Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

FORMAL TECHNIQUES FOR VERIFICATION OF SYNCHRONOUS SEQUENTIAL CIRCUITS

APPROVED BY
DISSERTATION COMMITTEE:

Supervisor: ______________________________

____

____

____

____

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

To my parents, Lata and Vasant Hoskote

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

FORMAL TECHNIQUES FOR VERIFICATION OF SYNCHRONOUS SEQUENTIAL CIRCUITS

by

YATIN VASANT HOSKOTE, 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

December 1995

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Acknowledgements

I wish to thank my research supervisor, Professor Jacob Abraham, for his guidance and encouragement throughout the course of my graduate work. His drive and enthusiasm has been an inspiration. I also wish to thank Professor Don Fussell for his guidance and unfailing support. Thanks also to Dr. Warren Hunt for his interest and time, and for his invaluable advice after my research proposal, which changed the course of my research and, in my opinion, saved this dissertation. I am also grateful to Professors Steve Szygenda and Charles Roth for graciously consenting to serve on my dissertation committee and for their valuable suggestions. The financial support extended by the Semiconductor Research Corporation and the Texas Advanced Technology Development and Transfer Program is also gratefully acknowledged.

These years would have been trying indeed without the company of good friends. Ashok, Naveena, Ghani, Fang, Karthik, Nasser, Hoon, Rathish, Dinos, Prakash, Nandan, Scott, and everyone else at CERC — thanks! Special thanks to Ashok, for his support at home and at school, and to Karthik, whose early encouragement and enthusiasm played a large part in my decision to go for my doctorate. I am also thankful to Linda and Shirley for all their help and assistance, and for making everyone at CERC feel part of a family.

I am indebted to my parents, Lata and Vasant Hoskote, because I owe everything to them, and to my sister, Vanita, for her love and support. I am also grateful to my parents-in-law, Jyotsna and Dinaker Rao, for their encouragement. Last but not least, I wish to thank my wife, Ashwini, for her love and support, and for making it all worthwhile.

YATIN HOSKOTE

The University of Texas at Austin

December 1995

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

FORMAL TECHNIQUES FOR VERIFICATION OF SYNCHRONOUS SEQUENTIAL CIRCUITS

Publication No. ___________________

Yatin Vasant Hoskote, Ph.D.
The University of Texas at Austin, 1995

Supervisor: Jacob A. Abraham

The integration of automated techniques for the formal verification of sequential circuits into the design cycle is imperative to handle today's complex circuits. This thesis presents a unified framework for the multilevel verification of sequential circuits viewed as finite state machines. Experimental results are presented on designs from industry.

Verification of circuit implementations against their specifications (circuit descriptions at a higher level in the design hierarchy) is formulated as checking for **containment** of the specification state machine in the implementation state machine to ensure that all specified behavior is correctly implemented. The concept of **strong containment** is introduced to exploit correspondences between data registers of descriptions at different stages in the design cycle. It is also shown that containment can be checked by considering subsets of outputs, thus reducing the sizes of the Binary Decision Diagrams used to represent the output functions for large circuits. The concept of strong

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

containment is extended to verify circuits after retiming transformations which destroy the simple data register correspondences.

A framework is also presented for the specification of correctness properties as **labeled** state machines to aid the designer in verifying the behavioral circuit descriptions which serve as top-level specifications in the design cycle. Algorithms have been developed and implemented for generating counterexamples and witnesses to help in debugging the design. In addition, the applicability of formal techniques to improve simulation-based verification is shown in the form of an Extracted Control Flow Machine model. This model, extracted automatically, embodies the flow of control in a sequential circuit and is used to evaluate the control flow coverage of the simulation vectors used for design verification.

The power of theorem proving techniques in handling data abstraction is utilized in the form of a library of Boolean templates to handle high level descriptions in a commercial hardware description language, VHDL. The correctness of the templates is proved using the Boyer Moore theorem prover before their inclusion in the library.

The hierarchical verification of a simple microprocessor, the Viper, is attempted in this framework. A behavioral description is verified against user-specified properties, including part of the instruction set, and a gate level implementation is proved to correctly implement the behavioral VHDL description.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Table of Contents

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

List of Tables

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

List of Figures

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Chapter 1

Introduction

The increasing size and complexity of present day circuits makes it imperative that techniques and tools be developed to aid designers in designing such circuits and in checking the correctness of their designs. Formal verification techniques have received heightened interest of late because of the perceived inadequacy of simulation-based verification to detect bugs in present day designs. For a large and complex design, it is impossible to simulate all possible inputs or sequences of inputs. A simulation-based approach then depends on the designer's experience and knowledge of the design to build a set of sequences of test vectors which exercise the important or error-prone behavior of the design in the hope of catching any errors. Obviously, this method cannot guarantee the absence of bugs. To compound the problem, simulation of a low level implementation, such as a gate level description, often takes months of CPU time. Formal verification techniques, on the other hand, work with abstract models and are better able to scale with the complexity of VLSI designs. Unlike simulation, these techniques cover all behaviors of the circuit model.

