Copyright

by

Vivekananda Murthy Vedula

2003

The Dissertation Committee for Vivekananda Murthy Vedula
certifies that this is the approved version of the following dissertation:

HDL Slicing for Verification and Test

Committee:

Jacob A. Abraham, Supervisor

Tony P. Ambler

Adnan Aziz

Craig M. Chase

Raghuram S. Tupuri

HDL Slicing for Verification and Test

by

Vivekananda Murthy Vedula, B.Tech., M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2003

Dedicated to my parents

who were my foremost teachers

Acknowledgments

This work has been made possible by the ideas, insights, support and sacrifices of several individuals. Borrowing a line from a *Telugu* song composed by the legendary saint-composer Thyagaraja, *Endaro Mahaanubhaavulu; Andariki Vandanamulu*, meaning "So many noble hearted people in this world; Salutations to you all".

I would like to express my deepest gratitude to my thesis advisor, Prof. Abraham, for his invaluable guidance and encouragement. His novel ideas and deep understanding of research has been a guiding force for this work. I am deeply indebted to him for helping me change my major to Computer Engineering, and providing me with financial support throughout my stay at the University. A major motivating factor for my research has been his insistence on the development of methodology, theory and tools that could be applied to problems in the "real" world.

I am thankful to my committee members for their constructive criticism and suggestions, and steering me on the course for the completion of this dissertation. As my committee chairman, Prof. Ambler has been instrumental in helping me understand the goals and expectations of the committee. Prof. Chase had pointed me to the related research in software, which led me develop the theoretical basis

for my work. Prof. Aziz's class in Formal Verification kindled my interest in this subject, and I deeply appreciate his willingness to discuss my research ever so often and offering insightful comments. I thank Dr. Tupuri, who laid the foundation for a significant portion of this work, for helping me understand and develop my ideas for this work.

I would like to offer special thanks to two people, who supported me both personally and professionally. My wife, Madhavi, whose love and constant encouragement helped me tide over many difficult phases during my research. She had taken special interest in understanding my research, and provided me with vital technical feedback for many of my key ideas. The other person is my close friend, Randy, who had helped me selflessly right through my graduate studies. He guided me at every step in changing my major to Computer Engineering, and was always there for me when I needed his help with my course-work and research. Most of my technical publications would have never seen the light of the day without both of their help in proof-reading and technical writing, many times at a moment's notice. I owe a lot to these two very special people.

I wish to acknowledge the technical help that I had received from several people at various stages of dissertation: Jayantha Bhadra, Motorola; Flemming Andersen and his colleagues at the Strategic CAD Labs, Intel; Jeff Russell and Arun Krishnamachary, The University of Texas at Austin; Prof. Daniel Saab, Case Western Reserve University; Jason Baumgartner, IBM. I was also fortunate to have had stimulating discussions with my current and former colleagues at CERC and UT: Victor, Karthik, Satish, Kamal, Suresh, Vaibhav, Abhijit, Ravi, Navin, Ramesh, Krishna, Bala, Padmini, Hak-soo, Sungbae, Kyoil, Whitney, and Adam. Special thanks to the last two for proof-reading parts of this dissertation.

I am grateful for the financial support extended to me by several funding

agencies: Semiconductor Research Corporation, State of Texas Advanced Technology Program, Intel Corporation, National Semiconductor Corporation and IBM. I am thankful to Linda, Shirley, Ruth and Andrew for their administrative/IT support, and making my life at CERC a wonderful experience.

I was also helped by Prof. C. H. Roth, Prof. M. F. Jacome, Melanie and Prof. K. E. Gray, all from UT, to settle down during my early years of my graduate studies. I would have probably lost my sanity, if not for the many friends (especially, the "Rio-Gang") that I made during my stay in Austin, who brought me cheer during my stressful times during my studies.

It is beyond words to express my gratitude to my family members, who have been immensely responsible for my success. Their constant love and encouragement has been an integral part of my life. I also wish to thank my teachers, at various schools and colleges that I had attended in India, for nurturing my thoughts and making me want to learn more.

VIVEKANANDA MURTHY VEDULA

The University of Texas at Austin
May 2003

HDL Slicing for Verification and Test

Publication No. _____

Vivekananda Murthy Vedula, Ph.D.

The University of Texas at Austin, 2003

Supervisor: Jacob A. Abraham

The semiconductor industry has been increasingly relying on computer-aided design (CAD) tools in order to meet its demand for high performance and stringent time-to-market requirements. However, practical application of state-of-the-art CAD tools is severely limited by the sheer size of the design sizes. Therefore, an appropriate methodology that exploits the inherent modular structure within the complex designs, is desired. This dissertation proposes such a methodology that is useful with a variety of CAD tools in design verification and manufacturing test generation.

Functional test generation using sequential automatic test pattern generation (ATPG) tools is extremely computation intensive and produces acceptable results only on relatively small designs. Therefore, hierarchical approaches are necessary to reduce the ATPG complexity. A promising approach was previously proposed in which individual modules in a design are targeted one at a time, using an ad-hoc abstraction for the reminder of the design derived from its register-transfer level

(RTL) model. Based on this approach, an elegant and a systematic approach based on “program slicing”, that allows it to be scalable for large designs, is developed. The theoretical basis for applying program slicing on hardware description languages (HDLs) is established, and a tool called FACTOR has been implemented to automate the approach for test generation and testability analysis.

Design verification requires exploring the complete design space to ensure the correctness of the design. A proof-by-contradiction approach called bounded model checking (BMC) has been proposed, which utilizes satisfiability (SAT) capabilities to find counterexamples for temporal properties within a specified number of time steps. The proposed scheme harnesses the power of sequential-ATPG tools to use structural information of a hardware design, to perform BMC more efficiently. This approach has been further augmented by the HDL slicing methodology for test generation, to accelerate the verification methodology.

Symbolic simulation uses symbols rather than actual values for simulating a hardware design, so that the responses to a class of values can be computed and checked for correctness in a single run. The effectiveness of this approach has been incorporated into a powerful verification methodology, called symbolic trajectory evaluation (STE), to verify properties of bounded state sequences, intermixed with properties of invariant behavior. Assertions are described in a limited form of temporal logic and are symbolically validated against the design under verification. The HDL slicing tool, FACTOR, has been appropriately applied to speed up the verification of the floating point adder-subtractor unit of the *Pentium 4* design in Intel’s Forte verification framework.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current advances in integrated circuit (IC) manufacturing technology allow designers to incorporate large amounts of functionality onto a single die. The resulting increases in the design complexity and the demand for shorter design cycles has necessitated the use of various levels of abstraction for the design, evolving from the beginning specification to the final physical design. *Hardware Description Languages* (HDLs), such as Verilog and VHDL, have been developed to represent the design in such a way that *Computer-Aided Design* (CAD) tools could be used to manipulate and analyze them in order to produce the desired chip.

The success or failure of a chip depends heavily on the quality of the validation effort invested in the design process. This effort is commonly performed at each level of abstraction, using appropriate CAD tools and methodologies. *Design verification*, which refers to validating a high level abstraction of the design such as a HDL program with respect to its specification, and *manufacturing test generation*, which refers to generating test patterns from a high level abstraction to check for manufacturing defects, constitute significant portions of the validation effort. However, the existing CAD tools to automate these tasks scale poorly with the increasing

complexity and sizes of current designs. Therefore, appropriate methodologies that exploit the modular structure, that is inherently present in large designs, are needed.

1.1 Manufacturing Test Generation

Generating effective manufacturing tests poses a severe challenge as the transistor density increases due to the reduced accessibility of the various parts of the chip. This daunting task is further exacerbated by the ever increasing use of sequential elements for implementing deeply pipelined stages in modern day processors. The test generation complexity grows exponentially with respect to the number of sequential elements [1]. As a result, the *test coverage*, which is a measure of the test quality, is also severely affected. *Automatic Test Pattern Generation* (ATPG) techniques are often ineffective on complete designs unless expensive testability features to partition the design have been incorporated. For example, *scan*-based approaches [2] incorporate additional circuitry to allow the sequential elements to be used as chip-level inputs and outputs. Similar techniques based on *Built-In-Self-Test* (BIST) [3, 4] require extra hardware circuitry that generates test patterns on the chip and determines its correctness through signature analysis of the test responses. Although these *Design For Testability* (DFT) techniques alleviate the test generation problem, they have a significant impact on both the performance and the area of the chip [5].

State-of-the-art VLSI designs are implemented using deep sub-micron technologies, for which functional testing continues to be the most widely accepted method for detecting manufacturing defects. Functional tests applied at-speed [6] can detect commonly prevalent defects such as opens, shorts and delays in deep sub-micron technologies. However, generating functional tests manually is too te-

dious. Sequential ATPG tools may be used to generate these tests from a synthesized description of a design in a HDL, but a complete design is often too large and too complex for the tools to handle. The complexity of sequential ATPG [7] is of the order 4^n if the initial state is known, and increases to 9^n if the initial state is unknown, where n is the number of sequential elements in the design.

The inherent hierarchy in a HDL design may be exploited to reduce the complexity of sequential ATPG. Figure 1.1 illustrates the motivation to develop functional test generation techniques for a module embedded in a relatively large design. Targeting an individual module would result in significantly higher fault coverage, as the tool would have complete access to the inputs and outputs of that module. The test generation time would also be a fraction of what would have been necessary if the patterns were generated with the rest of the design in place. However, due to the constraints imposed on the inputs and outputs by the remainder of the design, many of the patterns generated cannot be translated to the chip level, thereby resulting in a heavy loss of fault coverage. Moreover, the translation of these patterns involves exhaustive reasoning on the behavior of the surrounding logic, which reverts the complexity to that of full chip test generation. Therefore, a suitable technique to derive the test patterns for the *Module Under Test* (MUT), which are valid at the processor level and possessing test coverage and test generation time that is comparable to that of the stand-alone module, is desired.

A variety of strategies have been suggested to generate test patterns on individual modules in a design by extracting the behavior of the surrounding logic. A test generation methodology using test procedures and pattern restrictions was described by Wohl and Waicukauski [8]. Lee and Patel [9] proposed a solution based on two custom ATPG packages, one for full chip level justification and propagation and another for test generation at the module level. Ramachandini and

Thomas [10] suggested a synthesis-based approach in which functional constraints are extracted during the synthesis process and are used to guide a custom ATPG tool. Vishakantaiah *et al.* [11], have developed a tool called ATKET to extract functional constraints from a VHDL *Register-Transfer Level* (RTL) design, which are used along with a custom test generation tool to obtain test patterns. Similar suggested approaches [12, 13] are based on the extraction of useful test generation knowledge from HDL descriptions, but require the use of custom ATPG packages and/or synthesis of complete designs.

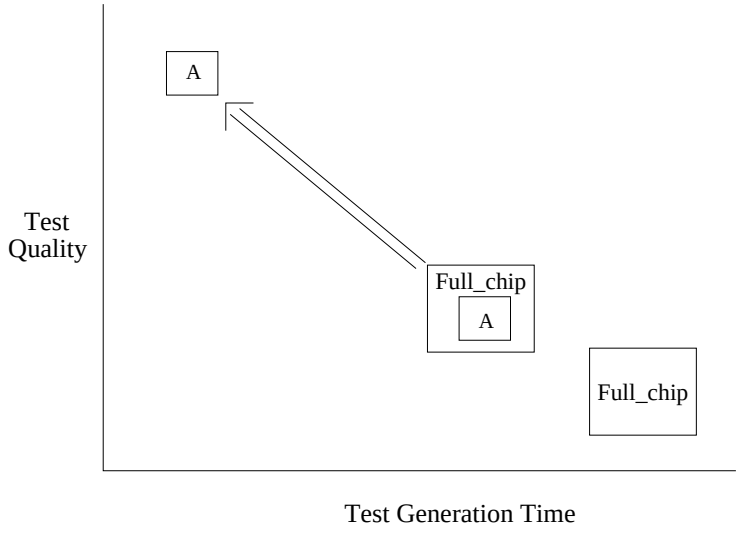

Figure 1.1: Motivation for Hierarchical Test Generation

Tupuri *et al.* suggested a hierarchical abstraction technique [14], wherein each module in a design hierarchy is targeted at a time. The logic surrounding this module is extracted and synthesized to the gate level. These synthesized constraints are provided to a commercial ATPG tool to generate test patterns for the MUT. This approach has shown promising results for generating high quality tests

in a reasonable amount of time. However, in larger designs, the submodules may themselves prove to be too complex for the ATPG tool, and this technique may not be directly applicable. Therefore, it is imperative to target modules at the lower levels of hierarchy, but then the surrounding logic may prove to be too complex for its abstraction.

1.2 Design Verification

It has been estimated that over 60% of the design effort is now in design verification [27], and this percentage is expected to increase in the future. The consequences of ignoring this phase of the design process may result in faulty designs causing huge market losses to the industry. However, existing verification tools and methodologies have also been unable to scale with the increasing design complexities. Functional simulation [28] of the design using validation tests is the main technique used in industry to verify large designs, owing to to its simplicity and scalability. Unfortunately, as designs become larger, the likelihood that the tests will uncover subtle bugs becomes smaller. Exhaustive simulation using all possible input sequences is impractical.

Although simulation is expected to remain a key technique in validating complex designs, there has been considerable interest in the use of formal methods [29] as a complementary approach in examining cases that have not been covered by simulation. These techniques incorporate an unambiguous specification of the design and use mathematical reasoning to systematically explore all possible ways to establish design correctness. Since the entire design space has to be searched (at least implicitly) in order to establish definitive correctness, the existing formal approaches are only applicable to small portions of a design. Moreover, the

automation of these methods requires the finite state machine of the system to be represented by either *Binary Decision Diagrams* (BDDs) or one of the other decision diagram variations. BDD sizes are extremely sensitive to variable ordering and for some circuits (such as adders and multipliers) always achieve exponential complexity [30].

At present, formal techniques have found a wide application in establishing the equivalence between the RTL and the logic level [31]. The state-of-the-art commercial tools require a correspondence between the state elements in the RTL description and the logic level description, essentially dealing with a combinational equivalence problem. This is still a very difficult problem (establishing the equivalence of two Boolean functions is NP-complete), but the use of good heuristics and the use of a combination of different techniques, including BDDs, ATPG, and *SATisfiability* (SAT) solvers, allow industry tools to formally verify the Boolean equivalence of combinational blocks in commercial designs.

To check the correctness of the RTL itself, several approaches such as theorem proving [32], model checking [33, 34] and symbolic trajectory evaluation [35] have been proposed. Each approach makes a tradeoff between expressiveness and capacity. Theorem proving techniques attempt to show that there exists a formal proof of the correctness formulas characterizing the design. Although a generic approach, generating the proof automatically is very difficult and therefore, theorem provers are mainly used to check proofs that have been generated manually. As a result, this approach lacks the level of automation required for it to be useful in practice.

Model checking uses a *State Transition Graph* (STG) to represent a digital system and to specify its "properties" using temporal logic formulas. These properties are validated by traversing the STG using powerful algorithmic search strate-

gies. The STG can be easily extracted from the HDL and traversed automatically, and the property specification can be performed with relative ease. While these factors make model checking a viable approach, the existing tools and methodologies for property checking use BDDs in some form to represent the STG, quickly reaching existing computational limits.

1.2.1 Bounded Model Checking

In practice, an automated formal method is often applied to reveal the presence of bugs rather than to provide proof of correctness. Based on this observation, Biere *et al.* [36] introduced a symbolic model checking technique called *Bounded Model Checking* (BMC), which searches for counterexamples of a maximal length (the bound) for the temporal properties of a system described using *Linear Temporal Logic* (LTL) [29]. This approach involves generating and solving for the satisfiability of propositional decision procedures, instead of BDDs. If it can be proven that the diameter of the state transition diagram of the design is no larger than the maximum bound of the time steps, BMC becomes complete thereby proving the unreachability of the target. The benefits of this approach has been demonstrated by Copty *et al.* [38] on real designs taken from Intel's Pentium 4 processor containing over 1000 state variables. Structural algorithms for computing a bound on the diameter of the STG, have been proposed [39] to guarantee the completeness of the bounded property check.

SAT solvers operate on boolean expressions, but do not suffer from the potential space explosion problem of BDD-based methods, as they do not use canonical representations. SAT-based techniques have been successfully applied in various domains [37]. However, SAT-based BMC usually requires explicit unrolling of the

circuits and perform a depth first search to find a satisfying variable assignment. As a result, the SAT clauses often become too large and too tedious to solve. Recent work [40, 41, 42, 43] for checking "safety" properties have shown that ATPG-based techniques may be able to overcome these limitations. In these approaches ATPG has been shown to perform efficient state-space search without building cumbersome representations of the STG.

1.2.2 Symbolic Trajectory Evaluation

Verification using conventional simulation requires tedious analysis of several possible input states and sequences along with analysis of their responses. In order to overcome this limitation, symbolic simulation which uses Boolean variables for input values has been introduced. The resulting responses are Boolean functions which capture the behavior of the circuit for an entire class of input values. These Boolean functions are represented using BDDs for efficient comparison of the simulation results for correctness, made possible by the canonicity of the representation [44].

Based on these symbolic simulation techniques, Seger and Bryant [35] proposed an efficient lattice-based model checking approach, called *Symbolic Trajectory Evaluation* (STE). This approach used an improved symbolic simulator to verify the correctness of circuit behavior described as trajectory assertions. These assertions are derived from a restricted subset of temporal logic over a bounded length of state sequences. The efficient use of a quaternary circuit model for enhanced capability for symbolic manipulation has rendered this technique very attractive for verifying industrial-strength designs in companies such as Intel, Motorola, and IBM.

A serious drawback of this approach has been the restricted expressiveness of its temporal assertions. A property that extends over an infinitely long state sequence cannot be expressed in STE. Recently Yang and Seger [45] proposed a generalized STE methodology to express and verify a richer set of properties, which shows great promise in verifying a larger and a wider class of designs. However, the underlying BDD-based representations quickly grow in size and require tedious application of incremental methodologies.

1.3 Organization of the Dissertation

The next chapter describes the development of an elegant technique for easing the complexity of HDL programs for easier analysis using CAD tools. The theoretical basis for using "program slicing" for hardware programs is established by adapting some of the key definitions from software engineering. The formal semantics for using HDL slicing with sequential ATPG tools are established to prove the soundness and completeness of this technique. This is followed by a discussion on a related technique called *cone-of-influence reduction* (COI) and its relevance to the slicing methodology.

Chapter 3 explains the test generation approach using HDL slicing. Next is a presentation of this methodology for a tool that has been developed to automate the approach, followed by the experimental results on Verilog benchmarks using this tool.

Chapter 4 introduces the basic ideas of model checking for verifying the correctness of designs. A SAT-based bounded model checking approach to check properties over bounded intervals of time is described to motivate the development of a better ATPG-based approach. This is followed by the details of symbolic trajec-

tory evaluation, another powerful technique to check bounded property assertions.

