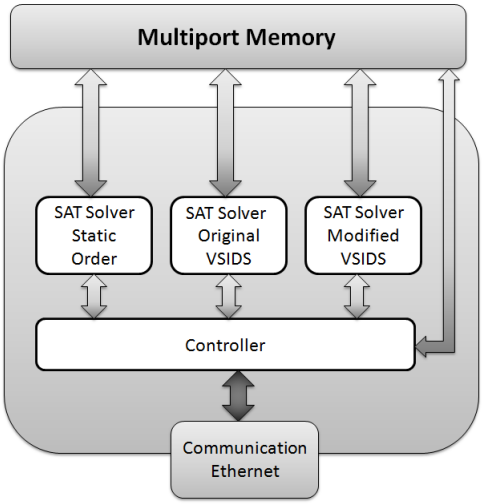

Figure 5.6: Evaluation's Hardware Architecture

load the model, which is abstracted in CNF format.

A controller, which accepts the model through Ethernet, is another soft processor core. The process of solving begins by sending a model to the controller through Ethernet communication. The controller stores the model on DDR2-SDRAM, then it signals the three SAT solvers to start.

The IBM Formal Verification Benchmark was used, described in table 4.1, to test our decision heuristic. The results are shown in section 4.1.

5.3 SAT solver and Online Model Checking's Hardware Architecture

In this section, we present our hardware architecture which integrates our SAT solver and online model checker in a single FPGA. Fig. 5.7, shows the communication between SAT solver, online model checker and the controller.

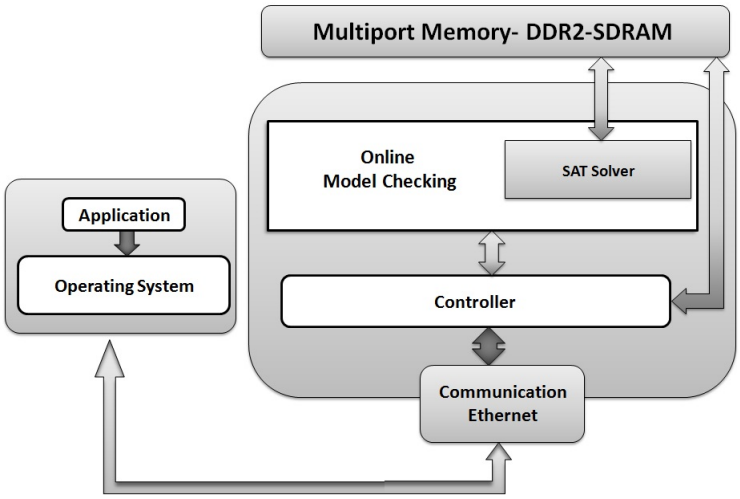

Figure 5.7: SAT solver and Online Model Checking's Hardware Architecture

The system application to be checked and the underlying (realtime) operating system are located in one node, while online model checker is in another node. The two nodes communicate with each other through Ethernet connection. The checking requirement together with the system model, the dominant variables and other necessary data is sent to the controller by the operating system. The controller loads the data into memory and then signals the online model checker to work.

Inside the operating system, there is a special system service named "monitor". The monitor observes the runtime state information s_i of the system execution from time

to time, then maps it to the corresponding abstract state $\widehat{s_i}$ and finally sends $\widehat{s_i}$ to the controller. Accordingly, the controller put each $\widehat{s_i}$ to a (ring) buffer, a specific region in the memory (DDR2-SDRAM). Every T time unit, the SAT solver tries to take a state from the buffer. If there is a state available, the SAT solver goes to search a path (counterexample) starting from this state; otherwise, it either continues the work of the last checking round, provided the work has not finished yet, or simply waits for the next checking round begins. The checking results (Yes/No/Unknown) then is sent to the controller, which in turn sends the results to the operating system. In practice, the controller can inform the operating system only when some error is found by the SAT solver.

To better exploit the parallel feature of FPGA as well as the multi-port memory, we present a new architecture which can support two SAT solvers as verification engine for online model checking as shown in Fig. 5.8.