Formal approaches to hardware verification use formalized, rigorous reasoning to prove some correspondence relation between the design and its specification. This relation may take several forms, such as, *the behavior of the design is consistent with that of the specification,*

or

the design implements all the specified behavior correctly.

The implementation and specification can be at different levels of abstraction. The description at a higher level of abstraction is the specification and the one at a lower level is the implementation, which may itself be the specification for another lower level. In the design cycle, the topmost specification is often some intuitive description of the circuit functionality required by the designer and the final implementation is a layout level circuit description.

1.1 Verification in the Design Cycle

A typical design cycle is shown in Figure 1.1. An abstract top level specification is first written in some informal language to describe the required functionality of the design. This specification is then refined into a concrete circuit description, usually in some hardware description language (HDL). In current design practice, this description is close to the micro-architecture level of abstraction. For the most part, algorithmic descriptions are accepted as input only by high-level synthesis tools, which are currently suitable only for specialized applications like signal processing. The top-level HDL description for most other classes of designs, as shown in Figure 1.1, usually has operations already scheduled into states or demarcated by clock edges. This description is often partially specified and may be described at a high level of abstraction using abstract data types like integers and operations on these data types like addition and subtraction. This description is then refined through a series of well-defined steps to produce designs at successively increasing levels

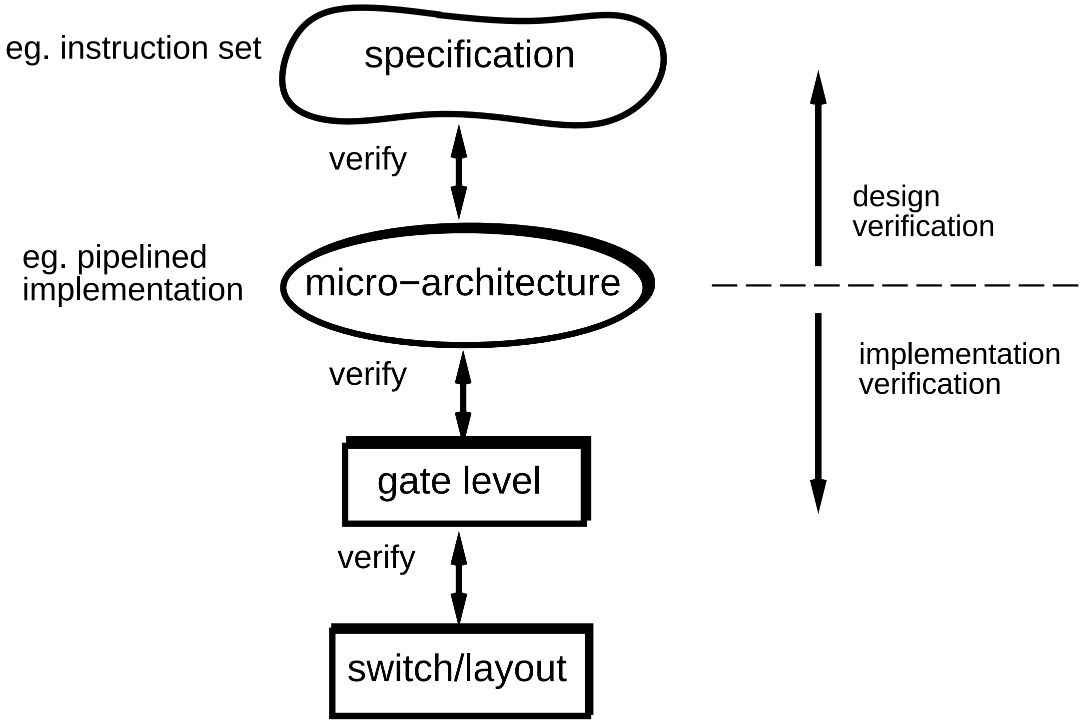


Figure 1.1: Verification in the design cycle.

of detail. These synthesis steps, whether manual or automated, include partitioning, hardware allocation, architectural optimization, logic synthesis and optimization, technology mappin,g and cell layout.