Chapter 5 presents a novel methodology for applying sequential ATPG for property checking. The monitor state machines that have been developed for use with sequential ATPG engines are described, along with some supporting results generated on benchmark circuits.

Chapter 6 describes the application of HDL slicing to the ATPG-based property checking methodology and to a STE-based verification in the Forte framework used at Intel.

The dissertation concludes in Chapter 7 with some final remarks and possible future CAD applications, which may benefit from abstraction methods similar to what has been described herein.

Chapter 2

HDL Slicing

Program slicing, as was originally proposed by Weiser [15], is a static program analysis technique to extract appropriate portions of programs that are relevant to an application. These portions are referred to as *slices* – artifacts that maintain exact information about the program's behavior projected onto the relevant segments of the original program. This technique potentially removes large quantities of extraneous code in the original program while maintaining the functional equivalence with respect to the aspect under analysis. This technique has been widely studied and applied to a myriad of applications in software engineering such as debugging [16], testing [17], maintenance [18] and reuse [19]. Software debugging, for example, involves tracing back from a statement that generates an incorrect result. This procedure may be perceived as computing and examining the "slice" of the program with respect to the variables associated with the statement, in order to identify the statement or statements causing the erroneous evaluation. Therefore, an automated approach to derive the desired slice would be an asset in a software development environment.

Such a source-to-source transformation technique offers an opportunity to

formulate a methodical approach to simplify a design described in a HDL for test generation without having to synthesize a prohibitively large design. However, most of these algorithms have been developed for sequential languages and cannot be directly applied to HDLs such as Verilog or VHDL, which allow concurrent constructs. Static slicing of concurrent software programs [20, 21] has been adapted for HDLs. Iwaihara *et al.* [22] suggested an approach to use program slicing for analyzing VHDL designs and presented the reduction obtained on a set of benchmarks using each of the outputs as the slicing criterion. An automated program slicing approach for VHDL was proposed by Clarke *et al.* [23], in which VHDL constructs were mapped onto constructs for C-like procedural languages. The primary focus of their work was to reduce the reachable state-space for formal property verification. The previous two contributions suggested potential applications to simulation, functional testing, regression testing, testability analysis, design modification and maintenance without any supporting experimental results.

Related approaches using data flow analysis were used to propagate test patterns for a MUT [24], to perform accessibility analysis to improve DFT [25], and to improve testability based on propagation of the value ranges of variables [26]. However, these techniques cannot be directly applied to many of the commercial tools for test generation and verification, that operate on gate-level descriptions of designs. Therefore, a systematic approach with a strong theoretical basis for using program slicing on HDL programs is necessary to apply this technique for practical use.

2.1 Basic Definitions

This section consists of some key definitions of the elements that constitute a program dependence graph, and are based on earlier work in program slicing [15, 19]. These definitions provide the foundation for slicing HDL programs using control flow and def-use graphs [47].

Definition 1 *A* digraph G_D *is a structure* $\langle N, E \rangle$*, where* N *is a set of nodes and E* $\subseteq N \times N$ *is a set of edges.*

Definition 2 *Given an edge* $(n_i, n_j) \in E$*,* n_i *is said to be a* predecessor *of* n_j*, and* n_j *is said to be the* successor *of* n_i*. PRED(n) and SUCC(n) are the set of predecessors and successors of a node n, respectively.*

Definition 3 *The* indegree *of a node n, denoted by IN(n), is the number of predecessors of n. The outdegree of a node n, denoted by OUT(n), is the number of successors of n.*

Definition 4 *A* flowgraph G_F *is a structure* $\langle N, E, n_0 \rangle$*, such that* $\langle N, E \rangle$ *is a* digraph *and* $n_0 \in N$ *such that IN(*n_0*) = 0.*

Definition 5 *A* hammock graph G_H *is a structure* $\langle N, E, n_0, n_e \rangle$ *such that both* $\langle N, E, n_0 \rangle$ *and* $\langle N, E^{-1}, n_e \rangle$ *are* flowgraphs *where* $E^{-1} = \{(a, b) \mid (b, a) \in E\}$.

Definition 6 *A* control flow graph *(CFG)* G_C *is a hammock graph which is interpreted as a program procedure.*

The nodes of a CFG represent simple statements such as assignments, branch and loop conditions, *etc*. The edges represent control flow transfer between statements. In the following discussion, N_G represents the set of nodes in a graph G.

Definition 7 *A* def-use graph G_{DU} *is a structure* $\langle G_C, \Sigma, D, U \rangle$ *where* Σ *is a set of variables in a procedure, D :* $N_{G_C} \mapsto \Sigma$ *and U :* $N_{G_C} \mapsto \Sigma$ *are the functions mapping the nodes of* G_C, i.e., N_{G_C} *to the set of variables defined or used in the statements corresponding to the nodes, respectively.*

Definition 8 *Given a node n* $\in N_{G_C}$*, the definition set D(n) is the set of variables (on the left-part of an assignment operator in a statement) that are defined at n. The usage set U(n) is the set of variables (on the right-part of an assignment operator in a statement) that are used at n.*

Definition 9 *A* slicing criterion *C is a pair* $\langle i, V \rangle$ *such that i* $\in N_{G_C}$ *and V* $\in \Sigma$*. In a procedure P, i is a statement and V is a subset of the variables.*

Given a slicing criterion C = $\langle i, V \rangle$, a set of statements I_s is said to affect (either directly or transitively) the values of V at i, when I_s computes a subset of V that is used in i. Similarly, I_s is said to be affected by (either directly or transitively) the values of V at i, when a subset of V that is defined at i computes the variables used in I_s.

Definition 10 *A* slice *S of a procedure P on a slicing criterion* $\langle i, V \rangle$ *is an executable subset of P containing all the statements that may affect (or may be affected by) the values of V at i.*

Slices may be classified [48] in three ways, – *static* or *dynamic*, *forward* or *backward*, and *closure* or *executable*. A *static* slice contains all statements that may be relevant to a computation, and a *dynamic* slice extracts all statements relevant to a computation for a specific set of inputs/outputs. A *forward* slice is a set of statements whose values may be affected by the values of V at i, and a *backward*

slice is a set of statements that may affect the values V at i. A *closure* slice relates to the variable of interest through a closure of dependencies and is not necessarily a syntactically valid program. An *executable* slice, on the other hand, is a reduced program that preserves the behavior of the original program with respect to the slicing criterion.

Definition 11 *The set of variables immediately relevant to a slicing criterion C, denoted by R^0_C(n), is defined as*

$$R^0_C(n) = \{v \in V \mid n = i\} \cup \{U(n) \mid D(n) \cap R^0_C(\ SUCC(n)\ ) \neq \emptyset\} \cup \{R^0_C(SUCC(n)) - D(n)\}$$

The superscript 0 indicates that this set of variables is directly relevant. The first subset is the base case; the second marks the variables used to assign values to other relevant variables, as relevant. The third subset removes a relevant variable for which all the immediately relevant variables have been found.

Definition 12 *The set of statements included in the slice by R^0_C(n), denoted by S^0_C, is defined as*

$$S^0_C = \{n \in N_{G_C} \mid D(n) \cap R^0_C(\ SUCC(n)\ ) \neq \emptyset\}$$

The set S^0_C does not include the control flow such as branch statements which allow execution of the statements in S^0_C. Therefore, for each statement of S^0_C, a *me-too* set denoted by MT is introduced [17]. The me-too set of a statement $i \in S^0_C$, is the set of all control statements which regulate the execution of *i*, and is defined as follows.

Definition 13 *The set of control statements which dominate the execution of statements in S_C^0, denoted by B_C^0, is defined as*

$$B_C^0 = \bigcup_{n \in S_C^0} MT(n)$$

2.2 HDL Slicing

A typical design described in a HDL is a non-halting program with several communicating processes that can execute concurrently. The processes communicate with each other through signals that are shared between the processes. The definitions for slicing procedures in software programs may be directly extended to such a HDL process. But unlike procedures in a software program, a process is not called explicitly, but is activated by appropriate changes in signal values in the sensitivity list of the process. These changes may be triggered by other processes executing concurrently. Therefore, unlike in software programs where explicit procedure call statements activate a procedure definition, any dependence of a process on a signal definition in other process(es) can potentially activate the process execution. To incorporate this inter-process communication, a notion of *signal dependency* is introduced [22, 23]. In the following definitions, a *process region* denotes an executable subset of statements in a process.

Definition 14 *A process region p is said to be* signal dependent *on a statement i, if i assigns a value to a signal to which p is sensitive.*

The next set of definitions deals with inter-process communication. These definitions are explained with respect to a commonly used synthesizable subset of

Verilog semantics, and precludes constructs including functions, tasks, forks, delay control statements and non-blocking assignments.

Definition 15 *An* inter-process control flow graph *for a module G_C^M is a structure $\langle G_{C1}, G_{C2}, \ldots, G_{Ck}, E_{SD}\rangle$ where $G_{C1}, G_{C2}, \ldots, G_{Ck}$ are control flow graphs representing the processes in the module, and E_{SD} is the set of edges representing the signal dependencies between the processes.*

Definition 16 *An* inter-process def-use graph *for a module G_{DU}^M is a structure $\langle G_C^M, \Sigma, D, U\rangle$ where Σ is the set of signals in the module, D : $N_{G_C^M} \mapsto \Delta(\Sigma)$ and U : $N_{G_C^M} \mapsto \Upsilon(\Sigma)$ are the functions mapping the nodes of G_C^M in the set of signals which are defined or used in the statements corresponding to the nodes.*

These definitions hold for other concurrent statements such as continuous assignments, that are considered as single-line unique processes.

Definition 17 *An* inter-process slice S^{ip} *within a module M, on a given criterion $\langle i, V\rangle$ is an executable subset of M obtained recursively containing (a) all the statements that may affect (or may be affected by) the values of V at i within the process P in which i is defined, and (b) all the slices on the slicing criterion $\langle i_s, V_s\rangle$, where i_s is the set of statements defining or using the set of signals V_s on which process P is dependent.*

These definitions form the basis for applying slicing techniques for analyzing Verilog designs, which is the HDL chosen for our implementation. The proposed methodology derives a static slice, which is executable and consists of both forward and backward slices.

2.3 Taming Design Complexity

A digital system described in Verilog is often a hierarchical composition of modules, each of which is composed of several processes. Consider a Verilog program [23], $\Pi = \|_{i=1}^{n} P_i$, where i and n are integers, P_i is a process in the program, and $\|$ is the composition operator [51]. This division of processes into modules may be exploited by existing CAD tools by employing a *divide-and-conquer* approach. The basic approach [52, 53] is illustrated in Figure 2.1. The necessary definitions to explain and prove the methodology, are derived in this section.

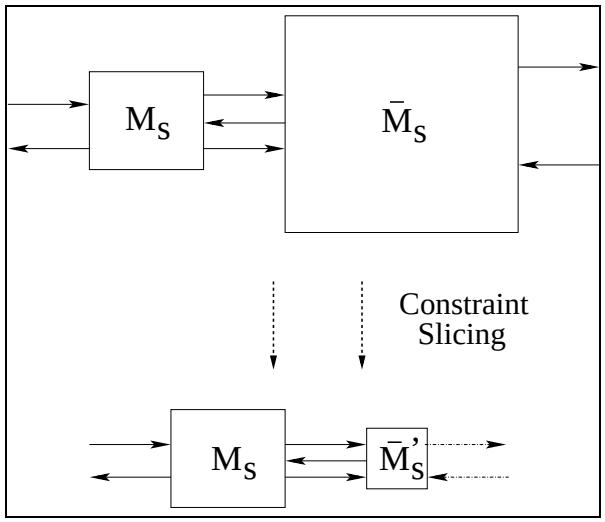

Figure 2.1: Basic Approach for Complexity Reduction

Let M_s be a sub-module in a design represented by $\Pi_1 = \|_{i=1}^{k} P_i$, where k is an integer such that $k < n$. Let $\bar{M}_s$ be the surrounding logic, which encompasses the rest of the design, and be represented by $\Pi_2 = \|_{i=k+1}^{n} P_i$.

Definition 18 *A* constraint slicing criterion C_{M_s} *is a pair* $\langle i, V \rangle$ *such that* $i \in N_{\Pi_1}$ *and* $V \in \Sigma_{\Pi_1}^{IO}$*, where*

- $\Sigma^{IO}_{\Pi_1}$ *is the set of input/output signals of* M_s *and*
- N_{Π_1} *is the set of input/output nodes representing the declarative statements of* $\Sigma^{IO}_{\Pi_1}$ *in* M_s.

Definition 19 *A* constraint slice S_{M_s} *is a slice obtained on the slice criterion* C_{M_s}*, which is an executable subset of* Π_2*, containing all statements in* $\bar{M}_s$ *which constrain the values of V declared at i. The set of processes in* S_{M_s} *is represented by* $\widehat{\Pi}_2 = \|_{i=k+1}^{n} \widehat{P}_i$

The set of input constraints is obtained by backward slicing on the input signals of M_s, while the output constraints are obtained by forward slicing on the output signals. The combined set of constraint slices represent the behavior of $\bar{M}_s$ visible to M_s. This reduced surrounding logic $\bar{M}_s{}'$, may be synthesized to gate level and used to analyze the module M_s for test generation and verification.

For all m $\geq$ 0 where m is an integer, the constraint slice is built recursively using the following equations [19].

$$R^{m+1}_{C_{M_s}}(n) = R^{m}_{C_{M_s}}(n) \ \cup (\bigcup_{b \in B^{m}_{C_{M_s}}} R^{0}_{\langle b, U(b) \rangle}(n) \ ) \tag{2.1}$$

$$S^{m+1}_{C_{M_s}} = \ \{n \in N_{G^{\bar{M}_s}_C} \mid D(n) \ \cap R^{m+1}_{C_{M_s}}(SUCC(n)) \neq \emptyset\} \cup \ B^{m}_{C_{M_s}} \tag{2.2}$$

$$B^{m+1}_{C_{M_s}} = \{b \in N_{G^{\bar{M}_s}_C} \mid MT(b) \ \cap S^{m+1}_{C_{M_s}} \neq \emptyset\} \tag{2.3}$$

These iterative steps terminate when the primary inputs/outputs are reached and no new statements/signals are included.

Definition 20 *The* termination condition *TC is defined as*

$S_{C_{M_s}} = S^{t+1}_{C_{M_s}}$ *where t is an iteration step such that*

$$\forall n \in N_{G_C^{\bar{M}_s}} : R^{t+1}_{C_{M_s}}(n) = R^{t}_{C_{M_s}}(n) = R_{C_{M_s}}(n).$$

Lemma 1 (Distributivity) *A constraint slice on the slicing criterion C_{M_s} is distributive over the set of Verilog signals $v_1, v_2, \ldots, v_n \in V$ at the statement i.*

$$S_{\langle i,\{v_1,v_2,\ \cdots\ v_n\}\rangle} = S_{\langle i,v_1\rangle} \cup S_{\langle i,v_2\rangle} \cup \cdots \cup S_{\langle i,v_n\rangle}$$

$$R_{\langle i,\{v_1 v_2,\ \cdots\ v_n\}\rangle}(n) = R_{\langle i,v_1\rangle}(n) \cup R_{\langle i,v_2\rangle}(n) \cup \cdots \cup R_{\langle i,v_n\rangle}(n)$$

Proof: Consider the iterative step for a signal $v_h \in V$, where h is an integer such that $h < n$, to obtain $R_{\langle i,v_h\rangle}$ and $S_{\langle i,v_h\rangle}$. This step computes the slice which may overlap with the slice on another signal $\bar{v_h} \in V$. From Equations 2.1-2.3, the union of these slices only contains some statements and signals that are computed more than once, and results in the same slice obtained using a union of these signals at the statement i. □

Lemma 1 shows that the constraint slice for the inputs/outputs of a module can be built by composing the slices obtained separately on each input and output. The next lemma shows that the generated slice correctly encompasses the constraints imposed on the inputs and outputs of the MUT in the original design.

Lemma 2 (Correctness) *Let Q be the constraint slice obtained from Π_2 with respect to a slicing criterion C_{M_s}. Let a process trace T = $\langle T_0, T_1, T_2, \ldots, T_c, \ldots\rangle$ be the sequence of the relevant signals in the surrounding logic $\bar{M}_s$, where c is the simulation cycle and $T_c \in \{0, 1, X, Z\}$. Then, the process traces of both Q and Π_2 are identical with respect to the signals defined by $R_{C_{M_s}}$.*

Proof: Let j $\geq$ 0 be an integer. Definitions 11, 12, 13 and Equations 2.1-2.3 are used for the proof by mathematical induction.

Basis: j = 0. The base set $R^1_{C_{M_s}}$(n) includes all immediately relevant signals that have potential effects on the def-use chain ending in $V \in \Sigma^{IO}_{\Pi_1}$. Therefore, the sub-slice of Q, represented by $S^1_{C_{M_s}}$ and $B^1_{C_{M_s}}$, is a subset of Π_2 that includes the values of V.

Hypothesis: For j = f, assume that Q_f, the sub-slice of Q defined by Equations 2.1-2.3, computes an identical subset of the trace produced by Π_2.

Induction step: Let j = f+1. The first part of the iterative step to compute $R^{f+2}_{C_{M_s}}(n)$ is the sub-slice given by the hypothesis, *i.e.*, $R^{f+1}_{C_{M_s}}(n)$. The second part derives those statements that have a relevance to the def-use chain ending in $R^{f+1}_{C_{M_s}}(n)$. Q^{f+2}, the sub-slice of Q at the $(f+1)^{th}$ step in the iteration, represented by $S^{f+2}_{C_{M_s}}$ and $B^{f+2}_{C_{M_s}}$, captures the dependencies for signals in $R^{f+1}_{C_{M_s}}(n)$. Therefore, Q^{f+2} ensures the computation of an identical subset of the trace produced by Π_2. □

Lemma 2 establishes the accuracy of this methodology in providing an accurate constraint slice for the inputs and outputs of a MUT. The next theorem proves that the reduced model retains the *transitive fan-in* and *transitive fan-out* cones from the chip-level inputs and outputs for every faulty signal in the MUT. These cones [1] have been traditionally defined with respect to signals in the gate-level description needed for justification and propagation of the signal assignments for ATPG. At the RT level, these gates correspond to the statements and signals that lie in the slice of a particular signal in a statement. The transitive fan-in cone (TFI) of a signal is the set of all statements and signals in a design that transitively drive the signal. Similarly, the transitive fan-out cone (TFO) of a signal is the set of all statements and signals in a design that are transitively driven by the signal.

Theorem 1 (Consistency) *An ATPG algorithm would generate a valid set of test patterns for faults in a MUT by using its constraint slice.*

Proof: Referring to Figure 2.2, consider a faulty signal, f in the MUT, M_s. An ATPG algorithm would require that all the signals and statements for justification and propagation of f are preserved.