The system model (CNF formula) and the dominant variables are stored in multi-port memory whose content can be accessed through different ports simultaneously. Each SAT solver is connected to the multi-port memory through Processor Local Bus. The MPMC is responsible for the synchronization with the two SAT solvers. We used Round Robin arbitration algorithm, to give each port equal priority.

The communication between the two SAT solvers and the controller is established using mailbox. The mailbox can be used to generate interrupts between the processors. Taking advantage of this feature, enable us to signal two SAT solvers, to take a state from the buffer. The two SAT solvers work in a similar way as the single solver case but in a pipelined manner. Section 5.4, shows the result for this hardware architecture.

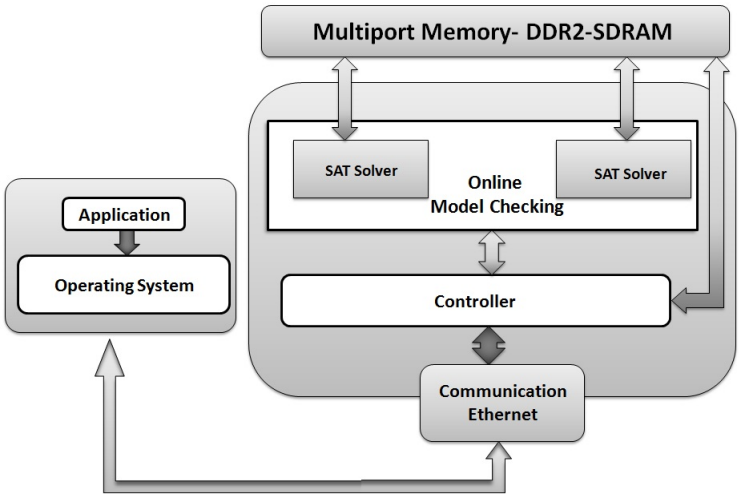

Figure 5.8: New Hardware Architecture

5.4 Experimental Results

We implement a quick prototype of the new hardware architecture for online model checking using two SAT solvers as verification engine on the 64 bit Windows platform with 2.13GHz i3 CPU and 4GB RAM.

The SAT solver is zChaff optimized for online model checking.Here, We take the "MSI protocol with transient states" from the NuSMV software package as our case study. Should we use different decision strategies or use the same decision strategy for the SAT solvers working in parallel? We make the two SAT solvers have different decision strategies: modified VSIDS and original VSIDS. Then, we do online model checking for the MSI protocol with path of length $k = 35, 40, 45$ and 50 from 201 different initial states, i.e., 201 checking rounds.

k	Total Variables	Dominant Variables	Total Clauses
35	6662	1620	22663
40	7602	1845	25873
45	8542	2070	29083
50	9482	2295	32293

Table 5.1: MSI Model with Different Length of Path

Table 5.1 lists the total variables, the dominant variables and the total clauses of the MSI model with different lengths of path. It is clear, increasing k (length of the path), generates CNF representation with more variables and clauses.

In this experiment, we do not set any time limit to each checking round, just let the SAT solvers run to the end and then start the new checking round. From one checking round to the next checking round, the learned clauses are reused by the SAT solvers. If a SAT solver works faster than the other SAT solvers, it will deal with more initial states, i.e., it will run more checking rounds.

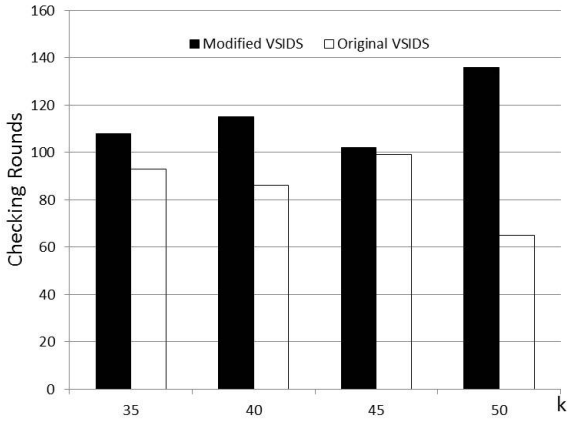

Figure 5.9: Performance Comparison of 2 Decision Strategies

The experimental result in Fig. 5.9, shows that the SAT solver with modified VSIDS decision strategy can run more checking rounds in each case. For each instance (SAT problem), we have 201 initial states, stored in a buffer. The SAT solver, which ends the current checking rounds, takes new initial state. Clearly, a SAT solver with modified VSIDS decision heuristic runs faster than the other. And so, it takes more initial states (more checking rounds).