Formal verification techniques are applicable at all stages in this design cycle. Proving that the design actually performs its intended function is usually referred to as **design verification**, whereas proving that the design is implemented correctly at successively lower levels is usually referred to as **implementation verification** (Figure 1.1). In line with this, the framework presented in this thesis takes the view that design verification involves proving that the design possesses a set of properties which embody the intended functionality, whereas implementation verification involves comparing two descriptions of the circuit at different levels of abstraction.

It is important to note that formal methods do not provide a guarantee of correctness. Firstly, verification is always performed with respect to a user-provided specification. The proof of correctness of a design holds only insofar as the specification is correct and complete. Not only does an incorrect specification render the verification useless, but it gives a false sense of security about the correctness of the design. Secondly, formal verification works with abstract models of the design, rather than the design itself. While these techniques benefit tremendously from this ability to abstract away irrelevant details, the models may not always capture the full reality. In such cases, correctness proofs for the models may not translate to correctness of the actual physical design. Besides these worries, of more immediate concern to industry is the fact that most automated formal verification techniques, with a few exceptions, have only been applied to small to medium sized circuits. Furthermore, they do not take as input the same circuit descriptions, usually in some hardware description language, that are used for other design tasks.

In spite of its limitations, formal verification is a step in the right direction to handle the ever-increasing complexity of present-day designs. If formal techniques are used simply as more powerful tools to exercise the design behavior and to detect and correct bugs in the design as it develops, rather than as a guarantee of correctness, they have much to offer. Most formal techniques provide some debugging capability which is far superior to debugging by wading through simulation traces. Furthermore, a designer's understanding of the design is improved due to the discipline and rigor of the verification process. Implementation verification techniques can cover the entire input space and obviate the need for expensive logic simulation. Since formal techniques are relatively fast, they can, wherever applicable, shorten the design cycle and enable more rapid and extensive exploration of the design space, whether at the design stage or at the implementation stage.

1.2 Scope of the Thesis

The primary thrust of this research has been to develop formal verification techniques that can handle realistic circuits at different levels of abstraction and to show the applicability of these techniques to actual industrial practice. As such, we have attempted to test them on examples from industry rather than varied academic benchmark circuits. In our view, the salient features of such a verification tool would be the ability to handle large circuits, ability to accept partial specifications, a high degree of automation, provision of a debugging capability, ability to accept input descriptions in commercial HDLs and absence of a requirement for user expertise in formal logic. A framework which addresses these needs and which includes techniques for design verification as well as implementation verification, both equally crucial for design correctness, is presented here. An important contribution of this work lies in its practical approach of providing powerful techniques for the specification and verification of sequential circuits in a unified, efficient and usable framework. In the following, we first present techniques for implementation verification of large synchronous sequential circuits, followed by techniques for design verification.

In the design cycle, as we have seen, implementations at successively lower levels from the micro-architectural level retain the precise input-output behavior exhibited by their specifications. In such a scenario, formal techniques for comparing the input-output behavior of finite state machines can be applied to prove the **containment** of the specification machine in the implementation machine by finding all pairs of **compatible** states. In order to handle realistic circuits with large data paths, we introduce the concept of **strong containment**, which exploits functional correspondences between registers in the data path of the specification and implementation. This technique is shown to handle large circuits even after retiming transformations which destroy the one-to-one correspondences between data registers in the specification and implementation. In addition, a technique to prove containment by considering subsets of outputs at a time is presented and shown to be indispensable for the verification of large circuits.

It is essential that the micro-architectural descriptions used as the top level specifications in the design cycle also be proved to possess the intended functionality. A new specification framework for expressing the desired functionality as a type of non-deterministic finite state machines, called **labeled** finite state machines, is presented. These specification machines describe state sequences where some states may be labeled, signifying that it is sufficient for any one outgoing transition from a design state to satisfy any outgoing transition from such a specification state for the two states to be output compatible. These labeled states may also have labeled transitions, *i.e.*, transitions without specified inputs. This enables checking for several classes of properties, including types of both safety and liveness properties. Algorithms for generating counter examples, which are execution paths not satisfying the property, and witnesses, which are execution paths satisfying the property, to aid in debugging the design have also been developed and incorporated into this system.

However, proving the completeness of a specification in the realm of formal design verification remains an unsolved problem and tackling large circuits still requires considerable expertise in formal methods. Consequently, the practice of design verification by behavioral simulation persists. In such a scenario, it is important to generate the best test suite possible so as to exercise all important behavior. This thesis attempts to use formal techniques to quantify the quality of design verification vectors by extracting the flow of control of a sequential circuit into a state machine (Extracted Control Flow Machine) and determining the states and transitions in this ECFM not covered by the verification vectors.