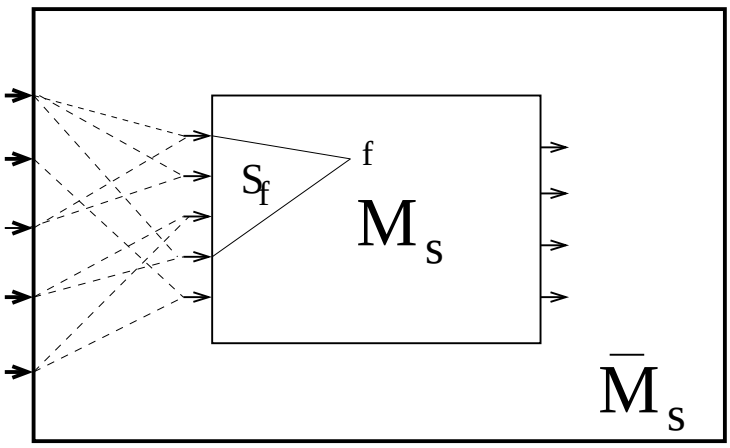

Figure 2.2: Construction of Transitive Fan-in Cone

Justification: The TFI of f is a subset of the union two parts, namely (a) all the signals and statements that transitively affect f, represented by S_f^I, where $I \in \Sigma_{\Pi_1}^I$, originating from $\Sigma_{\Pi_1}^I$, and (b) all the signals and statements that transitively affect these inputs of M_s, originating from the inputs of the program Π.

The first part is implicitly included in the set of processes Π_1, which is not a part of the slicing criterion C_{M_s}. The TFI for each input $i \in N_{\Pi_1}$, represented by TFI_i, is the set of signals and statements in Π_2 that transitively affect i. Definition 19 shows that $TFI_i, \forall\ i \in N_{\Pi_1}$, is contained in the executable subset S_{M_s}. Therefore, the set of processes defined by $\widehat{\Pi} = \Pi_1 || \widehat{\Pi}_2$ includes the TFI of f, which

is sufficient for signal justification.

Propagation: The proof for TFO of f follows similar reasoning as shown for justification. However, in addition to the TFO of f, represented by TFO_f, propagation requires that the set of all signals $\Upsilon = \bigcup_{n \in \widehat{\Pi}} U(n)$ (not including the signals in the TFO), of the statements in TFO_f, need to be justified to the primary inputs. The set of TFIs for justifying the signals in Υ is a subset of the union of two parts, namely (1) Υ_1, the set of TFIs for signals in Π_1, and (2) Υ_2, the set of TFIs for signals in $\widehat{\Pi}_2$.

The first part is contained in S_{M_s}, as shown in the proof for justification. Equation 2.1 recursively includes signals $U(n)$, in the TFI as shown in Definition 11. Equations 2.2 and 2.3 recursively include the relevant statements in the TFI, as they compute statements including the signals computed in Equation 2.1. Hence, all the necessary propagation paths for each faulty signal in M_s are included in the constraint slice. □

2.4 Hierarchical Slicing

The application of this methodology may be too tedious on large designs with multiple-levels of hierarchy. In such cases, the MUT that can be effectively handled by an ATPG tool may be embedded several levels down the hierarchy, and the surrounding logic may be too big to extract the constraint slice directly. A solution to this problem is to perform slicing hierarchically from one module level to the next in the hierarchy. These *hierarchical slices* are composed to obtain the complete constraint slice.

Referring to the Figure 2.3, let the processes in the environment Π_2 be composed of subsets $\Phi_1, \Phi_2, \ldots, \Phi_n$ representing the surrounding logic $\bar{M}_s^1,\ \bar{M}_s^2,\ \ldots,\ \bar{M}_s^n$

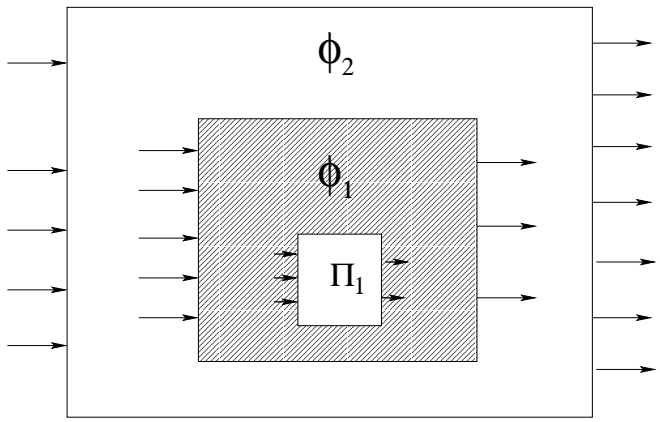

Figure 2.3: Snapshot in a Design Hierarchy

at each level of hierarchy from M_s. A module M_s^k, at a given level of hierarchy k, is obtained by composing the MUT with the subset of processes in the surrounding logic that are contained in k. This is represented by $\Pi_1 \parallel (\|_{i=1}^{k} \Phi_k)$, where $1 < k \leq n$. The slice of a subset Φ_k, on a module M_s^k, is represented by $\widehat{\Phi}_k$.

Theorem 2 (Composition) *The composition of the constraint slices at each level of hierarchy yields the desired constraint slice S_{M_s}.*

Proof: Let k $\geq$ 0 be an integer.

Basis: k = 1 is a trivial case, as the constraint is directly obtained defined $\Pi_1 \parallel \widehat{\Phi}_1)$.

Hypothesis: k = h, assume that the hierarchical slices composed up to the hierarchy h, are contained in the desired constraint slice given by
$\Pi_1 \parallel (\|_{i=1}^{h} \widehat{\Phi}_k)$

Induction: k = h+1. The slicing criterion $C_{M_s^h}$, is the pair $\langle i, V \rangle$ such that i $\in$ N_{Π_h} and V $\in$ $\Sigma_{\Pi_h}^{IO}$. The constraint slice $S_{M_s^h}$ is obtained recursively using

Equations 2.1-2.3, and is defined by $\widehat{\Phi}_{h+1}$. Composing this with the slice obtained for k = h, $\Pi_1 \parallel (\|_{i=1}^{h} \widehat{\Phi}_k) \parallel \widehat{\Phi}_{h+1}$, which reduces to $\Pi_1 \parallel (\|_{i=1}^{h+1} \widehat{\Phi}_k)$, which is the desired constraint slice composed up to the hierarchy h+1. □

An additional benefit is that the slices at higher levels of hierarchy may be reused for extracting constraint slices on other modules formed by a subset of processes, say Π_3, within the same level of hierarchy, say 1. It is sufficient to derive the constraint slice within module M_s^1 and compose it with the slices of processes Φ_k, where $2 < k \leq n$. Therefore, the constraint extraction time is reduced and such a methodology scales well for large designs.

2.5 Cone of Influence Reduction

A closely related technique to HDL slicing approach is *cone of influence* (COI) reduction, though they differ in many respects. The primary idea of COI [62] is to construct a dependence graph of the program and traverse it starting from the variables in the specification. The dependence graph is usually represented as a vector of transition functions generated from a symbolic encoding of a system such as BDDs. The set of state variables reached (and the necessary transition relations between them) during this traversal form the COI of the variables in the specification. This may be viewed as *post-encoding* slicing.

However, as the designs grow in size, the generation of the transition relation from a description language can itself be a bottleneck. Therefore, this approach can only be used for localized reduction of the complexity for a specific site in a gate-level description. Thus *pre-encoding* slicing techniques are needed to reduce the complexity of large designs for CAD tasks.

HDL slicing operates on the HDL source code directly and uses the infor-

mation about the language semantics (such as *if-then-else, case, for, while* loops) to generate the relevant subset of the program. The COI reduction can be viewed as a special case of slicing where an assignment statement is the only language construct. HDL slicing is applied at the RT level and can, therefore, handle complete designs.

The proposed scheme performs slicing on the MUT interface rather than on individual fault sites in the gate level descriptions. An added advantage is that the synthesis tool would need to generate the gate level netlist only for the relevant sliced design and not for the whole chip. A disadvantage of HDL slicing is that it is specific to the language being used. Clarke *et al.* [23] have discussed other differences in more detail with respect to model checking. The proposed slicing methodology complements any of the already existing post-encoding reduction techniques, such as COI, in the state-of-the-art commercial CAD tools for design verification and test generation.

Chapter 3

Test Generation using Slicing

Functional test generation using sequential ATPG tools is extremely computation intensive on full chip designs. Therefore, it is necessary to adopt a *divide-and-conquer* strategy for generating high quality test patterns in a reasonable amount of time. Such a strategy may be realized by exploiting the hierarchical structure that is inherently prevalent in current designs. However, a naive approach of targeting faults in individual modules would necessitate a cumbersome translation of these patterns to the chip level. In addition, some of these tests for faults in the MUT may be impossible to apply from the chip level; searching for alternate tests for these faults would be comparable to generating tests on the complete design. Therefore, a scalable and a systematic methodology to hierarchically generate effective test patterns is desired.

This chapter describes the proposed test generation methodology, which analyzes the RTL model of a full chip design to derive the ATPG view of the MUT in a synthesizable form. For illustration, a simple RTL example is shown along with the corresponding gate level structure in Figure 3.1. The module M21 is chosen as the MUT. The module definitions for M1, M3 and M22 and the corresponding gate

level descriptions are not provided since they are not necessary for illustrating the approach.

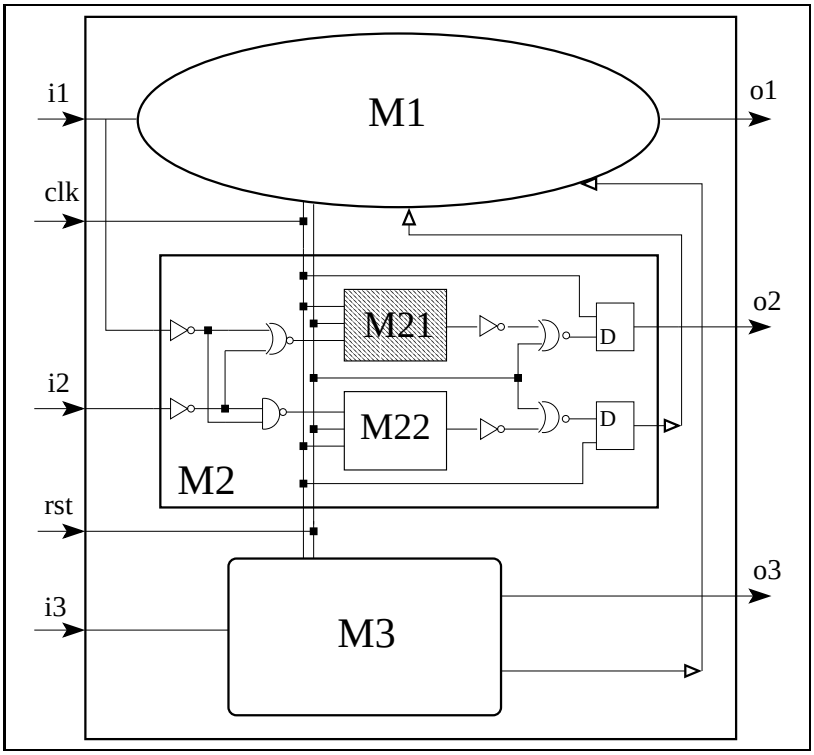

Figure 3.1: Original Design

```
/* RTL example: Original Design */
module Top (clk, reset, i1, i2, i3, o1, o2, o3);
  input clk, reset;
  input i1, i2, i3;
  output o1, o2, o3;
  wire w12, w13;
  M1 M1_I (clk, reset, i1, w12, w13, o1);
```

```
  M2 M2_I (clk, reset, i1, i2, w12, o2);
  M3 M3_I (clk, reset, i3, w13, o3);
endmodule
module M2 (clk, reset, i1, i2, w12, o2);
  input clk, reset;
  input i1, i2;
  output o2, w12;
  reg o2, w12;
  wire i21, i22, o21, o22;
  assign i21 = i1 & i2;
  assign i22 = i1 | i2;
  always @(posedge clk) begin
    if(reset) begin
      o2 = 0;
      w12 = 0;
    end
    else begin
      o2 = o21;
      w12 = o22;
    end
  end
  M21 M21_I (clk, reset, i21, o21);
  M22 M22_I (clk, reset, i22, o22);
endmodule
```

Test generation targeting the faults in the MUT requires only the relevant surrounding logic. The rest of the logic is removed methodically from the RTL design using a program slicing methodology that will be described in later sections of this chapter. The "sliced" RTL is synthesized to provide the gate level ATPG view as shown in Figure 3.2 along with the corresponding RTL description.

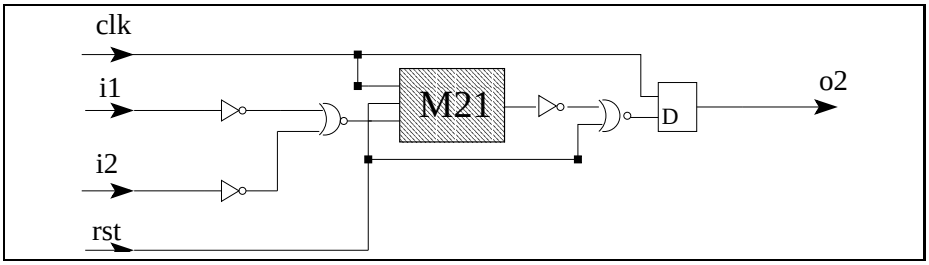

Figure 3.2: Sliced Design

```
/* RTL example: Constraint Slice on M22_I */
module Top (clk, reset, i1, i2, o2);
  input clk, reset;
  input i1, i2;
  output o2;
  M2 M2_I (clk, reset, i1, i2, o2);
endmodule
module M2 (clk, reset, i1, i2, o2);
  input clk, reset;
  input i1, i2;
  output o2;
  reg o2;
  wire i21, o21;
```

```
  assign i21 = i1 & i2;
  always @(posedge clk) begin
    if(reset) o2 = 0;
    else o2 = o21;
  end
  M21 M21_I (clk, reset, i21, o21);
endmodule
```

A commercial sequential ATPG tool is used to generate the desired test patterns targeting the faults in M21. Test generation for all faults in module M21 in the sliced design would be much faster than when targeting the same faults in the original design, since the sliced design is much simpler. The tests generated for the slice can be directly mapped to the original design, since the entire environment of M21 is captured in the slice. The rest of this chapter describes the process of generating such slices and provides theoretical proofs and experimental evidence for the soundness and completeness of the approach [49].

3.1 PIERs

A constraint slice may be reduced by using the set of directly accessible internal registers, called *PIERs* (Primary Input/output accEssible Registers) [14]. These registers which can be read from (or written into) using the load/store instructions are identified *a priori* and are treated as pseudo primary inputs/outputs of the design during constraint slicing. Therefore, the iterative Equations 2.1 - 2.3 (derived in Chapter 2) would terminate earlier at a PIER, where applicable. This results in

faster slicing and results in a smaller slice, thereby reducing the sequential depth during the test generation process. The instructions to load/unload these PIERs are used to translate these patterns to the chip level.

Generating tests for the PIERs can performed using either of the following two conventional techniques. If the PIERs are implemented as a register file, traditional memory test algorithms such as March C [54] are sufficient. For PIERs which are implemented in random locations, the functional testing method proposed by Brahme and Abraham [55] is very effective. This approach assigns a unique code to each of the registers, while ensuring that there is no conflict with the generated patterns.

3.2 FACTOR

The methodology described in Section 2.3 is used for the implementation of FACTOR (**F**unction**A**l **C**ons**T**raint extract**OR**). The tool automatically derives the constraint slices on modules in a Verilog program. It has been implemented in PERL using the *Rough Verilog Parser* [56]. The parser supports both Register-Transfer (RT) and gate level Verilog constructs. The parse tree supporting the internal data structure has been suitably modified to incorporate the *def-use* chains and *use-def* chains for each signal in order to calculate the slices more efficiently. These chains contain the statements where a signal definition is used and where a signal usage is defined, respectively. For each statement during program traversal,

1. If the statement is a controlling statement (*always, if, else, case etc.*), it is added to an already existing parse tree. Otherwise a new node is initialized and the statement is stored.

2. For all others, the current statement is added to the use-def chain of the

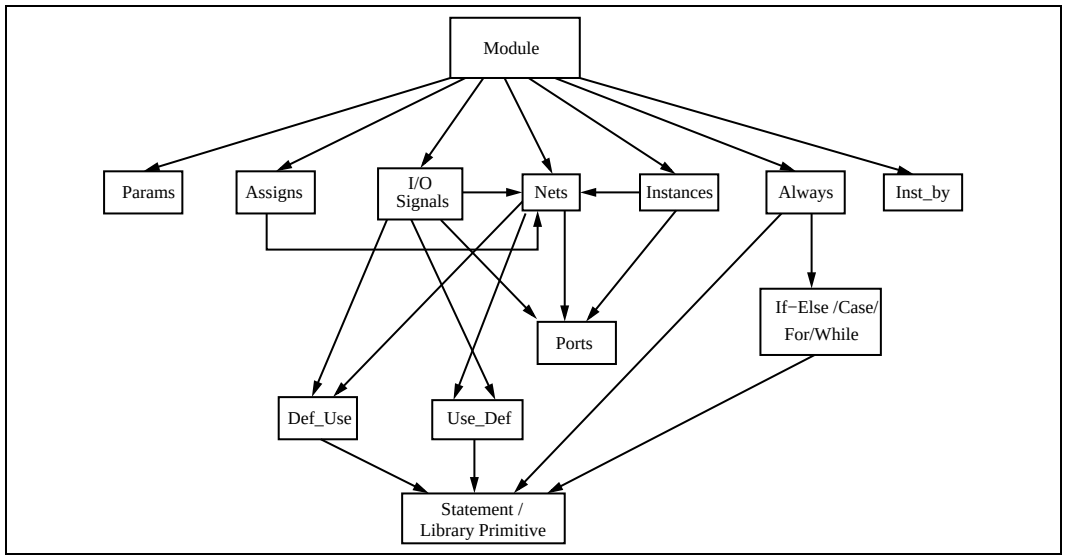

Figure 3.3: Internal Data Structure

LHS (of the assignment operator) signal, and the def-use chains of each RHS (of the assignment operator) signal respectively.

The algorithm is explained using the notation introduced and defined in the previous chapter. Each statement *i*, in a module is associated to its set B_C^0, and each signal *s*, in the statement is associated with the immediately relevant statements S_C^0, where C = $\langle i, s \rangle$. The statements in S_C^0 constitute the *def-use* and *use-def* chains of a program. The data structure constructed by the parser is shown in Figure 3.3. Note that the leaf nodes of this connectivity tree are either Verilog statements or library primitives.