One may note that for large systems (more variables and clauses e.g. k=50), the SAT solver with modified VSIDS runs better compared to a SAT solver with original VSIDS.

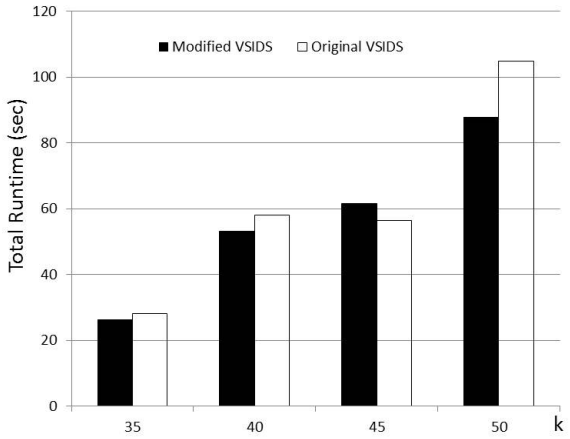

Figure 5.10: Performance Comparison of 2 Decision Strategies

The experimental result in Fig. 5.10, shows the total execution time in seconds of the SAT solvers in each case. Total runtime (Vertical axis) represents the total runtime of all checking rounds, which the SAT solver did. For example, the instance with k=50, the number of checking rounds for SAT solver with modified VSIDS, is 136. While SAT solver with original VSIDS, runs 65 checking rounds. The execution time to run 136 checking rounds, is 87.726 seconds for modified VSIDS and 104.95 seconds for original VSIDS.

The performance of the modified VSIDS decision strategy is better than the VSIDS decision strategies. This result is consistent with the one we've shown in section 4.1. Therefore, we prefer using the same decision strategy (i.e., modified VSIDS) for the SAT solvers to do online model checking. Moreover, this graph shows that increasing K, not only increases the CNF representation or formula, but also makes the system more large and complex. Since it needs more time to run the system model of the path of length k.

How many SAT solvers work in parallel can get the best performance? We've used 1, 2, 3, 4 and 5 SAT solvers respectively to do online model checking for the MSI protocol with path of length $k = 50$ from 201 different initial states, i.e., 201 checking rounds.

In this experiment, we also do not set any time limit to each checking round, just let the SAT solvers run to the end and then start the new checking round.

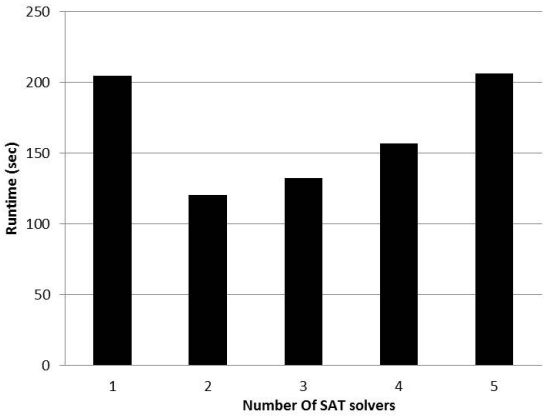

Figure 5.11: Performance Comparison of Multi-SAT Solvers

The experimental result in Fig. 5.11, shows the maximum execution time in seconds of the SAT solver in each case. For example, in the case of 5 SAT solvers, we take the maximum execution time among the 5 SAT solvers. It is easy to see that the two SAT solvers' case outperforms all the other 4 cases. The possible reasons are as follows:

- The more SAT solvers we use, the less states each SAT solver can deal with, thus the less learned clauses are produced to prune the state space for each SAT solver.

- The SAT solver needs to access cache and memory very frequently, therefore, our platform could not simulate the parallel feature of FPGA very well.

From this experiment, we found that the learned clauses have a large impact on the performance of the SAT solver. In the 5 SAT solvers' case, one SAT solver deals with only 8 states, but its total execution time is even more than that of the single solver case, by which 201 states are processed.