We also describe the methodology followed to handle input descriptions in a commercial hardware description language, VHDL [1]. Theorem-proving techniques are used, in an off-line manner, to handle data abstraction and translate high level VHDL descriptions into appropriate form for input to the verifier.

These verification techniques are brought together in a single framework for the hierarchical, multilevel verification of sequential circuits. An example is presented of the verification in this framework of a simple microprocessor, the Viper, from a partial specification of the instruction set down to a gate level design.

1.3 Related Research

There has been a plethora of research in the area of formal verification in the past few years. For a comprehensive survey, the interested reader is referred to tutorials by Yoeli [2] and McFarland [3], as well as an extensive survey by Gupta [4]. This section only attempts to provide a brief description of approaches which are similar in their goals or spheres of application to the work presented in this thesis. Other approaches which employ closely related techniques are discussed throughout the thesis.

Approaches based on describing both the circuit and its specification in a formal logic and verifying the circuit by proving theorems about its correctness using the axioms and rules of inference in the formal system are referred to as theorem-proving based approaches. They are very powerful and flexible in handling the different abstraction mechanisms encountered throughout the design cycle and are especially well suited to the hierarchical nature of circuit descriptions, which is an inherent feature of large, complex designs. Such approaches have been successful in verifying several large circuits, notably the FM8501 microprocessor in Boyer-Moore logic [5], the Tamarack microprocessor in higher-order logic [6], and the mini-Cayuga processor in the functional language Clio [7], among others. However, these techniques require a high degree of expertise in the particular formal logic and experience with using the theorem-proving system. As such, it is not practical to expect circuit designers to embrace them.

On the other hand, state-based approaches like model checking and checking for containment are highly automated and more likely to find favor with designers. Most of these techniques work with Boolean models and perform the verification task by exploring the state space of the circuit being verified. The use of directed acyclic graphs, called Binary Decision Diagrams (BDDs), for representing circuits as Boolean functions was popularized by Akers [8] in the 1970s. At the same time, Fortune, Hopcroft and Schmidt [9] used

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

BDDs to prove equivalence of Boolean functions by ordering the variables in the BDD. In a seminal paper published in 1986 [10], Bryant presented efficient algorithms for manipulating these ordered BDDs (OBDDs) and illustrated their applicability in analyzing properties of Boolean functions. This representation was chiefly used for verifying combinational circuits until much later. A breakthrough in state space representation was presented by Coudert *et al.* in [11, 12] with sets of states represented by their **characteristic functions**, which are predicates on Boolean state variables, or by **vectors of Boolean functions**. At the same time, McMillan [13] presented a technique of representing the transition structure of a state machine by its **transition relation**, which is a Boolean function expressing the relationship between inputs, current state variables and next state variables. These characteristic functions and transition relations could now be represented as BDDs, and the circuit state space could be enumerated implicitly rather than explicitly. These improvements significantly increased the size of circuits that could be handled by state-based approaches.

McMillan applied BDDs to the technique of temporal logic model checking first developed by Emerson *et al.* [14], which takes formulas in a branching-time temporal logic (CTL) as specifications and verifies that they hold true of an implementation which is a finite state system. In McMillan's work, called symbolic model checking, the implementation is described in the input language of the Symbolic Model Verifier (SMV). Using implicit state enumeration techniques, abstractions [15] and efficient techniques for building the transition relations [16], large systems can be verified [17]. SMV has been applied to the verification of industrial protocols using highly abstract models [18], a task for which it is eminently suited. The focus of SMV has thus been design verification, using abstraction as the key to handling large systems. The Murϕ specification and verification system [19] is also targeted at protocol verification and includes both a C++-based verifier, which enumerates states explicitly, and a BDD-based verifier.

Another powerful approach for checking that a system satisfies certain user-specified properties is the language containment approach of Kurshan [20]. In this approach, the property is described as an ω-automaton and the system is described as a non-deterministic L-process, a type of ω-automaton. The language of the system is then checked to be contained in the language of the property to prove that all behaviors of the system are consistent with the property. A hierarchical specification and verification framework, COSPAN [21], has been built around this language containment paradigm, in which a complete system is verified in a series of verification and refinement steps. In this framework, homomorphic reduction transformations are defined by the user, not always an easy task, to simplify both the property and the system being verified, such that proving correctness of the simpler system with respect to the simpler property implies correctness of the original system with respect to the original property.