The slices are hierarchically derived for each signal, and an executable subset of the surrounding logic is produced. Let *s* be a signal in a process *P*. For each input signal of the MUT, the recursive subroutine *find_bck_slice (s, P)* is called, and for each output signal, the recursive subroutine *find_fwd_slice(s, P)* is called. Let D^0

be the function that maps the pair *(s, P)* to the set of pairs (s_d, P_d), where each signal in s_d represents an immediately relevant signal of *s* present in the corresponding process in P_d, where s is defined. Let U^0 be the function that maps the pair *(s, P)* to the set of pairs (s_u, P_u), where each signal in s_u represents an immediately relevant signal of *s* present in the corresponding process in P_u, where s is used. A skeleton of the algorithms used in the recursive subroutines is given below.

```
/* Compute Backward Constraint Slice */
find_bck_slice (s, P) {
  Compute (s_d, P_d) = D^0(s, P)
  Search/Save S^0 for each (s_d, P_d)
  For each S^0, find/save B^0
  For each signal u ∈ U(S^0),
   find_bck_slice (u, P_d) /*RHS signals*/
  For each signal p ∈ U(B^0),
   find_bck_slice (p, P_d) /*Control signals*/
}

/* Compute Forward Constraint Slice */
find_fwd_slice (s, P) {
  Compute (s_u, P_u) = U^0(s, P)
  Search/Save S^0 for each (s_u, P_u)
  For each S^0, find/save B^0
  For each signal u ∈ U(S^0), u ≠ s_u,
   find_bck_slice (u, P_u) /*RHS signals*/
  For each signal d ∈ D(S^0),
   find_fwd_slice (d, P_u) /*LHS signals*/
```

```
  For each signal p ∈ U(B^0),
   find_bck_slice (p, P_u) /*Control signals*/
}
```

The backward slices obtained on all input signals of the MUT are composed to provide the input constraint slice, which defines the justification paths for the inputs of MUT. Similarly, the forward slices obtained on all output signals of the MUT are composed to provide the output constraint slice, which defines the propagation paths. The PIERs in the design are identified manually and are provided to the tool as pseudo primary inputs and outputs. The computation of the slices terminates if any of the primary inputs or outputs (as the case may be) are encountered.

Although slicing preserves the necessary statements and signals necessary for HDL analysis, a forward slice thorough a *case, casex, casez* and the *if-then-else* constructs, may remove some of the irrelevant branches. If these branches happen to be the default branch to be taken, slicing may introduce asynchronous latches during synthesis that can potentially produce errors during test generation. In these cases, a default statement is generated which assigns “dont care” value to the signal that was assigned in the other branch(s).

During constraint slicing, the tool also provides a trace of the signals on an aborted path, if any. This information provides valuable and early insight into the testability of the design, and the designer could choose to make suitable modifications to remove these testability bottlenecks.

3.3 Complexity Analysis

The complexity of the FACTOR algorithm lies in building the data structure and calculating the backward and forward slices. Let the total number of statements in the program be N_s and signals N_g.

To build the data structure

Space complexity: In the worst case, the def-use and use-def chains for each signal consist of all the statements in the program, resulting in $O(N_s)$ complexity. Using a similar argument, the complexity to build the MT set for each statement is also $O(N_s)$.

Time complexity: The use-def chain of the LHS signal adds the statement to itself. Therefore, the worst case complexity is $O(N_s)$. However, for each statement, all the RHS signals need to be examined to add the statement to the respective def-use chains, resulting in $O(N_s \cdot N_g)$. The MT set is obtained by traversing the parse tree which may contain all the statements, and hence the complexity is $O(N_s)$.

To calculate the backward constraint slice

Space complexity: The worst case is when all the statements in the surrounding logic occur in this slice, which is $O(N_s)$ in space.

Time complexity: The statements in the use-def chain of the signal are obtained in constant time. For each of these statements the MT set is also obtained in constant time. For each reaching definition (RHS signals and signals in the MT set) the combined set of statements (all of N_s in the worst case), the backward slice is computed. Therefore, the complexity is $O(N_s \cdot N_g)$.

To calculate the forward constraint slice

Space complexity: This is the same as computing the "Backward Constraint Slice": $O(N_s)$.

Time complexity: The statements in the def-use chain of the signal are obtained in constant time. For the LHS signal in each statement, the forward slice is computed in $O(N_s)$ time. For each statement in the def-use chain, the MT set is obtained in constant time. For each unique reaching definition (RHS signals different from the def-use chain index + signals in the MT set) the backward slice is computed in $O(N_s \cdot N_g)$ time.

3.4 Experimental Results

A Verilog RTL model of the ARM-2 processor [57] was used as the benchmark to study the effectiveness of FACTOR. The proposed technique was applied to modules embedded two or more levels in the hierarchy. Table 3.1 gives the characteristics of the modules under test.

Table 3.1: Modules in ARM

Module Name	Hierarchy Level	Primary Inputs	Primary Outputs	Gates in Design	Gates in Surrounding Logic	Stuck-at Faults
ALU	2	77	36	3836	11463	11868
RF_STR	3	101	99	7641	9658	28260
EXC	2	8	3	15	17284	108
FWD	2	29	4	84	17215	548

A commercial ATPG tool was used to generate test patterns targeting the stuck-at faults in the MUT, using the original design (without slicing). The results presented in Table 3.2, are generated using a 450 MHz UltraSPARC-II dual processor with 1 GB RAM. The processor level test generation was not run until completion, as it would be too time consuming. However, it would be sufficient

Table 3.2: Test Generation using the Original Design

Module Name	Processor Level		Stand-Alone	
	Test Coverage (%)	Time (s)	Test Coverage (%)	Time (s)
ALU	0.19	18.6	99.8	0.7
RF_STR	0.16	66.9	81.67	1.2
EXC	66.67	5.8	100	0.5
FWD	0.23	5.8	100	0.5
Average	0.83	14.3	95.02	0.68

to examine the test coverage versus time (in system CPU seconds) taken by the ATPG tool to understand the difficulty in test generation for modules embedded in a fairly large design. The average (geometric mean) of the test coverages and times illustrates the complexity of sequential ATPG.

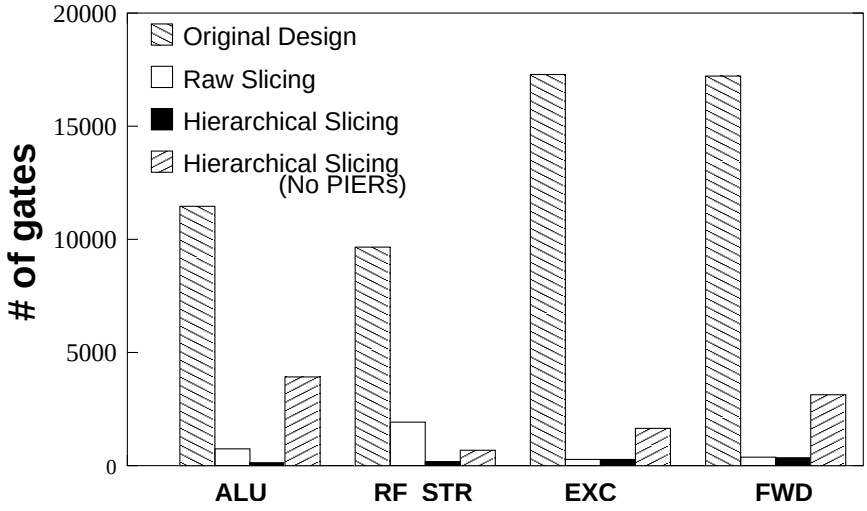

Figure 3.4: Reduction in Surrounding Logic

FACTOR was used to derive the required constraint slices. Figure 3.4 shows the reduction in surrounding logic when constraint slices were extracted using the raw technique (without composition of slices), and when composed using the hier-

archical technique. The composition theorem validates the hierarchical technique to obtain the desired constraint slice. The hierarchical technique was also applied without using PIERs to study it contribution to the methodology.

The modules ALU, EXC and FWD are instantiated at the same level of hierarchy in the same module. The slices obtained at a higher level of hierarchy during slicing on ALU were reused for constraint slicing on EXC and FWD. Such a constraint slice provides the required subset (consistency theorem) of the original program. The higher level slices were also used to derive the constraint slice on RF_STR, which is embedded deeper in the same hierarchy.

Constraint slicing was performed using a 1 GHz Athlon processor with 256 MB RAM, while the UltraSPARC-II processor (specified previously) was used for synthesis. Table 3.3 shows the time (in system CPU seconds) for constraint slicing and synthesis. Note that hierarchical slicing time may be marginally higher since it involves multiple steps. However, this difference is substantially compensated during synthesis and test generation due to the smaller amount of surrounding logic obtained. Note that the partial slices at each level of hierarchy include a smaller and a more relevant set of statements in the instantiated modules. In contrast, the instantiated modules during raw slicing include a union of all the statements required by various instantiations. Therefore, the final constraint slice obtained through hierarchical slicing is smaller than the slice obtained through raw slicing.

The slices were synthesized and combined with the respective gate level MUTs. The commercial ATPG tool (same as the one used on the original design) was used to generate test patterns targeting the faults in the MUT. The consistency theorem implies that the resulting patterns may be translated to the full-chip level without any loss of test coverage. Table 3.4 shows the test generation results using

Table 3.3: Slicing and Synthesis Times

Module Name	Raw Slicing		Hierarchical Slicing			
			PIERs		No PIERs	
	Slicing Time (s)	Synthesis Time (s)	Slicing Time (s)	Synthesis Time (s)	Slicing Time (s)	Synthesis Time (s)
ALU	0.21	1.23	0.23	0.73	0.78	2.46
RF_STR	1.60	0.98	0.53	0.72	1.97	2.34
EXC	0.51	0.69	0.32	0.75	0.42	0.85
FWD	0.82	0.73	0.40	0.71	0.93	1.18

raw slicing. Tables 3.5 and 3.6 show the corresponding results using hierarchical slicing, with and without PIERs, respectively. The abort limit was maintained the same in all the three situations.

Table 3.4: Test Generation using Raw Slicing

Module Name	Test Coverage (%)	ATPG Efficiency (%)	ATPG Time (s)	Total Time (s)
ALU	79.49	86.02	5.8	7.24
RF_STR	16.03	22.49	23.0	25.58
EXC	100.00	100.00	0.7	1.9
FWD	87.12	87.66	0.6	2.15
Average	57.72	64.17	2.74	5.25

The total time for each MUT includes the time taken for constraint slicing and synthesis. The stuck-at test coverage and the tool efficiency improve significantly while achieving drastic reduction in the overall test generation time. In the particular case of the module *RF_STR*, hierarchical slicing (with and without PIERs) improved the coverage significantly. Hierarchical slicing, enhanced with PIERs, was able to reduce the overall test generation time by a factor of almost 10. This module is the biggest among the modules considered and the most deeply em-

Table 3.5: Test Generation using Hierarchical Slicing with PIERs

Module Name	Test Coverage (%)	ATPG Efficiency (%)	ATPG Time (s)	Total Time (s)
ALU	80.87	88.58	3.0	3.96
RF_STR	81.49	84.56	1.4	2.65
EXC	100.00	100.00	0.5	1.57
FWD	94.91	95.12	0.6	1.71
Average	88.93	91.87	1.06	2.3

Table 3.6: Test Generation using Hierarchical Slicing without PIERs

Module Name	Test Coverage (%)	ATPG Efficiency (%)	ATPG Time (s)	Total Time (s)
ALU	47.62	63.11	8.6	11.84
RF_STR	71.09	78.74	20.4	24.71
EXC	86.40	89.05	0.9	2.17
FWD	59.23	65.89	1.2	3.31
Average	64.14	73.48	3.71	6.77

bedded in the design, which shows that the hierarchical approach is useful to deal with large hierarchical designs. Also, note that the coverage and test generation times for the module EXC remain about the same in all the three cases. In this case, the raw slicing was by itself able to achieve the smallest possible slice from the surrounding logic.

The use of PIERs improved (in a majority of the modules targeted) the test coverage using both raw and hierarchical slicing. This is mainly because slicing terminates at the PIERs, and thereby eliminates additional registers that would be present if slicing were to terminate only at the primary inputs/outputs, *i.e*, without using PIERs. This fact is illustrated by the results using hierarchical slicing without PIERs, which show that the test generation times are higher than those enhanced by PIERs.

During the process of constraint slicing, the tool also helps estimate any loss in fault coverage. For example, the coverage obtained for the *ALU* module is restricted to about 80% because 10 out of its 13 inputs are driven by signals decoded from a single *alu_operation* signal. FACTOR flags a warning in such cases and provides details about the MUT signal that is affected along with a trace of all the signals in the aborted path. This information can be exploited to modify/add design elements to improve testability.

The results on the benchmarks illustrate the various aspects of the methodology. All the three slicing approaches helped obtain a higher test coverage per unit time. The averages (geometric means) of the test coverage and time prove that constraint slicing reduces the ATPG complexity to an acceptable level. The slicing methodology produces a smaller design through which the ATPG tool effectively identifies the COI for each fault site, and generates the test pattern in a reasonable amount of time. These results validate the overall test generation methodology based on program slicing, and this technique holds great promise to reduce test generation complexity for large designs.

Chapter 4

Fundamentals of Bounded Property Checking

Formal verification depends on the recognition that programs are mathematical objects, with well-defined semantics and behavior. Therefore, in principle it should be possible to prove formally that programs are correct with respect to certain specifications which define the allowed behaviour of the system. There are two main approaches to specifying hardware behaviour, namely, (1) use temporal logic [60] to express "properties", and (2) generating a high-level model of the system [31]. Correspondingly, verification is performed by either showing that the system satisfies the properties or is consistent with the high-level model.

Model checking [62] takes the first approach, *i.e.*, specifying properties in temporal logic and using them to validate the system behavior. Temporal logic is a logic (usually propositional or first order) augmented with temporal modal operators, which allow reasoning about how the truth values of assertions change over time [58]. This logic can be used to express properties of a system behavior, and

can be broadly classified as safety and liveness properties [59].

- A **safety property** expresses the fact that *something bad will never happen*.
- A **liveness property** expresses the behavior whereby *something good will eventually happen*.

For instance, this logic can be used to express the assertion that if proposition p holds in the present, then proposition q holds at some instant in the future. There are several ways of specifying properties using temporal logic depending on the underlying model of time and the use of temporal operators.

In *Linear Time Logic* (LTL), the notion of time is that of a linearly ordered set which can be thought of as a possible sequence of states. Four operators are used to describe LTL properties in dealing with hardware verification, *viz.* **F**, **G**, **U** and **X**. These formulas are defined recursively. If p and q are LTL formulas, then

- $\mathbf{F}q$ is true in the present if at some moment in the future q is true.
- $\mathbf{G}q$ expresses the fact that q is true at every moment of the future.
- $p\mathbf{U}q$ means that q will hold true at some moment in the future until which time p will hold at all moments.
- $\mathbf{X}q$ is true in the present if q is true in the next instant of time.

Additionally, $\neg p$, $p \vee q$ and $p \wedge q$ are also LTL formulas. The **U** operator is called *weak until* if q does not necessarily hold in the future, and *strong until* if q definitely holds in the future.

Another temporal logic framework, *Computational Tree Logic* (CTL) is used to express the fact that at each instant of time there exist many possible futures [61].

In addition to the operators defined for LTL properties, it uses the "universal" and "existential" quantifiers, $\mathbf{A}$ and $\mathbf{E}$, to express properties over a discrete model of time. Each branch is defined as a maximal linearly ordered set of states. For example,

- q is *necessarily* true along all branches is represented by the formula $\mathbf{AG}q$.
- q is *possibly* true at least along some branch is represented by the formula $\mathbf{EG}q$.

Figure 4.1 illustrates some of the CTL formulas. Truth or falsehood of tense formulas are thought of as being relative to a given branch of the tree ordered frame. For example, $\mathbf{EG}q$ is true in the state represented by the root node, if there exists a sequence of states for which q holds in each one of them.

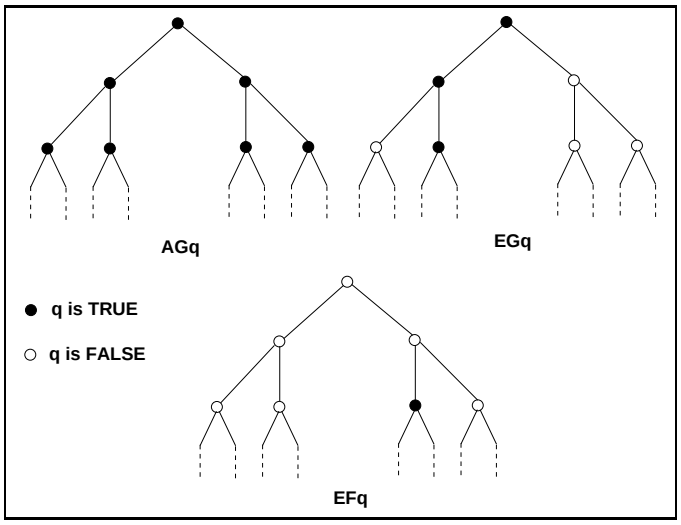

Figure 4.1: Illustration of CTL formulas

To illustrate the use of these operators to express safety and liveness properties, consider the design of a processor. It fetches instructions, decodes them,

fetches operands, executes the instructions, writes the results back and returns to the fetch state. The design of the control for the processor includes the above sequence of states, and may also include illegal states (depending on the state assignment).

A *safety property* could be that the processor never enters an illegal state, say S_i. This can be expressed as $\mathbf{G}\neg S_i$ in LTL. In order to find a counterexample (a bug) to this property, it is sufficient to check for the existence of a witness for the CTL property, $\mathbf{EF}S_i$, *i.e.*, along some execution path, at some time in the future, the processor enters an illegal state.

A *liveness property* could be that the processor never hangs, *i.e.*, it will always return to the fetch state, S_f. This can be expressed in LTL as $\mathbf{F}S_f$. A counterexample for this would be an infinite sequence of states which does not include the fetch state, expressed in CTL as $\mathbf{EG}\neg S_f$, *i.e.*, along some execution path, the fetch state is never reached.

4.1 Model Checking

The process of analyzing a design for the validity of properties stated in temporal logic is called model checking [62]. The input to a model checker is a formal description of the design, and the result is a set of states which satisfies the given property, or a witness of a sequence which violates the property.

Model checking can be done by explicitly checking if the property holds for each state, but this is possible only for designs with small numbers of states. McMillan [63] developed efficient techniques to manipulate Boolean formulas in model checking using Ordered Binary Decision Diagrams (OBDDs). The use of OBDDs allows the analysis of designs without explicitly enumerating their states. However, OBDDs are vulnerable to the state explosion problem for even moderately complex

designs [30].

In practice, the primary benefit of model checking is finding bugs rather than providing formal proofs establishing design correctness. Moreover, designers can usually define the bounds on the number of steps within which a property should hold. This leads to the idea of *Bounded Model Checking* which exploits these ideas for a more pragmatic approach.