It is reasonable to use 2 SAT solvers for online model checking. In addition, we let the 2 SAT solvers share the shortest learned clauses with each other, but the performance improvement is not very satisfying at least for this example. Therefore, we keep the 2 SAT solver independent of each other.

The two SAT solvers work in a pipelined manner as shown in Fig. 5.12. Every T time unit only one SAT solver tries to take a state from the buffer. Each SAT solver has $2T$ time unit to do the search work from the given state. For each SAT solver, if there is a state available, the SAT solver will go to search a path (solution) starting from this state. Otherwise, the SAT solver will resume the search work of the last checking round, provided that the work has not finished yet; or it will wait until the next checking round begins.

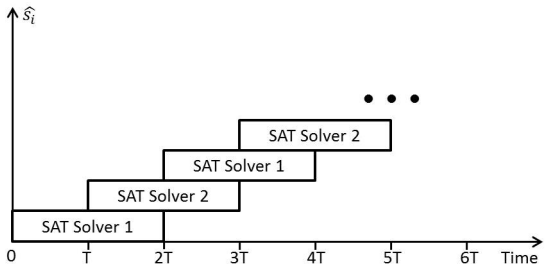

Figure 5.12: 2 SAT solvers work in a pipeline manner

To simulate a general buffer setting, we use a randomly generated Boolean value to decide if a SAT solver can take a state from the buffer or not.

The experimental result in Fig. 5.13, shows the performance of one SAT solver for online model checking the MSI protocol with the path of length $k = 35$ from the 33rd to 65th checking rounds with time limit $2T = 0.08s$ for each checking round.

At the checking rounds 45, 47, 49 and 53 the SAT solver can not take a state from the buffer and the work of the last checking round has been done, therefore, it does nothing and has to wait for the next checking rounds. At the checking rounds 37 upto 40 the SAT solver also can not take a state from the buffer, since the work at the checking round 36 has not finished yet, therefore, the SAT solver resumes the work of the checking round 36. At the checking rounds 44 the SAT solver also has to resume the work of the checking round 43 and finally gets a definite result this time, which means no error path of length $k \leq 35$ starting from the given state is found.

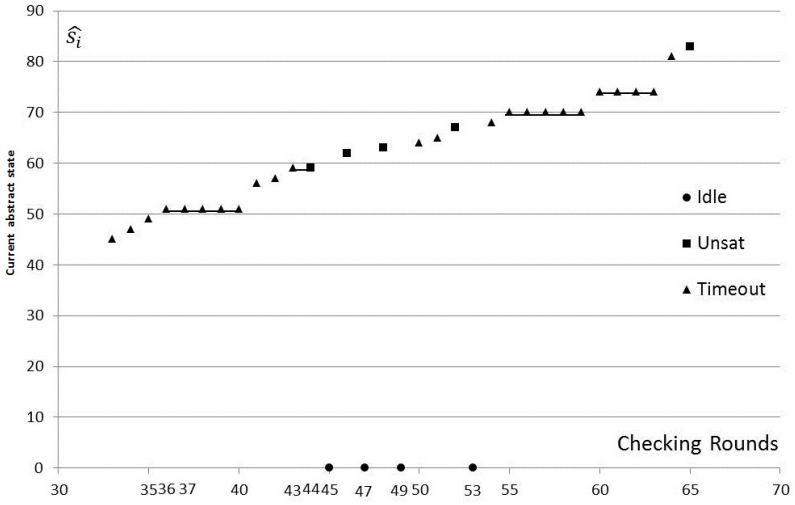

Figure 5.13: Performance of SAT solver with random buffer setting

Chapter 6

Conclusion and future directions

In this chapter, we summarize and conclude the work presented in this thesis. In addition, we outline the directions and the future intended work.

6.1 A brief summary

This thesis presents several techniques for SAT solvers to improve the overall performance of online model checking. We exploit the online model checking specific features. We propose and implement several techniques to optimize SAT solver for online model checking. We introduce an efficient decision strategy for SAT solver specific to online model checking. The objective is to speed up the verification time needed by the online model checker in order to predict potential errors before they have really happened. This is done by a new technique that accepts on-the-fly constraints and performs smart backtracking.

Moreover, we present a new underlying architecture based on FPGA using two SAT solvers working in a pipelined manner for online model checking.

We show that our techniques improve the performance of SAT solvers by evaluating our approach on a standardized Formal Verification Benchmark and other instances,

are generated for online model checking. We observed that a wider and a large model of benchmark would assess our work.

Our experimental results show an improved performance over previous techniques. As a result, our techniques achieved significant speed-up for online model checking, by reducing the solving time by 30% compared to other techniques, for some models.

6.2 Future directions

To achieve significant increase in performance of online model checking, we may use the idea of decomposition and partitioning of SAT problem. We may decompose SAT problem into independent sub-problems to run them in parallel. The idea of partitioning and decomposition have been investigated in the context of constraint satisfaction problems [33].
Recently, such approaches have also found application in SAT solver [34, 35, 36].

We may use hypergraph partition tool hMeTiS[37], to decompose SAT problem which is converted into a hyper-graph,where variables are represented as hyper-graph edges and clauses are represented as vertices.

Moreover, we may implement a reconfigurable hardware sat solver to exploit the fine granularity and massive parallelism of FPGA. FPGAs have gained a large popularity in accelerating satisfiability problem. Recently, several hardware architectures have been proposed, to accelerate SAT solving using reconfigurable computing [38, 39, 40, 41].

However, the first work in accelerating model checking, is represented in [42, 43]. They presented a pipelined hardware accelerated explicit-state model checker.

While our work is focused on building an online symbolic model checker based on satisfiability. Consequently, the next step is to implement a new hardware architecture by integrating different techniques together, such as decomposition, pipelined hardware SAT solver and online model checking.

We may decompose the abstract model into independent models (partitions) that can be processed in parallel, to exploit the fine granularity and massive parallelism of FPGA. Where we may map each partition to different processing element (or processor); we benefit to reduce the local memory for each processing element (or processor). At the same time, each processing element keeps the useful learnt clauses which related to its partition.

Bibliography

[1] M. Q. Sufyan Samara, Yuhong Zhao, "Accelerating online model checking," *Submitted and Accepted to be appeared at Sixth Latin-American Symposium on Dependable Computing Conference (LADC)*, Apr 2013.

[2] K. Baier, *Principles of Model Checking*. The MIT Press, Cambridge, Massachusetts, 2008.

[3] K. H. Rosen, *Discrete mathematics and its applications*. McGraw-Hill, 2012.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Form. Methods Syst. Des.*, vol. 19, pp. 7–34, July 2001.

[5] Y. Zhao and F. Rammig, "Online model checking for dependable real-time systems," in *15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, (Shenzhen, China), pp. 154–161, IEEE Computer Society, April 2012.

[6] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[7] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Integrating bdd-based and sat-based symbolic model checking," in *Frontiers of Combining Systems* (A. Armando, ed.), vol. 2309 of *Lecture Notes in Computer Science*, pp. 265–276, Springer Berlin / Heidelberg, 2002.

[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, DAC '01, (New York, NY, USA), pp. 530–535, ACM, 2001.

[9] http://www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/BMC/description.html, "Ibm formal verification benchmark,"

[10] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, pp. 156–173, 2005.

[11] J. Garey, *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman; First Edition edition, 1979.

[12] L. Zhang and S. Malik, "The quest for efficient boolean satisfiability solvers," in *Computer Aided Verification* (E. Brinksma and K. Larsen, eds.), vol. 2404 of *Lecture Notes in Computer Science*, pp. 641–653, Springer Berlin / Heidelberg, 2002.

[13] J. a. P. M. Silva and K. A. Sakallah, "Grasp -a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, (Washington, DC, USA), pp. 220–227, IEEE Computer Society, 1996.

[14] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discrete Appl. Math.*, vol. 155, pp. 1549–1561, June 2007.

[15] N. En and N. Srensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of *Lecture Notes in Computer Science*, pp. 333–336, Springer Berlin / Heidelberg, 2004.

[16] R. J. Bayardo, Jr. and J. D. Pehoushek, "Counting models using connected components," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 157–162, AAAI Press, 2000.

[17] J. Marques-Silva and K. Sakallah, "Grasp: a search algorithm for propositional satisfiability," *Computers, IEEE Transactions on*, vol. 48, pp. 506 –521, may 1999.