4.2 Bounded Model Checking

In this paradigm, a finite prefix of a path, that may be a solution to an existential model checking problem, is considered. For instance, checking the "universal" property $\mathbf{F}(x = 0)$ is equivalent to finding a witness to its dual which is the "existential" property $\mathbf{EG}(x \neq 0)$. The key idea is based on searching for a counterexample in a finite "unfolding" (with respect to time steps) of the design under verification. By systematically increasing the unfolding depth to a bounded integer *k*, this approach checks whether the target can be reached in *k+1* steps. The approach suggested by Biere *et al.* [36] introduced a SAT based bounded model checking procedure for checking LTL properties. It involved a two stage process of (1) generating a set of propositional clauses from the design under consideration, and (2) using an efficient SAT solver to find a satisfying variable assignment in polynomial time.

The approach avoids the expensive fixed-point computations in the model checking algorithms, while being practically useful for verifying temporal properties. Copty *et al.* [38] described the benefits of bounded model checking at Intel, on designs taken from the Pentium 4 processor. A BMC checker with a SAT solver is reported to be superior to one with a BDD package. This approach becomes com-

plete and proves unreachability of the target, if it can be conclusively proven that the diameter of the STG is no larger than the largest bound for which the property has been checked. Structural algorithms to estimate a tight bound on this diameter [39] with a linear sweep of the design has shown great promise in making this approach more robust and viable for verifying large designs.

Current implementations rely on SAT techniques, which were originally invented to analyze Boolean expressions and not hardware circuits. Therefore, it is imperative for any CAD technique which uses a SAT solver, to translate the problem into a Boolean formula usually represented by propositional clauses described in conjunctive normal form (CNF). However, the CNF for a function given as a circuit is in general exponentially larger than the circuit itself [64] because it introduces variables for each internal wire. This also means that a SAT-based approach for hardware verification cannot make use of the structural information of the design that is inherently present in its HDL program. Additionally, sequential designs need to be time-wise unrolled based on the pre-determined set of time frames, into a purely combinational circuit. This is a serious limitation of this technique when applied to the large sequential circuits, which are prevalent today. These circuits also use artifacts such as tri-state logic which cannot be handled by native SAT solvers without extensive remodeling.

A simulation based approach for BMC was recently proposed to overcome some of these limitations has been proposed by Bingham and Hu [64]. The case-splitting heuristics, which have been the hallmark of SAT solvers, have been sacrificed for the capacity that simulation based techniques can offer. This may severely undermine its capability to establish unsatisfiability, thereby still leaving the need for structural techniques incorporating simulation such as sequential ATPG.

4.3 Symbolic Trajectory Evaluation

Traditional simulation of hardware designs to check their correctness is not only ineffective in catching subtle bugs, but is often a tedious process. Symbolic simulation, which uses symbols rather than actual values, was first proposed for hardware reasoning in the late 1970s [65]. However, it was the development of OBDDs [44], for efficient representation of Boolean functions and their manipulation, that gave impetus to this approach.

Ternary simulation of VLSI circuits uses a third value X to the set of possible signal values $\{0, 1\}$, to indicate an unknown or indeterminate logic value. This value X is a lower bound on every pair of values, and allows tremendous speedup of simulation as each pattern potentially represents more than one of the operating conditions. Information ordering was introduced for more efficient reasoning of the simulation results. In order to make all the operations monotonic over information ordering, an upper bound that completes the lattice structure, $\top$, representing an over-constrained logic value, was introduced to imply a node which is both 0 and 1 at the same time. Thus, a quaternary model $\{0, 1, X, \top\}$ was developed to efficiently represent and manipulate the simulation process.

4.3.1 Trajectory Formulas

A new generation of symbolic simulation-based hardware verification [35] uses a modified symbolic simulator to verify formulas described in a limited form of temporal logic. This logic allows the user to express properties of a circuit over a bounded-length of valid sequences (or *trajectories*) of circuit states. Formulas for describing these trajectories, or Trajectory Formulas (TFs) are defined recursively as follows.

- *Simple predicates.* If nd is a node in a digital circuit model, then nd **is** 0 and nd **is** 1 are trajectory formulas.

- *Conjunction.* If F_1 and F_2 are trajectory formulas, then so is $(F_1 \wedge F_2)$.

- *Next Time.* If F is a trajectory formula, then so is $\mathbf{N}F$.

- *Domain Restriction.* If F is a trajectory formula and E is a Boolean expression over symbolic variables, then F $\mathbf{when}$ E is a trajectory formula.

A conjunction has a natural interpretation as the least upper bound on the lattice elements. For example, if F_1 specifies that a node must be 0 at a particular time step, and F_2 specifies that the same node must be 1 at that time step, then $(F_1 \wedge F_2)$ specifies the node must be $\top$ at that time step. This may be flagged as an inconsistency in the specification. However, disjunction does not correspond to the greatest lower bound, and therefore disjunction and negation (which can be used with conjunction to imply a disjunction) are not allowed in TFs. Each TF is a set of 5-tuples, where each 5-tuple is of the form $< g, n, v, t1, t2 >$, where g is a Boolean guard condition, n is the name of a node in the circuit, v is a Boolean value, $t1$ is the start time and $t2$ is the end time. This tuple may be translated into the following statement:

Node $\mathbf{n}$ *has a value* $\mathbf{v}$ *from the time* $\mathbf{t1}$ *to* $\mathbf{t2}$ *under the condition that* $\mathbf{g}$ *is true. If* $\mathbf{g}$ *is false and the tuple is part of the antecedent, then the node* $\mathbf{n}$ *takes on the value of its next-state function applied to its present state. If* $\mathbf{g}$ *is false and the tuple is part of the consequent, then no check is performed on the node.*

The logic retains enough expressive power to describe timing and state transition information, but remains simple enough to be checked by a symbolic simu-

lator. It is designed as a compromise between expressive power and ease of evaluation.

4.3.2 Assertions

A trajectory assertion has the form $[\mathcal{A} \Rightarrow \mathcal{C}]$, where $\mathcal{A}$ and $\mathcal{C}$ are trajectory formulas representing the antecedent and consequent. An antecedent is a user constraint on the inputs and a consequent asserts the expected values of simulation on the output nodes. A trajectory assertion is implicitly universally quantified over any symbolic variables that appear in it. For example, the simple assertion $[in_{node}$ **is** $1 \Rightarrow out_{node}$ **is** $0]$ asserts that from any state in which in_{node} is 1, in the next state out_{node} must be 0.

More succinct specifications may be expressed by using symbolic variables to represent a set of such assertions. Other types of assertions [35] to describe *sequences* and *iterations* allow more expressiveness. A successful simulation establishes that each sequence of states in the circuit model which satisfies $\mathcal{A}$ also satisfies $\mathcal{C}$. Any mismatch is reported as a bug trace which encompasses the complete set of failing bugs, which is a very useful feature for debugging.

These assertions are usually expressed in a general purpose language suitable for the simulation engine. Complex assertions may be naturally expressed in a hierarchical manner using functions and procedure calls provided by the language. A modified event-driven symbolic simulator is sufficient to check the complex assertions on a target design.

Chapter 5

ATPG-Based Property Checking

Although ATPG techniques were originally developed for generating manufacturing tests, the capabilities of these techniques to successfully deal with complex designs has attracted considerable attention in other design automation domains. For example, ATPG algorithms have been successfully utilized [67] for logic synthesis, design rule checking, equivalence checking, and false path identification. However, the interest in using ATPG techniques for design verification has been restricted to a subset of properties.

A counterexample to a safety property implies something bad happens within the circuit. Using our processor example from the previous chapter, the "bad" event is that the processor goes into an illegal state. An ATPG tool could be used to search through the potential solution space, seeking to find the counterexample by justifying the illegal state back to an initial state.

The use of sequential ATPG for model checking was proposed by Boppana [40]. This work focused primarily on safety properties and studied the efficiency of sequential ATPG algorithms for state-space exploration. It was noted that the main benefits of using ATPG are (a) implicit storage of states at each time-

frame, and (b) an optimum balance between a breadth-first search and a depth-first search. Cheng and Krstic [67] discussed the use of ATPG for verification, including combinational equivalence checking and safety property checking. The properties suggested included tristate bus contention and asynchronous feedback loops. ATPG and SAT algorithms were compared by Parthasarathy *et al.* [68]. This work identified the tradeoffs between the two algorithms, and pointed out that ATPG could deal naturally with real-world primitives such as tri-state buses and high-impedance logic values. Hsiao and Jain [42] also suggested the use of sequential ATPG for verifying safety properties of the type $\mathbf{EF}p$ and compared this with OBDD-based approaches using a simulation-based ATPG with a genetic algorithm. All these approaches have shown that ATPG could perform the state-space search without needing the complete state-space information at one time.

Although checking for a safety property or more precisely, finding a counterexample to an invariant, can be done easily using ATPG techniques, it was never shown how liveness properties can be checked. A methodology [46] for overcoming this limitation is presented in the following section.

5.1 Proposed Approach

A key requirement for a technique to be usable by designers is that it should fit seamlessly into the normal design flow. Therefore, in this approach an existing ATPG tool is used without any modifications along with a circuit description which is compatible with the ATPG tool. In addition, a small circuit is used which will guide the ATPG tool in checking for a desired property.

Properties stated as temporal logic formulas of the type $\mathbf{A}p$, where p is a restricted path formula in which only state sub-formulas that are atomic propositions

are considered [69]. The proposed approach automatically maps both safety and liveness properties, as well as an upper bound on the number of steps to be verified, into a monitor circuit with a target fault. This allows any existing ATPG tool to be used to generate a test for the fault. A test for the fault becomes a witness for the property. If the fault is determined to be untestable, the property is guaranteed to hold within the bound. If ATPG aborts, *i.e.*, it neither generates a test nor proves that the fault is untestable, nothing can be concluded about the property.

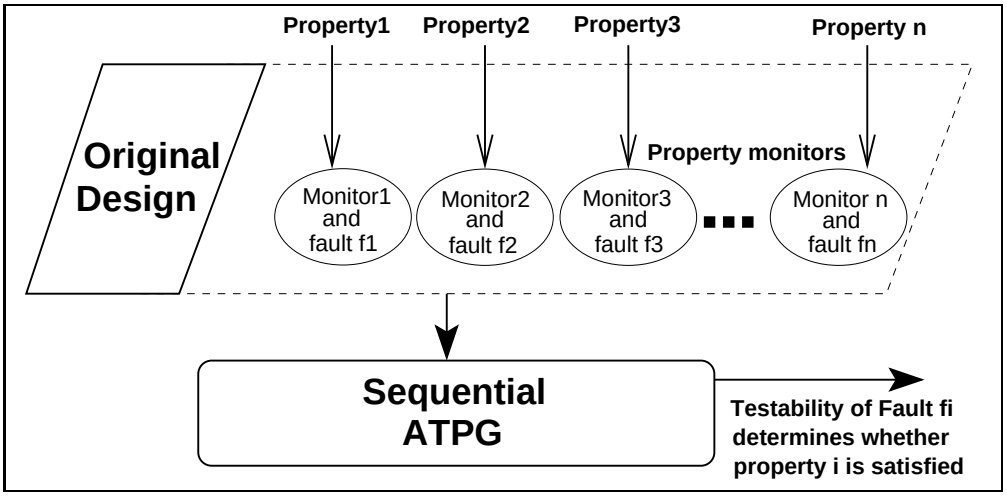

Figure 5.1: Property Verification Approach using ATPG

Figure 5.1 illustrates the essential features of this approach. Monitor circuits are generated for properties of interest using the relevant signal names from the original design, without modifying the original design or the ATPG tool.

ATPG is well suited for finding a test (or a witness) for a value on a chosen signal in a circuit. Therefore, checking an LTL property in the circuit can be reduced to finding a witness for the dual of the equivalent existential CTL property [36]. Thus, in order to test for an LTL property with a bound of n, $\mathbf{F}p$ (at some time in

the future within n cycles, p will be true), ATPG can be used to find a sequence of length n where p is not true (*i.e.*, a witness for $\mathbf{EG}\neg p$). The finite-state machines representing the monitor circuits for the properties, $\mathbf{EF}p$ and $\mathbf{EG}p$ are described in the next section for a bound of n. The property is satisfied if the final (or *accepting*) state is reached. The target fault for ATPG is the output of the gate which encodes this final state.

5.2 Property Monitors

Figure 5.2 illustrates the monitor for the $\mathbf{EF}p$ property. The search starts in the state indicated by 1, and if at any time p is satisfied, a transition is made to the accepting state indicated by n. ATPG is used to generate a stuck-at fault on the output of a gate encoding the final state n. A valid test generated for the fault indicates that the property is satisfied, and an untestable fault proves that there does not exist a state sequence, of length that is less than or equal to n, that satisfies the property p.

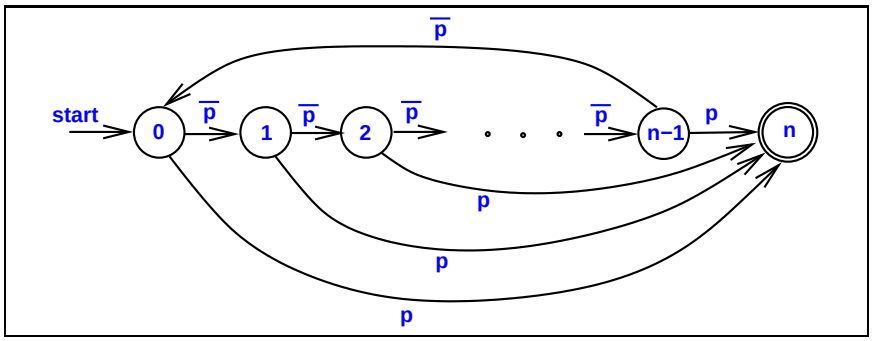

Figure 5.2: Monitor for $\mathbf{EF}p$

A similar monitor is used to check another property, (*bounded liveness*) denoted by $\mathbf{EG}p$, as illustrated in Figure 5.3 for a bound of n. Here, a transition to

the accepting state labeled n, is made only if p is satisfied at every other state in a state sequence starting from the start state. Therefore, a successfully generated test sequence proves the existence of a sequence where p is true at every state, and this state sequence is a witness to the property.

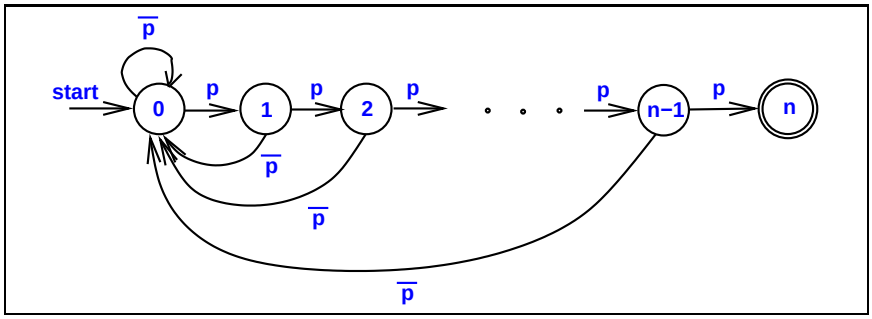

Figure 5.3: Monitor for $\mathbf{EG}p$

In a case in which the actual number of states in a witness sequence is longer than n, the last n states of the sequence are returned by ATPG. The design would be forced into the start state by ATPG using an initialization sequence.

5.3 Results

The technique was applied to the ISCAS 89 benchmark circuits. The property monitors have been described in Verilog, then synthesized to the gate level, and finally composed with the original design. A commercial sequential ATPG tool [70] was used to generate the witnesses. The results were compared with a free research version of a bounded model checker, which was originally developed at Carnegie-Mellon University, and is now available from Cadence Berkeley Labs [72]. All of the experiments were performed on a Sun Microsystems Ultrasparc II system with dual processors, each running at 450 MHz and with 1 GB of memory.

The properties of the ISCAS 89 benchmark circuits were derived by logical composition of all the outputs without any specific distinction, as sufficient to evaluate this approach. LTL properties, without nesting, have been used as required by Cadence-SMV. BMC for LTL properties is accomplished by searching for a witness to the dual of the corresponding existential CTL property.

For each of the benchmark circuits, the safety property that was considered was $\mathbf{G}((o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 0)$, where o_i is the ith output of the circuit, *i.e.*, for all times in the future there exists at least one output that is zero. ATPG was used to find a witness for $\mathbf{EF}((o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 1)$, *i.e.*, there exists a future state in which all of the outputs are 1 simultaneously. Table 5.1 shows the results of this experiment. The first four columns give the circuit name and its details such as the number of I/O pins, the number of combinational gates and the number of sequential elements respectively. The fifth column indicates whether or not a counterexample exists (indicated by "Y" or "N"); the property is satisfied for some circuits but not for others. The final two columns show the CPU times in seconds for both BMC and ATPG, respectively. BMC was unable to generate the SAT clauses for the largest circuits as indicated by a "-" in the table.

The liveness property considered was $\mathbf{F}((o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 0)$, where o_i is the ith output; *i.e.*, there exists a state sequence of length n within which at least one of its outputs is 0. ATPG was used to find a witness for the dual CTL property, *i.e.*, $\mathbf{EG}((o_1 \cdot o_2 \cdot \ldots \cdot o_k) = 1)$, *i.e.*, a sequence of states of length n, in which at each state all the outputs are simultaneously 1.

The results are presented in Table 5.2 for different values of the bound with n = 5, 15, and 25 respectively. For each bound, the first column indicates whether a counterexample to the property exists (indicated by "Y" or "N"), while the next two columns give the times taken by BMC and ATPG, respectively.