[18] D. Berre and L. Simon, "The essentials of the sat 2003 competition," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), vol. 2919 of *Lecture Notes in Computer Science*, pp. 452–467, Springer Berlin Heidelberg, 2004.

[19] E. Zarpas, M. Roveri, A. Cimatti, K. Winkelmann, RaikBrinkmann, YakovNovikov, OhadShacham, and M. Farkash, "Improved decision heuristics for highperformance sat based static property checking," tech. rep., 2004.

[20] O. Shtrichman, "Tuning sat checkers for bounded model checking," in *Computer Aided Verification* (E. Emerson and A. Sistla, eds.), vol. 1855 of *Lecture Notes in Computer Science*, pp. 480–494, Springer Berlin / Heidelberg, 2000.

[21] O. Shacham and E. Zarpas, "Tuning the vsids decision heuristic for bounded model checking," in *Microprocessor Test and Verification: Common Challenges and Solutions, 2003. Proceedings. 4th International Workshop on*, pp. 75 – 79, may 2003.

[22] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a powerpc microprocessor using symbolic model checking without bdds," in *Computer Aided Verification* (N. Halbwachs and D. Peled, eds.), vol. 1633 of *Lecture Notes in Computer Science*, pp. 686–686, Springer Berlin / Heidelberg, 1999.

[23] P. Bjesse, T. Leonard, and A. Mokkedem, "Finding bugs in an alpha microprocessor using satisfiability solvers," in *Computer Aided Verification* (G. Berry, H. Comon, and A. Finkel, eds.), vol. 2102 of *Lecture Notes in Computer Science*, pp. 454–464, Springer Berlin / Heidelberg, 2001.

[24] L. Guerra e Silva, L. M. Silveira, and J. Marques-Silva, "Algorithms for solving boolean satisfiability in combinational circuits," in *Proceedings of the conference on Design, automation and test in Europe*, DATE '99, (New York, NY, USA), ACM, 1999.

[25] O. Grumberg and D. E. Long, "Model Checking and Modular Verification," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 843–872, 1994.

[26] A. Borälv, "The industrial success of verification tools based on stalmarck's method," in *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, (London, UK, UK), pp. 7–10, Springer-Verlag, 1997.

[27] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996. Foreword By-Wolper, Pierre.

[28] S. Berezin, S. V. A. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, (London, UK), pp. 81–102, Springer-Verlag, 1998.

[29] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[30] http://nusmv.fbk.eu/, "Nusmv software package,"

[31] http://www.xilinx.com/, "All programmable technologies from xilinix,"

[32] S. L. M. Samara, *Adaptable OS Services for Distributed Reconfigurable Systems on Chip*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn, 2010.

[33] R. Dechter and J. Pearl, "Network-based heuristics for constraint-satisfaction problems," *Artificial Intelligence*, vol. 34, no. 1, pp. 1 – 38, 1987.

[34] E. Amir and S. Mcllraith, "Solving satisfiability using decomposition and the most constrained subproblem (preliminary report)," *Electronic Notes in Discrete Mathematics*, vol. 9, no. 0, pp. 329 – 343, 2001. ¡ce:title¿LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)¡/ce:title¿.

[35] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-based decision heuristics for image computation using sat and bdds," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, (Piscataway, NJ, USA), pp. 286–292, IEEE Press, 2001.

[36] F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Mince: A static global variable-ordering for sat and bdd," 2001.

[37] http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview, "hmetis - hypergraph and circuit partitioning,"

[38] M. Safar, M. El-Kharashi, M. Shalan, and A. Salem, "A reconfigurable five-stage pipelined sat solver," in *Microprocessor Test and Verification (MTV), 2009 10th International Workshop on*, pp. 95 –100, dec. 2009.

[39]

[40] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "Designing an efficient hardware implication accelerator for sat solving," in *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, (Berlin, Heidelberg), pp. 48–62, Springer-Verlag, 2008.

[41] T. Pagarani, F. Kocan, D. Saab, and J. Abraham, "Parallel and scalable architecture for solving satisfiability on reconfigurable fpga," in *Custom Integrated Circuits Conference, 2000. CICC. Proceedings of the IEEE 2000*, pp. 147 –150, 2000.