Table 5.1: Checking EF Property for AND of All Outputs (Bound = 15)

Circuit Name	Primary In + Out	Comb. Gates	Seq. Elem.	Counter-Example?	BMC CPU(s)	ATPG CPU(s)
s27	6+1	10	3	Y	0.22	0.1
s820	20+19	289	5	N	1.52	0.1
s832	20+19	287	5	N	1.49	0.1
s1488	10+19	653	6	N	2.61	0.2
s1494	10+19	647	6	N	2.73	0.1
s386	9+7	159	6	N	0.72	0.1
s510	21+7	211	6	N	0.82	0.1
s208.1	12+1	104	8	Y	0.73	0.1
s298	5+6	119	14	N	0.96	0.2
s344	11+11	169	15	N	3.32	0.1
s349	11+11	170	15	N	4.64	0.1
s420.1	20+1	218	16	Y	1.53	0.1
s1196	16+14	529	18	N	3.17	0.1
s1238	16+14	508	18	N	3.27	0.1
s641	37+24	380	19	N	2.68	0.1
s713	37+23	393	19	N	2.43	0.1
s382	5+6	158	21	N	1.25	0.2
s400	5+6	164	21	N	1.26	0.1
s444	5+6	181	21	N	1.44	0.1
s526	5+6	193	21	N	1.68	0.2
s953	18+23	395	29	N	3.4	0.2
s838.1	36+1	446	32	Y	3.48	0.1
s1423	19+5	657	74	Y	5.15	2.2
s5378	37+49	2779	179	N	10.22	0.3
s9234.1	38+39	5597	211	N	24.5	0.3
s9234	21+22	5597	228	N	24.29	0.3
s15850.1	79+150	9772	534	N	78.23	1.1
s15850	16+87	9772	597	N	76.85	0.9
s13207.1	64+152	7951	638	N	65.63	0.7
s13207	33+121	7951	669	N	64.14	0.7
s38584.1	40+304	19253	1426	N	-	3.6
s38584	14+278	19253	1452	N	-	2.9
s38417	30+106	22179	1636	N	-	28

Table 5.2: Checking EG Property for AND of All Outputs (Various Bounds)

Circuit Name	Bound = 5			Bound = 15			Bound = 25		
	Ctr.-Ex.	BMC sec.	ATPG sec.	Ctr.-Ex.	BMC sec.	ATPG sec.	Ctr.-Ex.	BMC sec.	ATPG sec.
s27	Y	0.21	0.1	N	0.21	0.2	N	0.31	0.2
s820	N	0.91	0.1	N	1.21	0.1	N	1.75	0.2
s832	N	0.92	0.1	N	1.34	0.1	N	1.73	0.1
s1488	N	1.54	0.1	N	2.02	0.2	N	2.95	0.2
s1494	N	1.51	0.2	N	2.05	0.1	N	3.1	0.2
s386	N	0.46	0.1	N	0.58	0.1	N	0.88	0.1
s510	N	0.54	0.2	N	0.8	0.2	N	1.09	0.1
s208.1	Y	0.4	0.1	N	0.65	0.3	N	1.08	0.3
s298	N	0.42	0.5	N	0.59	0.4	N	0.93	0.4
s344	N	0.86	0.1	N	1.73	0.1	N	4.73	0.2
s349	N	0.89	0.1	N	1.91	0.1	N	4.39	0.1
s420.1	Y	0.71	0.2	N	1.47	0.3	N	2.73	0.4
s1196	N	1.2	0.1	N	1.73	0.1	N	2.35	0.2
s1238	N	1.24	0.2	N	1.74	0.1	N	2.54	0.1
s641	N	0.91	0.1	N	1.24	0.1	N	1.86	0.1
s713	N	0.94	0.2	N	1.19	0.1	N	1.57	0.1
s382	N	0.56	0.2	N	0.68	0.1	N	0.98	0.1
s400	N	0.52	0.1	N	0.71	0.1	N	0.98	0.1
s444	N	0.51	0.2	N	0.69	0.1	N	1.01	0.1
s526	N	0.64	3.6	N	3.94	4.2	N	168.6	3.5
s953	N	1.16	0.1	N	1.68	0.2	N	2.16	0.2
s838.1	Y	1.57	0.1	N	2.88	0.3	N	5.28	0.5
s1423	N	1.75	2.2	N	2.63	2.3	N	3.65	2.2
s5378	N	7.13	0.2	N	8.77	0.3	N	11.75	0.3
s9234.1	N	10.83	0.5	N	12.53	0.5	N	14.95	0.5
s9234	N	10.49	0.4	N	12.34	0.5	N	14.82	0.5
s15850.1	N	32.08	0.9	N	36.77	1	N	45.55	0.9
s15850	N	32.81	1	N	36.87	0.799	N	46.41	1
s13207.1	N	25.99	0.6	N	30.95	0.6	N	38.68	0.7
s13207	N	25.67	0.6	N	30.73	0.699	N	38.6	0.5
s38584.1	N	-	3.4	N	-	3.3	N	-	3.2
s38584	N	-	2.8	N	-	2.9	N	-	2.7
s38417	N	-	3	N	-	2.9	N	-	2.8

The peak memory usage for BMC (SMV + the "Zchaff" SAT solver) was approximately 150 MB for the largest circuit that it could handle, *i.e.* s13207. It could not handle the larger designs. In contrast, the peak memory requirement for ATPG, including the largest design, is only about 50 MB. However, for the smallest design, *i.e.* s27, ATPG requires up to 34 MB, while BMC requires only around 1 MB. The SAT-based approach requires less memory for very small circuits, however the memory requirement grows very quickly as the number of generated clauses grows with the size of the circuit and the bound, n. In contrast, the memory requirement for ATPG grows less rapidly with the increasing design size, as its primary memory requirements are for the representation of the circuit structure and the values of the nodes.

Finally, verification at the gate level using structural techniques such as ATPG, allows proving properties in designs with "real" artifacts such as tri-states, multiple clocks and switches.

5.4 Selecting Bounds

Table 5.2 also highlights some interesting trends in the behavior of the two approaches as the bound, n, is increased. As described previously, the recommended approach when using bounded model checking is to start with a small bound, look for a counterexample, and then to progressively increase the bound for checking whether the property holds for larger and larger bounds. The time required by BMC increases as the bound is increased. In particular, for circuit s526, the time increases from less than a second for $n = 5$ to over 150 seconds for $n = 25$. In contrast, the time required by ATPG remains approximately the same, independent of the bound for the circuit. In real designs with large sequential depths, it is not

always easy to guess an accurate initial value for the bound. An ATPG-based approach can start with a relatively large bound for n without incurring a large penalty in CPU time. Recent techniques in estimating the diameter of the STG [39] could provide this bound *a priori*, guaranteeing the completeness of the methodology for property checking.

5.5 SAT Versus ATPG Engines

Both SAT and ATPG are computationally expensive algorithms and belong to the class of NP-complete problems [73, 74]. However, in practice several heuristics have been proposed for both of these techniques to prune the search space effectively. These solutions are based on search techniques in which all possible combinations of inputs are searched implicitly.

Most leading, non-commercial SAT solvers are based on the Davis-Putnam Procedure [75], which is based on the identification of unit clauses and on computing the resulting implications. The best known version of this procedure is based on a backtracking search algorithm that at each node in the search tree, selects an assignment and then prunes the subsequent search space by iteratively applying the unit clause and the pure literal rules [76]. Another SAT solver GRASP [77] combines previously proposed search/pruning techniques and identifies some new ones. Additionally, GRASP uses a powerful conflict detection procedure and records the causes of conflicts to increase the speed of pruning. Zchaff [78] is a high performance SAT solver that achieves an improvement of two orders of magnitude over previous SAT solvers through the clever and efficient implementation of *Boolean Constraint Propagation* (BCP) and also through low overhead decision strategies. More recently, a hybrid approach [79] combining the advantages of both structural

and *Conjunctive Normal Form* (CNF) based algorithms was shown to be faster than ZChaff.

Combinational ATPG algorithms [1], such as D and PODEM, use the structural information of a design for decision ordering, justifying and propagating the decision assignments. These algorithms have comparable complexity to the SAT techniques. The search algorithms employed for sequential systems use a modified version of a combinational circuit search system. Figure 5.4(b) illustrates an *Iterative Logic Array* (ILA) model of the sequential circuit shown in Figure 5.4(a). The ILA model is an unrolled version of the sequential circuit in which the flip-flops are replaced by environment variables S_i and PI_i, which represent the state of the circuit and its primary inputs, respectively at time i. In this model, a copy of the circuit at time i is referred to as a $frame\ i$. Note that every frame is a combinational circuit with primary inputs consisting of the set (PI_i, S_i).

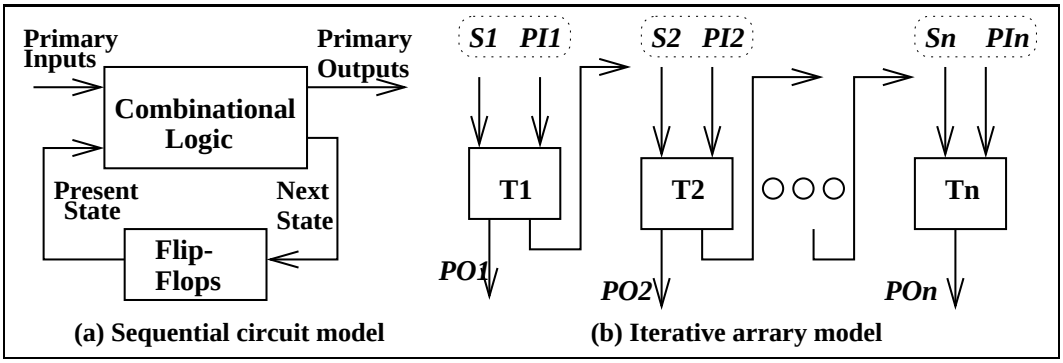

Figure 5.4: A Sequential Circuit and its ILA Model

The algorithms developed for SAT and combinational ATPG could be applied to the ILA model, but this naive approach is inherently limited in the size of the designs that it can handle. The number of time frames for which the designs need to be unrolled grows exponentially large as the number of sequential elements

increase. However, since all the time frames have the identical structure, there is no need to actually construct the complete model of the ILA. Instead, a single copy of the structural model is stored and only the signal values in different time frames need to be maintained. Moreover, the time frames can be incremented in a step-wise manner to find the shortest possible test which is more memory efficient than unrolling the design as described previously.

An efficient ATPG search algorithm uses an ILA model and assumes that the circuit starts in an unknown state, *i.e.* all "Xs", then finds a reset or synchronization sequence to a known state. In the property checking framework, the ATPG starts the circuit in the all "X" state and searches for a sequence of input vectors that will activate the circuit property by producing a logic one at the output of the property monitor. This framework is consistent with the one used in classical BMC that tries to find the set of states which satisfy or contradict a property. If the search determines that the monitor output cannot be set, then it is guaranteed that there is no sequence of states of length $\leq n$ which will satisfy or contradict the property.

It is clear that only the sequential ATPG state justification phase is needed in order to prove that a property is satisfied. The memory requirements are also significantly lower for ATPG, since unlike SAT, ATPG needs only to store the states of the flip-flops and the primary inputs for each time frame [80].

Chapter 6

Design Verification using Slicing

The complexity of the verification process requires its division into simpler tasks by exploiting the modular structure of the system. This is often achieved by decomposing and abstracting both the implementation and specification. Such methodologies for hierarchical verification include *assume-guarantee* reasoning [81] and *compositional minimization* [82]. Compositional techniques have been shown to improve the scalability of model checking [83]. RTL reduction [84] to eliminate unnecessary portions of the circuit has been proposed to speedup practical verification using symbolic model checking. This section develops a similar approach using HDL slicing to improve the performance of the proposed ATPG-based approach for bounded model checking.

Sequential ATPG tools by themselves work well only on moderately large designs [85] and are not able to handle a complete processor design, for example. Therefore, in order to be able to use the ATPG-based property checking (as described in Chapter 5), a divide-and-conquer approach is desired to check the properties of individual components or modules within a large design. This would require that the counterexample generated for a property be translated to the system

interface. Additionally, the assumptions made about the interactions between the *Module Under Verification* (MUV) and its surrounding modules, need to be incorporated into the temporal logic specification.

The proposed approach extends the ATPG-based property checking for unbounded liveness properties and uses the program slicing methodology for accelerating test generation (described in Chapter 3) to provide a powerful framework for validating designs. The program slicing approach can be used to overcome the problems of modular bounded property checking when dealing with large designs. The constraint slice for a MUV encompasses its interactions with the surrounding environment and a synthesized constraint slice implicitly forces a gate-level ATPG tool to generate a witness (or a counterexample), which is valid at the system level. If the fault is untestable (with the constraint slice in place), then the property of the MUV holds true within the given bound.

Note that program slicing techniques can be used with the SAT-based approach for BMC as well. However, the size of the modules that can be handled by Cadence-SMV (using the ZChaff SAT Solver) is limited, as shown by the results in Chapter 5. This would require selecting smaller and more deeply embedded modules within a design, which would result in a surrounding environment that is too large for a tool such as Cadence-SMV to handle. Moreover, the approach has been designed to check LTL properties and may not be suitable for other kinds of properties that can be described using synthesizable HDL.

The constraint slicing approach also makes use of directly accessible internal registers, called *PIERs* (Primary Input/output accEssible Registers) [14] as pseudo-primary inputs/outputs for constraint slicing. The use of PIERs in this methodology will accelerate the generation of witnesses (or counterexamples) for properties. Note that the target fault is an output by itself; and therefore, the use of PIERs

as pseudo primary outputs is not required. It is sufficient to identify those registers which can be written into, and to be able to translate the generated witness (or counterexample) to the system level using the same load instruction. As the PIERs only act as additional inputs and provide better controllability of the target fault, an untestable fault implies that a test pattern cannot be found at the system level as well. Similarly, a successfully generated test pattern can be translated back to the chip-level, using the load instructions for the pseudo-primary inputs, and is therefore a valid witness (or a counterexample) for the property being checked.

6.1 Unbounded Liveness

The monitor circuit for $\mathbf{EG}p$, described in Figure 5.3, checks for *bounded liveness*, which is searching for a sequence of states where p is true, within the selected bound. However, *unbounded liveness* checking requires finding a loop in the STG, in which every state satisfies p. This means that the final accepting state arrived at by the witness, should be a state that has already been reached in the prior time steps. This is done using an additional set of N (where N is the bound) registers which is added to the synthesizable HDL description of the monitor by defining it as a memory. A valid sequence of the state encoding is stored by this memory as the search expands. At every clock cycle where the desired state is reached, it is compared against the set of prior valid states and any repetition is inferred as a loop. This approach performs exactly the same check as in SAT-based BMC approach to check for a loop in the STG [36].

An example generic monitor for checking unbounded liveness is given as follows. It checks for counterexamples with up to 31 steps, with a monitor state machine with up to 15 states. These values are shown for illustration, and can be

changed appropriately to suit the requirements in specific instances on BMC.

```
/* Property Monitor for Unbounded Liveness */
module monitor (clk, state, p, SA0);
  parameter N = 31;
  parameter R = 4;
  parameter M = 3;
  input clk, p;
  input [M:0] state;
  output SA0;
  reg [R:0] state_ctr, i;
  reg [M:0] state_mem [1:N];
  reg loop_flag;
  assign SA0 = loop_flag;
  always @(posedge clk)
    begin
    if (p && (state_ctr < N))
       begin
       state_ctr = state_ctr + 1;
       state_mem[state_ctr] = state;
       if (state_ctr == 12)
          begin:break
          for (i = N-1; i > 0; i = i-1)
              begin
              if (state == state_mem[i])
                 begin
```

```
                loop_flag = 1;
                disable break;
                end
            end
        end
      else loop_flag = 0;
      end
   else
      begin
      state_ctr = 0;
      loop_flag = 0;
      end
   end
endmodule
```

The property monitor for $\mathbf{E}p\mathbf{U}q$ which conforms to the standard definition of the "strong until" operator in CTL is shown in Figure 6.1. This property checks for a state such that there exists a finite sequence of inputs $i_0, i_1, i_2, \ldots, i_n$ such that q is true at the state resulting in the application of the input sequence $i_0, i_1, i_2, \ldots, i_n$ and p is true in all prior states. The argument for why this circuit would guide ATPG into finding a witness for the property is similar to the one for the $\mathbf{EG}p$ property above.

These improvements lead to an elegant approach to check properties of the type $\mathbf{A}p\mathbf{U}q$, which can be expressed using the existential operators of CTL as $\neg\mathbf{EG}\neg q \wedge \neg\mathbf{E}[\neg q\ \mathbf{U}(\neg p \wedge \neg q)]$. The necessary bounded check may be performed by searching for a witness to either $\mathbf{EG}\neg q$ or $\mathbf{E}[\neg q\ \mathbf{U}(\neg p \wedge \neg q)]$. It is easy to in-

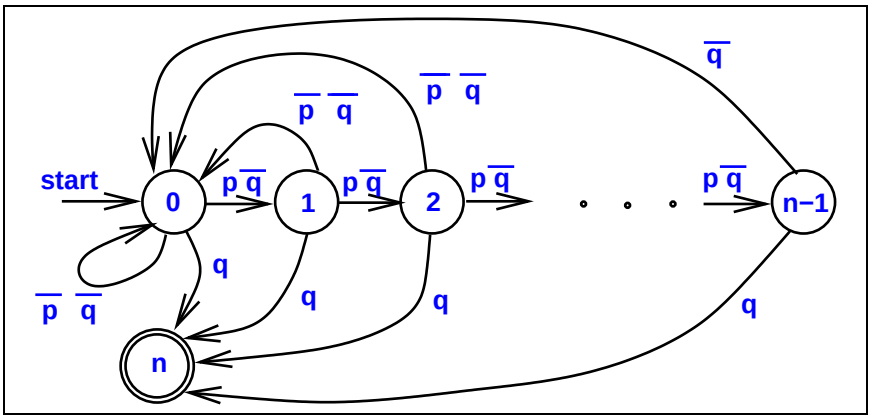

Figure 6.1: Monitor for $\mathbf{E}p\mathbf{U}q$

fer that the monitors for both of these "existential" properties (as described earlier) may be directly used to find a counterexample to this important property.

The property checking technique can easily be adapted to check "existential" CTL properties as well and therefore equally suitable for CTL property checking. This is achieved by searching for a witness to the existential property within a given number of time steps. If a counterexample is found, the property holds true; otherwise, the property does not hold within the specified bound. The methodology is also not limited to checking CTL or LTL properties only. It can be used to check other properties of the design that can be expressed in synthesizable HDL, but may not have direct correspondence to either of the formal languages. The unique nature of this approach makes it very powerful for verifying designs in practice.

6.2 Results and Analysis

The proposed methodology uses FACTOR (described in Section 3.2), which was implemented for generating constraint slices for test generation. The methodology

was implemented on a Verilog RTL model of VIPER (Verifiable Integrated Processor for Enhanced Reliability), which is a 32-bit microprocessor designed for safety critical applications [86].

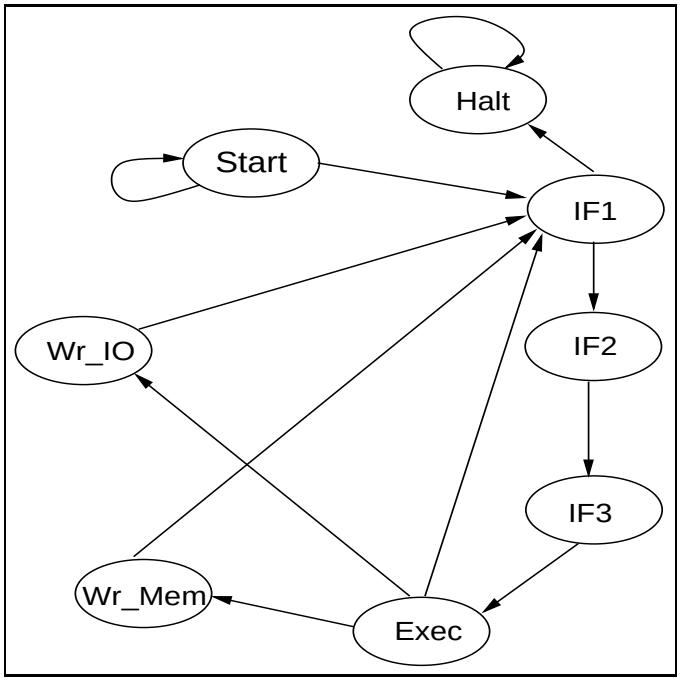

Figure 6.2: Simplified State Transition Graph for VIPER Control

The control module is chosen as the MUV, and its finite state machine is presented in Figure 6.2. The set of properties that have been checked are itemized as follows.