[42] M. Fuess, M. Leeser, and T. Leonard, "An fpga implementation of explicit-state model checking," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pp. 119 –126, april 2008.

[43] M. E. Tie and M. Leeser, "Implementing murf: Accelerating large state space exploration on fpgas," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, p. 243, 2012.

SAT Solver for Online Model Checking

إعداد: منى موسى نظمي قناديلو
إشراف: الدكتور سفيان سماره

ملخص الرسالة

مما لا شك فيه أن عصرنا الحالي هو عصر التكنولوجيا والحاسوب الذي امتدت تطبيقاته لتشمل كافة مناحي الحياة في هذا العصر. و بسبب تعدد هذه التطبيقات و سرعة تطورها فقد أصبح من الضروري التركيز على الوسائل و التقنيات التي تضمن الوصول إلى النتيجة المبتغاة من هذه التطبيقات و مراقبة الجودة المطلوبة.
و قد اتبعت تقنيات تقليدية لاختبار البرمجيات المختلفة والتحقق من ملاءمتها للاحتياجات المطلوبة منها وكذلك خلوها من الاخطاء. إلا أن هذه طريقة عمل التقنيات التقليدية لا يمكن أن تضمن خلو البرنامج أو النظام من الاخطاء نهائيا.

و مع ازدياد تعقيدات تطبيقات البرمجيات الحاسوبية واعتماد كثير منها على العمل عبر شبكات العمل و الشبكة العنكبوتية فقد أصبح من الضروري القيام بفحص هذه التطبيقات بتقنيات حديثة تكون أكثر كفاءة في اثبات صحة البرمجيات و صلاحيتها و من أبرز هذه التقنيات الحديثة هي تقنية فحص النموذج (Model Checking)، التي يتم فيها بناء نموذج للنظام الحقيقي و التحقق ما اذا كان هذا النموذج يقابل مواصفات معينة بشكل تلقائي و شامل.

إلا أن هذه الطريقة في الفحص (Model Checking) قد لا تعمل مباشرة و بشكل متواصل أثناء العمل الفعلي للنظام أو البرنامج. و للحصول على أفضل النتائج فقد تم التركيز في هذه الرسالة على الاستفادة من نظام فحص النموذج بشكل متواصل مع عمل النظام الحقيقي، دون الحاجة الى ايقافه (Online Model Checking). حيث يتم في هذه الطريقة مراقبة النظام الحقيقي من وقت لاخر للاستفادة من توجيه نظام فحص النموذج، الذي يبدأ بالتحقق من صلاحية النظام ابتداء من المعلومات الحقيقية و بتغطية جزء من نطاق النظام.

إن عملية تسريع نظام فحص النموذج يتيح الفرصة لاكتشاف الاخطاء قبل وقوعها، لذا فقد تم في هذه الرسالة استخدام الحلول المتوفرة لحل مشكلة Satisfiability Problem (SAT) مع الاستفادة من سمات نظام فحص النموذج (Online Model Checking). حيث قمنا في هذه الدراسة بتعديل استراتيجية تحديد القرار التي يتم فيها اختيار المتغير في نظام SAT solver وذلك لإعطاء الاولوية للمتغيرات الحقيقية التي يتم استقبالها من النظام أثناء عمله الفعلي. بالاضافة الى تعديل نظام (SAT solver) لاستقبال مواصفات و معلومات جديدة للنظام الحقيقي من وقت لاخر، مع ضمان تقليل نسبة التراجع و الخطأ في اختيار المتغير.

بالاضافة الى ذلك, تم بناء هذا النظام على Field Programmable Gate Array (FPGA) للاستفادة من امكانية تشغيل أكثر من نموذج فحص في نفس الوقت. كل نموذج فحص يقوم بتغطية جزء من نطاق النظام. وذلك ببناء نظام يتكون من معالجين لتنفيذ نظامين فحص في أن واحد.

و بفضل الله , أظهرت نتائجنا تحسن في اداء نظام الفحص مقارنة مع التقنيات السابقة. حيث تم تسريع نظام الفحص و تقليل المدة الزمنية المستغرقة في تنفيذ هذه النظام بنسبة 30% لبعض الانظمة و البرامج .