- **VP1**: There exists a finite input sequence resulting in a state where an exception is raised.
- **VP2**: There exists a finite input sequence resulting in the *halt* state.

- **VP3**: An instruction (after being fetched) requires at most 5 clock cycles to complete, if an exception is not raised.

- **VP4**: It is possible that the execution of an instruction may not return to the *fetch* state within 5 clock cycles, if an exception is allowed to be raised.

- **VP5**: It is possible to enter the *halt* state and remain there forever. Unbounded liveness is implied.

- **VP6**: If the system is in the *halt* state, there exists a finite input sequence resulting in some other state (without having to reset the machine).

- **VP7**: If the system is in a state other than the *Start* state, for every infinite sequence of inputs, $p = fetch \parallel assign\_r \parallel write\_addr$ is true at some resulting state. Unbounded liveness is checked using memories.

- **VP8**: It is always true that the STOP signal remains un-asserted until the system enters the *halt* state. The methodology for checking CTL properties of the type $\mathbf{A}p\mathbf{U}q$ is used here.

- **VP9**: The trivial case $\mathbf{EG}\neg q$ for checking the property **VP8** is eliminated and re-checked.

These properties were methodically chosen such that they check for important features (explained later in the section) of the MUV and eliminate the chance of any non-trivial solutions. The *fetch* state refers to *IF1* in the figure. Dummy outputs needed by the property monitors to observe the signals in the MUV were added to the original VIPER design and synthesized to the gate level. The PIERs in the design were identified and used as pseudo primary inputs during the generation of the constraint slice using FACTOR. The relevant slice of the design S_VIPER,

was generated by combining the MUV with the generated constraint slice and synthesizing it to the gate-level. The slicing (in the case of S_VIPER) and synthesis times taken for generating the two target designs, VIPER and S_VIPER, and their characteristics (combinational gates and sequential elements) are presented in Table 6.1.

Table 6.1: Benchmark Characteristics

Module Name	Comb. Gates	Seq. Elem.	Time (secs.)
VIPER	3202	219	76.80
S_VIPER	449	14	37.11

The times shown are in user CPU seconds and include synthesis time (using flatten mode) and slicing time (for S_VIPER) time for each target. All the experiments were conducted using a 450 MHz UltraSPARC-II dual processor with 1 GB RAM. Commercial tools were used for all gate-level synthesis and sequential ATPG.

The monitors for each property described have been implemented in synthesizable Verilog RTL, and synthesized to the gate level. A simple top level wrapper module was generated for each target, which instantiates the target as well as the monitor. The characteristics of the monitors and the synthesis times are shown in Table 6.2. Note that these monitors need to be generated only once and can be used for both of the target designs. Although the monitors have been generated for the current work, most of these monitors are generic and can be reused to check properties of other designs as well.

A commercial ATPG tool was used to generate a test sequence targeting the stuck-at fault at the output of each monitor, which determines the existence of

a witness or counterexample depending on the property. The results of the test generation for both of the targets are presented in Table 6.2.

Table 6.2: Bounded Property Checking with/without HDL Slicing

Property Checked	Monitor Generation			Test Generation			
	Comb. Gates	Seq. Elem.	Time (secs)	Test Found?	Property Valid?	Time (secs)	
						VIPER	S_VIPER
VP1	0	0	11.28	Yes	Yes	15.5	1.0
VP2	1	0	12.09	Yes	Yes	16.2	1.0
VP3	20	3	13.80	No	Yes	33.1	1.9
VP4	19	3	13.51	Yes	Yes	16.7	1.1
VP5	33	5	13.15	Yes	Yes	16.1	1.1
VP6	3	3	12.46	No	No	15.2	1.0
VP7	84	22	16.53	Yes	No	41.1	2.9
VP8	58	23	18.13	Yes	No	16.1	1.3
VP9	16	1	13.99	No	Yes	15.5	0.9

The time (in CPU seconds) taken for generating the monitors for each property and the characteristics of each monitor are shown in columns 2-4. Test generation was performed using these monitors composed with the original and sliced designs. If a test is found (indicated in column 5 by a “Yes”, otherwise a “No”) for the specified bound, column 6 indicates if the property being checked is valid or not (indicated in by a “Yes” or a “No”, respectively) for that bound. The last two columns show the test generation times (in CPU seconds) using the original and the sliced design.

In order to check for the property **VP3** and avoid generating a trivial counterexample, it is necessary to verify the existence of a sequence of inputs that raises an exception, and to verify that the system enters the *halt* state at least once. Properties **VP1** and **VP2** were verified for this purpose. The property **VP4** checks the validity of **VP3** while allowing an exception to be raised. The test pattern that was generated showed that this is indeed true, because the system enters the *halt* state.

The next logical step is to check if it was possible for the system to enter the *halt* state and remain there forever. The property **VP5** checked this condition for a bound of 25 and found that it was possible. Property **VP6** was verified to see if it was possible to come out of the *halt* state without having to reset the system.

The property **VP7** checks if one of the address lines (*fetch, assign_r, write_a--ddr*) is always assigned. This was achieved by checking for unbounded liveness of the condition $p = \neg(fetch \ \&\& \ assign\_r \ \&\& \ write\_addr)$. A 7-bit wide memory register was used to save the set of valid prior states and to check against the current state. A test was found for the dual ECTL property indicating that this property is not valid.

Properties **VP8** and **VP9** verify a property of the type $\mathbf{A}p\mathbf{U}q$ in order to demonstrate the approach suggested in Section 4. A test pattern was generated for **VP8** which q is never asserted, and this case was eliminated to produce the more interesting property **VP9**. These properties are useful to check if the system can halt only when an illegal instruction is fetched or when an overflow condition occurs.

All of the generated test patterns have been manually verified against the control flow in the state machine to ensure their correctness and validity. Note that the generated slices include the forward slices from the outputs of the MUV, which are not necessary for verification as the targeted fault is on the output (gate encoding the accepting state) of the monitor. The size of the slice, slicing and verification times would significantly change if the tool is suitably modified.

The results show a tremendous reduction in the time taken for property checking by applying program slicing. The benefits of both approaches [46, 50] have been systematically incorporated in the proposed methodology.

6.3 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation (STE) derives its strength from three factors: a simple specification language, a symbolic simulation based model checking algorithm, and a quaternary circuit model. The quaternary abstraction of the circuit assigns each node with a value from the four possible values $\{0, 1, X, \top\}$, where 0 and 1 represent specific, fully defined values (low and high voltages), X denotes an unknown value (or absence of information), and $\top$ indicates an over-constrained value [35].

Many real circuits do not always start from an initial state. Hence, any node without a value at a time in the antecedent is assumed to have an initial value of X. By doing so, the model checking complexity is vastly reduced with respect to circuit representation [45]. In general, a circuit with n Boolean nodes has 2^n unique states; therefore there are 2^{2^n} combinations of them. However, using the quaternary model, there are only 4^n quaternary assignments.

These performance improvements make this technique applicable to fairly large designs. However, the BDD-based representation for state-space can grow exponentially with the size of the operations, especially for complex circuits such as adders and multipliers. Therefore, sequential circuit verification needs hierarchical methodologies, such as HDL slicing, to reduce the size of the target and make the verification process faster.

6.3.1 Forte Verification Framework

A robust framework has evolved from the Voss verification system [87], which included a simulator back-end and a language front-end. The front-end is a compiler/interpreter for a small, fully-lazy (delayed evaluation based on necessity),

meta-language (ML) with precisely defined semantics. A specification program built in this language is executed to build a simulation sequence that completely verifies the specification. The symbolic simulator is event-driven and can handle both gate-level and switch-level descriptions.

The Forte scripting language is a strongly-typed functional language (FL) that is derived from the ML family, with BDDs built into the language. It provides a flexible interface for interfacing and orchestrating model-checking runs, serves as an macro-language for expressing specifications, and provides the control language for Forte's theorem prover.

Forte also includes a graphical interface using Tcl/Tk for programming and debugging. It provides a node browser, a waveform viewer and a circuit browser. The HDL source code is compiled into a formal circuit model and uses the graphical interface to display circuit structures and waveforms. The STE model checker automatically validates a simple temporal logic formula for arbitrary inputs, and computes an exact characterization of the disagreement if the formula is not unconditionally satisfied.

6.3.2 Verification Methodology

Direct application of the Forte-based STE verification tool is impossible on complete processor designs, and therefore would require a hierarchical approach targeting individual modules in the design. However, many modules in a processor are designed to work properly only under certain environmental constraints. For example, a decoder may require that its inputs be legal instructions, or a pipelined execution unit may require a certain delay between consecutive operations. Capturing these constraints to guide the tool to correctly check the MUV can be a daunting

task.

Several techniques, such as sequential and parallel composition along with case splitting, have been used to verify complex designs using the Forte framework [88]. However, this approach requires a theorem prover to link the low-level proofs to the specification at the chip-level. The HDL slicing methodology that was proposed in Chapter 2 extracts these constraints from an RTL description of the surrounding design. FACTOR achieves this by statically analyzing the RTL to identify the relevant statements and signals with respect to the MUV interface. The generated synthesizable RTL description of the surrounding logic (performed hierarchically, if necessary) implicitly describes the desired environment constraints. Lemma 2 shows that the process trace of the slice is preserved with respect to the MUV interface, which implies that the simulation of the MUV using the sliced design is equivalent to performing the same using the original design.

A previously verified module, with its STE assertions described with respect to the constraints imposed by the surrounding logic, is chosen. The objective is to compare the CPU performance for checking these assertions using both the original and the sliced designs.

6.3.3 Results

The floating-point (FP) adder-subtractor in Intel's Pentium 4 design is chosen as the MUV for evaluating the effectiveness of the slicing methodology. It is one of the many modules embedded within the FP cluster; the Pentium 4 is logically divided into six clusters [89]. The adder performs IEEE-compliant FP addition and subtraction at single, double, and extended precision. It supports four rounding modes: towards 0, towards $-\infty$, towards $+\infty$, and towards the nearest real number

representable in floating-point format.

The Pentium 4 processor validation effort was the first project of its kind at Intel for which formal verification was used on a large scale. However, formally verifying even at the cluster level was beyond the reach of the state-of-the-art tools. Therefore, the *FP_add_sub* module was originally verified with STE assertions written at an intermediate level within the FP cluster. These assertions were split into 32 cases for easier debugging.

The high-level design was implemented using iHDL, an internal Intel-clone of the VHDL language. This was not suitable to be used with FACTOR, which was implemented to handle Verilog designs. Therefore, an internal translator was used to translate the intermediate-level model to Verilog, which introduced some bugs (for assertions in cases 7 and 8) that were caught by checking with the previously described assertions. This model is sufficient for demonstrating the slicing methodology, provided that the same bug-trace is produced with the sliced model. This is because a bug-trace in the symbolic domain describes the complete set of failures for that assertion.

Forte operates on an Intel-internal compiled version of the RTL code. The size of the original compiled model was 3858 KB. FACTOR was applied to obtain the constraint slice on the inputs and outputs of the MUV. The original RTL description of the MUV was combined with its constraint slice and compiled into the Intel-internal format. The sliced design reduced the target to 2965 KB, which is a 23% reduction. The sliced model was used to check all of the previously described assertions. The CPU times (in seconds) for verification of each case using the original and the sliced model are shown in Table 6.3. The experiments were performed using a 1.5 GHz Pentium 4 processor with 1 GB RAM running Linux.

A comparison of the verification run times reveals that the use of the sliced

Table 6.3: CPU Times for STE with/without HDL Slicing

Case #	Original (secs)	Slice (secs)	Speed-up	Case #	Original (secs)	Slice (secs)	Speed-up
01	2624.6	674.87	3.9	17	1069.67	256.2	4.2
02	2176.27	465.11	4.7	18	1431.31	310.86	3.6
03	2031.71	495.04	4.1	19	1886.65	367.93	5.1
04	2125.78	536.67	4.0	20	2404.24	490.95	4.9
05	2346.91	585.96	4.0	21	1685.77	324.51	5.2
06	2521.91	634.09	4.0	22	1408.51	228.47	6.2
07*	3835.92	1317.03	2.9	23	1632.04	281.47	5.8
08*	4585.92	982.79	4.7	24	2021.0	365.33	5.5
09	2406.41	633.27	3.8	25	2418.92	458.89	5.3
10	2173.76	575.23	3.8	26	2699.27	534.07	5.1
11	2054.84	527.37	2.9	27	3107.4	704.64	4.4
12	1864.63	490.22	3.8	28	692.81	209.39	3.3
13	2041.06	453.69	4.5	29	652.34	188.29	3.5
14	1735.96	483.96	3.6	30	922.51	239.17	3.9
15	587.33	181.48	3.2	31	1242.07	284.3	4.4
16	785.84	211.38	3.7	32	1658.46	341.88	4.9

* indicates cases where bugs were present.

design brings down the execution times by a significant amount. The third column gives the speed-ups obtained (rounded off to nearest one decimal place) which are computed as follows.

$$Speedup \ = \ \frac{Speed_{slice}}{Speed_{orig}}, \ where \ Speed \ = \ \frac{\#\ of\ assertions\ checked}{Time}$$

Since $\#\ of\ assertions\ checked$ is the same in original and sliced design, the speed-up is given by:

$$Speedup \ = \ \frac{Time_{orig}}{Time_{slice}}$$

Table 6.3 shows that the use of the sliced design instead of the original design, resulted in speed-ups varying from 3X to 6X. Note that these results are obtained without using any compiler optimizations to eliminate redundant constraints in the RTL slice. Similarly, PIERs were not used since the golden assertions have been already described without using them. The improvements in speed are a result of the reduction in sizes of the BDD nodes needed for simulation due to slicing. The peak BDD sizes needed for each case of assertion checking using both the original and the sliced designs are shown in Figure 6.3.

These results reinforce the fact that the proposed HDL slicing methodology is effective in reducing the analysis time of CAD tools on industrial-scale circuits. This improvement is in addition to the already existing structural COI reduction present in the STE framework. Forte computes all the nodes that were specified, irrespective of whether or not they are within the COI of the nodes to be traced for checking the assertions. HDL slicing could be potentially used to identify the set of nodes to be traced for each assertion, and speedup STE further.

Verifying a design requires showing that the circuit exhibits correct behavior for all possible initial states and input sequences. This process is not only computationally intensive, but requires that any reduction/abstraction techniques do not lose any necessary information. Therefore, many abstraction mechanisms require that a counterexample trace be simulated on the original design to establish that it is a “true” negative. Additionally, they require inference rules to remove the chance of “false” positives. The proposed methodology eliminates the necessity for post-analysis of a verification effort, since it retains all the necessary environmental constraints for a MUV in its original form.

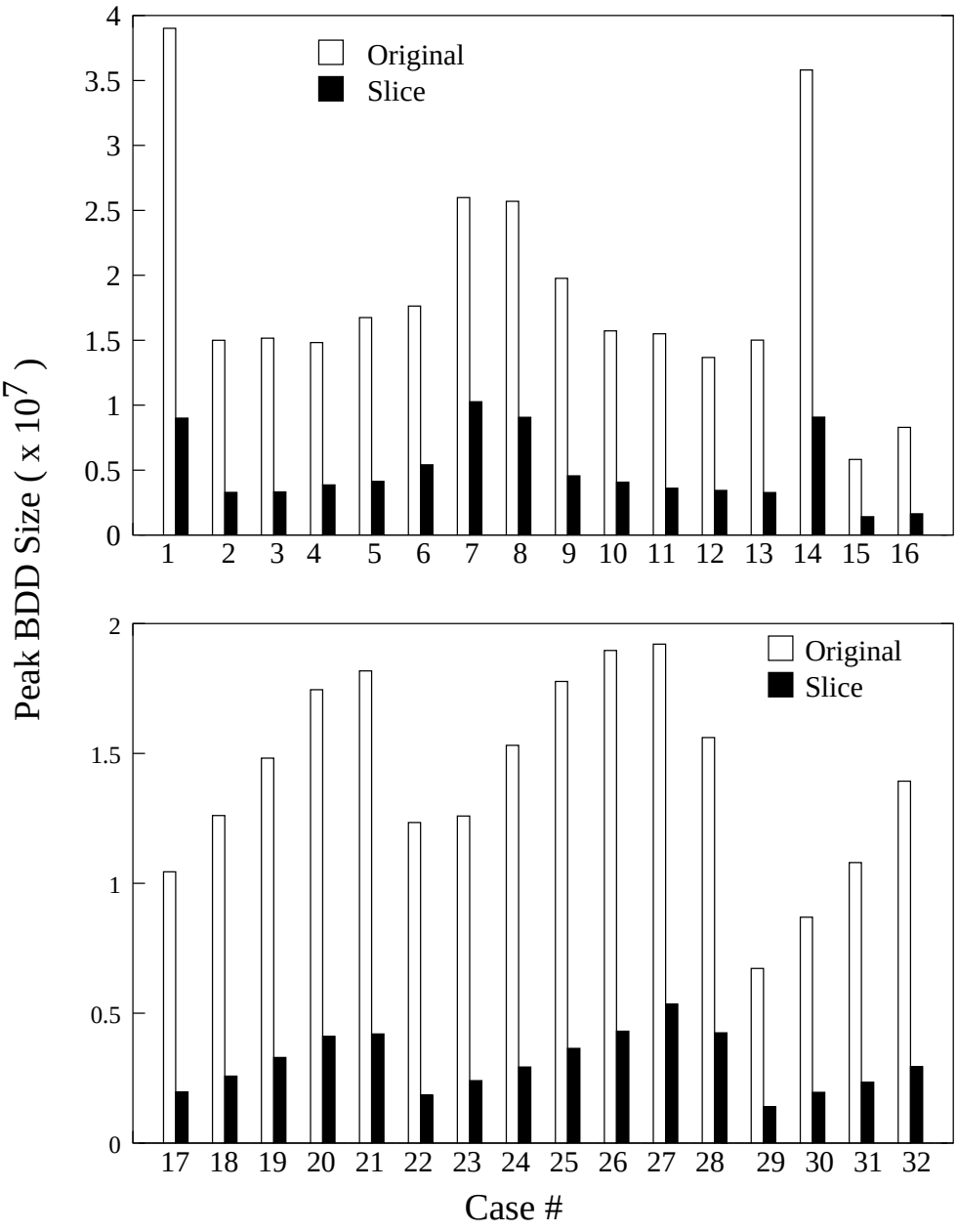

Figure 6.3: Peak BDD Sizes for Each Case in Table 6.3

Chapter 7

Concluding Remarks

Many complex designs are hierarchical, since the use of hierarchy makes the design process easier. The proposed approach utilizes this inherent hierarchy in designs to simplify the complexity of sequential ATPG, and uses it to derive tests on individual modules in the design using existing ATPG tools. The surrounding logic of each MUT has been distilled to produce the relevant slice that defines the ATPG view for the MUT.

The proposed approach formalizes and extends a previous approach [14] that exploits the hierarchy to simplify the generation of slices. This allows the reuse of previously derived slices. The technique exploits the efficiency of netlist graph traversal rather than performing a search through the STG. An elegant theoretical foundation has been proposed to use program slicing for HDLs, and its benefits for test generation were demonstrated. The methodology was successfully implemented as a prototype tool FACTOR, which was effectively used for improving the coverage and time taken to generate test patterns on Verilog designs. The tool reduces the overall test generation time while also providing valuable insight into the testability of a design. The results generated using the tool show its capability

to ease the ATPG complexity, and offers a systematic and sound technique that is scalable for large designs.

An ATPG-based property checking approach has been described for validating HDL designs. The approach fits well into the normal design flow and does not require any significant modifications to the design to be verified. The basic approach was explained in detail in Chapter 4. Sequential ATPG capabilities that have matured over more than three decades have been successfully used to check an important class of temporal properties for bounded semantics. The ATPG framework also allows designers to easily specify properties which are currently checked using simulation. The results show that ATPG-based search techniques are superior to conventional SAT-based search, and is able to deal with much larger designs than is possible with SAT-based approaches. This is because ATPG has knowledge of the structural information in the design under verification, which is missing in a SAT-based approach. A hybrid approach which combines the benefits of the two approaches such as the one proposed by Ganai *et al.* [79] may offer a good solution to this problem. The approach, proposed in this dissertation, is able to prove properties in designs with "real" artifacts such as tri-state buffers, and can be inserted seamlessly into the normal design flow, making it easy to apply to real designs. However, the approach needs to be augmented by a strategy to check nested properties.

The novel verification approach has been successfully integrated with the proposed slicing methodology to produce a powerful technique to speed up design verification by several orders of magnitude. Verification requires only the backward slices of the signals used to specify the property, and FACTOR could be potentially restricted to generate there slices. The slicing approach has also been successfully applied to an industrial scale design in an STE-based verification framework.

The methodology augments already existing target reduction techniques, and helps speed up the verification process.

The capability to incrementally derive the constraint slices and reuse previously derived slices opens a new approach for efficient HDL analysis for other CAD applications. Some of these would be to derive functional test patterns for Systems-on-Chip designs, DFT, testability analysis, regression testing, and ATPG-based sequential equivalence checking.

Bibliography

[1] M. L. Bushnell and V. D. Agarwal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, 2000.

[2] E. B. Eichelberger, E. Lindbloom, J. A. Waicukauski, and T. W. Williams. *Structured Logic Testing*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[3] V. D. Agarwal, C. R. Kime, and K. K. Saluja, "A Tutorial on Built-In-Self-Test, Part 1: Principles," *IEEE Design & Test of Computers*, Vol. 10, No. 1, Mar. 1993, pp. 73–82.

[4] V. D. Agarwal, C. R. Kime, and K. K. Saluja, "A Tutorial on Built-In-Self-Test, Part 2: Principles," *IEEE Design & Test of Computers*, Vol. 10, No. 2, Jun. 1993, pp. 69–77.

[5] P. K. Nag, A. Gattiker, S. Wei, R. D. Blanton, and W. Maly, "Modeling the Economics of Testing: A DFT Perspective," *IEEE Design & Test of Computers*, Vol. 19, No. 1, Jan.–Feb. 2002, pp. 29–41.

[6] P. C. Maxwell and R. C. Aitken, "Test Sets and Reject Rates: All Fault Coverages Are Not Created Equal," *IEEE Design & Test of Computers*, Vol. 10, No. 1, Mar. 1993, pp. 42–51.

[7] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. Piscataway, New Jersey: IEEE Press, 1994, Revised printing.

[8] P. Wohl and J. Waicukauski, "Test Generation for Ultra-Large Circuits using ATPG Constraints and Test-Pattern Templates," *Proc. International Test Conference*, Oct. 1996, pp. 13–20.

[9] J. Lee and J. H. Patel, "Hierarchical Test Generation under Architectural Level Functional Constraints," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 9, Sep. 1996, pp. 1144–1151.

[10] R. S. Ramachandini and D. E. Thomas, "Behavioral Test Generation using Mixed Integer Non-linear programming," *Proc. International Test Conference*, Oct. 1994, pp. 958-967.

[11] P. Vishakantaiah, J. A. Abraham, and M. Abadir, "Automatic Test Knowledge Extraction from VHDL (ATKET)," *Proc. Design Automation Conference*, Jun. 1992, pp. 273-278.

[12] J. D. Calhoun and F. Brglez, "A Framework and Method for Hierarchical Test Generation," *Proc. International Test Conference*, Sep. 1989, pp. 480–490.

[13] D. Bhattacharya and J. P. Hayes, "A Hierarchical Test Generation Methodology for Digital Circuits," *J. Electronic Testing: Theory and Applications*, Vol. 1, No. 2, May 1990, pp. 103-123.

[14] R. S. Tupuri, "Hierarchical Sequential Test Generation for Large Circuits," *Ph.D. Dissertation*, The University of Texas at Austin, May. 1999.

[15] M. Weiser, "Program Slicing," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 4, Jul. 1984, pp. 352–357.

[16] Y. Deng, S. Kothari, and Y. Namara, “Program Slice Browser,” *Proc. International Workshop on Program Comprehension*, May 2001, pp. 50–59.

[17] J. R. Lyle and K. B. Gallagher, “A Program Decomposition Scheme with Applications to Software Modification and Testing,” *Proc. Hawaii International Conference on System Sciences*, Vol. II, Jan. 1989, pp. 479–485.

[18] K. B. Gallagher and J. R. Lyle, “Using Program Slicing in Software Maintenance,” *IEEE Trans. Software Engineering*, Vol. 17, No. 8, Aug. 1991, pp. 751–761.

[19] F. Lanubile and G. Visaggio, “Extracting Reusable Functions by Flow Graph-Based Program Slicing,” *IEEE Trans. Software Engineering*, Vol. 23, No. 4, Apr. 1997, pp. 246–259.

[20] J. Cheng, “Slicing Concurrent Programs - A Graph-Theoretical Approach,” *Lecture Notes in Computer Science, Automated and Algorithmic Debugging*, May 1993, pp. 223–240.

[21] M. Nanda and S. Ramesh, “Slicing Concurrent Programs,” *Proc. International Symposium on Software Testing and Analysis*, 2000, pp. 180–190.

[22] S. Ichinose, M. Iwaihara, and H. Yasuura, “Program Slicing on VHDL Descriptions and Its Evaluation,” *IEICE Trans. Fundamentals of Electronics, Communications and Computer Science*, Vol. E81-A, No. 12, Dec. 1998, pp. 2585–2597.

[23] E. M. Clarke, M. Fujita, P. S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, “Program Slicing of Hardware Description Languages,” *Proc. Conference on Correct Hardware Design and Verification Methods*, Sep. 1999, pp. 298-312.

[24] K. Roy and J. A. Abraham, "High Level Test Generation Using Data Flow Descriptions," *Proc. European Design Automation Conference*, Mar. 1990, pp. 480–484.

[25] C. H. Chen, C. Wu, and D. G. Saab, "Accessibility Analysis on Data Flow Graph: An Approach to Design For Testability," *Proc. International Conference on Computer Design*, Oct. 1991, pp. 463–466.

[26] S. Seshadri and M. S. Hsiao, "An Integrated Approach to Behavioral-Level Design-For-Testability Using Value-Range and Variable Testability Techniques," *Proc. International Test Conference*, Sep. 1999, pp. 858–867.

[27] A. Silburt, "Functional Verification of Silicon Intensive Systems," *Proc. DesignCon98 On-Chip System Design Conference*, Jan. 1998.

[28] S. Kang and S. A. Szygenda, "The Simulation Automation System (SAS): Concepts, Implementation, and Results," *IEEE Trans. VLSI Systems*, Vol. 2, Mar. 1994, pp. 89–99.

[29] C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey," *ACM Trans. Design Automation of Electronic Systems*, Vol. 4, No. 2, Apr. 1999, pp. 123–193.

[30] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Trans. Computers*, Vol. 40, No. 2, Feb. 1991, pp. 205–213.

[31] S.-Y. Huang and and K.-T. Cheng. *Formal Equivalence Checking and Design Debugging*, Frontiers in Electronic Testing, Vol. 12, Kluwer Academic Publishers, Jun. 1998.

[32] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. New York: Academic Press, 1998.

[33] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, " Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation*, Vol. 98, No. 2, , Jun. 1992, pp. 142–170.

[34] A. Aziz, "Formal Methods in VLSI System Design," *Ph.D. Dissertation*, University of California at Berkeley, May 1996.

[35] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, Vol. 6, No. 2, Mar. 1995, pp. 147–189.

[36] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," *Tools and Algorithms for the Analysis and Construction of Systems*, Mar. 1999, pp. 193–207.

[37] D. Du, J. Gu, and P. M. Pardalos. *Satisfiability Problem: Theory and Applications*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Vol. 35, 1997.

[38] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, "Benefits of Bounded Model Checking at an Industrial Setting," *Proc. Computer-Aided Verification*, Jul. 2001, pp. 436–453.

[39] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property Checking via Structural Analysis," *Proc. Computer-Aided Verification*, Jul. 2002, pp. 151–165.

[40] V. Boppana, S. P. Rajan, K. Takayama, and M. Fujita, "Model Checking Based on Sequential ATPG," *Proc. Computer-Aided Verification*, Jul. 99, pp. 418–429.

[41] C.-Y. Huang and K.-T. Cheng, "Using Word-Level ATPG and Modular Arithmetic Constraint-Solving Techniques for Assertion Property Checking," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 3, Mar. 2001, pp. 381–391.

[42] M. Hsiao and J. Jain, "Practical Use of Sequential ATPG for Model Checking: Going the Extra Mile Does Pay Off," *Proc. High-Level Design Validation and Test Workshop*, Nov. 2001, pp. 39–44.

[43] S. Sheng, K. Takayama, and M. S. Hsiao, "Effective Safety Property Checking Using Simulation-Based Sequential ATPG," *Proc. Design Automation Conference*, Jun. 2002, pp. 813–818.

[44] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, Vol. C-35, No. 8, Aug. 1986, pp. 677–691.

[45] J. Yang, C.-J. H. Seger, "Introduction to Generalized Symbolic Trajectory Evaluation," *Proc. International Conference on Computer Design*, Sep. 2001, pp. 360–367.

[46] J. A. Abraham, V. M. Vedula, and D. G. Saab "Verifying Properties Using Sequential ATPG," *Proc. International Test Conference*, Oct. 2002, pp. 194–202.

[47] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts: Addison-Wesley, 1986.

[48] G. A. Venkatesh, “The Semantic Approach to Program Slicing,” *Proc. ACM Conference on Programming Language Design and Implementation*, Jun. 1991, pp. 107–119.

[49] V. M. Vedula, J. A. Abraham, and J. Bhadra, “Program Slicing for Hierarchical Test Generation,” *Proc. VLSI Test Symposium*, Apr. 2002, pp. 237–243.

[50] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. S. Tupuri, “A Hierarchical Test Generation Approach Using Program Slicing Techniques on Hardware Description Languages”, *J. Electronic Testing: Theory and Applications*, Vol. 19, No. 2, Apr. 2003, pp. 149-160.

[51] H. Garavel and M. Sighireanu, “A Graphical Parallel Composition Operator for Process Algebras,” *Proc. Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, Oct. 1999, pp. 185–202.

[52] V. M. Vedula and J. A. Abraham, “A Novel Methodology for Hierarchical Test Generation using Functional Constraint Composition,” *Proc. International High-Level Design Validation and Test Workshop*, Nov. 2000, pp. 9–14.

[53] V. M. Vedula and J. A. Abraham, “FACTOR: A Hierarchical Methodology for Functional Test Generation and Testability Analysis,” *Proc. Design Automation and Test in Europe*, Mar. 2002, pp. 730–734.

[54] A.J. van de Goor. *Testing Semiconductor Memories: Theory and Practice*. Chichester, U.K.: John Wiley & Sons, 1991.

[55] D. Brahme and J. A. Abraham, “Functional Testing of Microprocessors,” *IEEE Trans. on Computers*, Vol. 33, No 6, Jun. 1984, pp. 475–485.

[56] C. Calamvokis, “v2html: Rough Verilog Parser, Ver. 6.0,” http://www.burbleland.com/v2html/rvp.html.

[57] J. K. Huggins and D. V. Campenhout, “Specification and Verification of Pipelining in the ARM2 RISC Microprocessor,” *ACM Trans. Design Automation of Electronic Systems*, Vol. 3, No. 4, Oct. 1998, pp. 563–580.

[58] E. A. Emerson, “Temporal and Modal Logic,” *Handbook of Theoretical Computer Science*, Vol. B, 1990, pp. 995-1072.

[59] L. Lamport, “Proving the Correctness of Multiprocess Programs,” *IEEE Trans. Software Engineering*, Vol. SE-3, No. 2, Mar. 1977, pp. 124–143.

[60] A. Pnueli, “The Temporal Semantics of Concurrent Programs,” *Theoretical Computer Science*, Vol. 13, No. 1, Jan. 1981, pp. 45–60.

[61] E. Clarke, E. A. Emerson, and A. Sistla, “Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications,” *ACM Trans. Programming Languages and Systems*, Vol. 8, No. 2, 1986, pp. 244–263.

[62] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[63] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[64] J. D. Bingham and A. J. Hu, “Semi-Formal Bounded Model Checking,” *Proc. Computer-Aided Verification*, Jul. 2002, pp. 280–294.

[65] W. C. Carter, W. H. Joyner, Jr., and D. Brand, “Symbolic Simulation for Correct Machine Design,” *Proc. Design Automation Conference*, 1979, pp. 280-286.

[66] A. Jain, “Formal Hardware Verification by Symbolic Trajectory Evaluation,” *Ph.D. Dissertation*, Carnegie-Mellon University, July 1997.

[67] K. T. Cheng and A. Krstic, “Current Directions in Automatic Test-Generation,” *IEEE Computer*, Vol. 32, No. 11, Nov. 1999, pp. 58–64.

[68] G. Parthasarathy, C.-Y. Huang, and K.-T. Cheng, “An Analysis of ATPG and SAT Algorithms for Formal Verification,” *Proc. High-Level Design Validation and Test Workshop*, Nov. 2001, pp. 177–182.

[69] E. M. Clarke, O. Grumberg, and H. Hamaguchi, “Another Look at LTL Model Checking,” *J. Formal Methods in System Design*, Vol. 10, No. 1, Feb. 1997, pp. 57–71.

[70] Mentor Graphics Corporation, http://www.mentor.com/dft/scan.html#flex

[71] W. -T. Cheng, ”The BACK Algorithm For Sequential Test Generation,” *Proc. International Conference on Computer Design*, Aug. 1988, pp. 66–69.

[72] Cadence Berkeley Laboratories, http://www-cad.eecs.berkeley.edu/∼kenmcmil/smv/

[73] M. R. Garey and D. S. Johnson. *Computers and Intractability*. New York: W. H. Freeman and Co., 1979.

[74] O. H. Ibarra and S. K. Sahni, “Polynomial Complete Fault Detection Problems,” *IEEE Trans. Computers*, Vol. C-24, No. 3, Mar. 1975, pp. 242-249.

[75] M. Davis and H. Putnam, “A Computing Procedure for Quantification Theory,” *J. ACM*, Vol. 7, No. 3, 1960, pp. 201–215.

[76] R. Zabih and D. A. McAllester, “A Rearrangement Search Strategy for Determining Propositional Satisfiability,” *Proc. National Conference on Artificial Intelligence*, 1988, pp. 155–160.

[77] J. M. Silva and K. A. Sakallah, “GRASP-A New Search Algorithm for Satisfiability,” *Proc. International Conference on Computer-Aided Design*, Nov. 1996, pp. 220–227.

[78] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an Efficient SAT Solver”, *Proc. Design Automation Conference*, Jun. 2001, pp. 530–535.

[79] M. K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik, “Combining Strengths of Circuit-based and CNF-based Algorithms for a High-Performance SAT Solver,” *Proc. Design Automation Conference*, Jun. 2002, pp. 747–750.

[80] D. G. Saab, J. A. Abraham, and V. M. Vedula, “Formal Verification Using Bounded Model Checking: SAT versus Sequential ATPG Engines,” *Proc. International Conference on VLSI Design*, Jan. 2003. To appear.

[81] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, “You Assume, We Guarantee: Methodology and Case Studies,” *Proc. Computer-Aided Verification*, Jul. 1998, pp. 440–451.

[82] O. Grumberg and E. Long, “Compositional Model Checking and Modular Verification,” *Trans. Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pp. 843–871.

[83] R. Jhala and K. L. McMillan, “Microarchitecture Verification by Compositional Model Checking,” *Proc. Computer-Aided Verification*, Jul. 2001, pp. 396–410.

[84] T. Nakata, S. Kowatari, H. Iwashita, and Koichiro Takayama, “Techniques for Effectively Applying Model Checking to Design Projects,” *Fujitsu Scientific and Technical Journal*, Jun. 2000, Vol. 36, No. 1, pp. 9–16.

[85] T. E. Marchok, A. El-Maleh, W. Maly, and J. Rajski, “A Complexity Analysis of Sequential ATPG,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 11, Nov. 1996, pp. 1409–1423.

[86] W. J. Culler. “Implementing Safety Critical Systems: The VIPER Microprocessor,” Kluwer Academic Publishers, 1987.

[87] C.-J. H. Seger, “Voss - A Formal Verification System, User’s Guide,” *Technical Report TR-93-45*, University of British Columbia, 1993.

[88] R. Kaivola and N. Narasimhan, “Formal Verification of the Pentium 4 Floating-Point Multiplier,” *Proc. Design Automation and Test in Europe*, Mar. 2002, pp. 20–27.

[89] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, “The Microarchitecture of the Pentium 4 Processor,” *Intel Technology J.*, 1st Quarter, 2001.

Vita

Vivekananda Murthy Vedula was born in Visakhapatnam, Andhra Pradesh, India on January, the twelfth in the year 1975, the son of Ratna and Rama Murthy Vedula. He received his Bachelor of Technology degree in Chemical Engineering from the Indian Institute of Technology, Madras in 1996. Thereafter, he joined the graduate program in Petroleum Engineering at The University of Texas at Austin, and a year later changed his major to Electrical and Computer Engineering at the same University. He received the Master of Science degree in Electrical and Computer Engineering in 1998, and continued to pursue a Ph.D. degree in the same department. During his graduate studies, he interned at various semiconductor companies such as Advanced Micro Devices in Austin, TX, National Semiconductor in Santa Clara, CA, and Intel Corporation in Austin, TX.

Permanent Address: F-304, Soundaryalahari Apts.,
Lawson's Bay Colony, Visakhapatnam - 530 017
INDIA