In symbolic trajectory evaluation [22], properties are specified as assertions about circuit state in the form of pre- and post-conditions and verified using symbolic simulation. To apply this technique for implementation verification, a representation function has to be provided by the user which maps the circuit state to its equivalent specification state. Then each step of execution of the circuit is shown to have the same effect, in terms of the representation function, on the circuit state as one step of the specification has on the specification state. For realistic circuits, the representation function can become very complex. The chief advantages of trajectory evaluation lie in using a symbolic simulator, an enhanced version of COSMOS [23], which operates at the switch level and can take low-level phenomena like precharging into account, and in handling complexity by simulating over only those state variables which appear in the specification. This technique has been used to verify that a switch level implementation of a non-pipelined microprocessor satisfies its instruction set specification [24].

Most other approaches to implementation verification of sequential circuits, where the specification and the implementation are both designs at different levels in the design hierarchy and can be represented as finite state machines (FSMs), are based on state machine comparison. Verifying the equivalence of finite state machines using implicit state enumeration was first developed by Coudert *et al.* [12] and has since been integrated into the design cycle at BULL. They verify the equivalence of two state machines by building the product machine and checking for discrepancies in the outputs of the two machines while performing a forward traversal from specified initial states. Unlike the transition relation method of SMV, they use characteristic functions to represent sets of states and functional vectors to manipulate them and specify a method for switching between the two. These techniques have also been extended to perform CTL model checking [25]. Implicit state enumeration has also been used to prove the equivalence of two gate level circuits by Pixley [26], by checking for a single aligning sequence for all pairs of states in the two circuits and computing all pairs of equivalent states in the process.

Verification of large circuits against their signal-flow graph (SFG) specifications has been presented in [27] and [28]. These methods are more flexible than state machine comparison because they do not require strict input-output equivalence between the specification and the implementation. However, they are closely tied to a specific design methodology using the CATHEDRAL-II synthesis system. In [27], the control unit of the implementation and that of a signal-flow graph specification (after hardware allocation) are shown to be input-output equivalent at all relevant time points. The operations in the signal-flow graph of the specification correspond to register transfers in the implementation. This correspondence must be specified by the user, and the subsequent verification checks that the control circuitry in the implementation initiates the proper events consistent with the operations in the SFG specification. Combinational circuitry in the data path is verified separately.

SFG-tracing [28] relies on partitioning of the signal-flow graph specification into manageable acyclic pieces by the user in order to handle large circuits. Mapping functions, in both time and space, between the interface signals of these partitions and corresponding signals in the implementation must be provided by the user or by the synthesis system and are used to verify pieces of the implementation against subgraphs in the specification. The technique proceeds by symbolic execution of the specification and implementation and comparison of the BDDs for corresponding partitions using the appropriate mapping functions.

For all the automated approaches mentioned above, automation and ease of use comes at a price. The most serious drawback is that these approaches are not as general as theorem provers. They work only for restricted classes of systems, and almost exclusively at the Boolean level. Since they always involve exploration of the state space, which can be huge, they are subject to the state explosion problem. Due to this flattened, state-based approach, they also cannot exploit the hierarchical nature of large circuit descriptions. However, recent attempts at compositional verification [29,30], abstraction [15] and reduction [31] hold a lot of promise for making the design verification problem tractable for large circuits. There has been much progress in implementation verification as well, as shown by the results presented in this thesis and elsewhere [32,33]. There has also been a growing realization that a multi-pronged verification strategy will be most effective, theorem-proving approaches at higher levels of abstraction and state-based approaches at lower levels.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Chapter 2

Implementation Verification

This chapter describes our techniques for verifying the correctness of implementations of large circuits with respect to their specifications, which are circuit descriptions at a higher level of abstraction in the design hierarchy. The lowest level of implementation considered here is the gate level. Both the specifications and implementations, above the gate level, may be partially specified, though an implementation is always more detailed than its specification. Thus, some behavior may be left undefined at the higher level, for example, the response of a processor to certain instructions. At each level, we wish to verify that an implementation correctly implements all the behavior defined in its specification. To achieve this, both the implementation and specification are viewed as deterministic, partially specified finite state machines, and their externally observable behavior is compared on every clock cycle. No distinction is made between sequential circuits and finite state machines. In this approach, the implementation has to retain precise input-output behavior with respect to the specification. This excludes implementations obtained through behavioral transformations or implementations that employ a different time scale from the specification. For example, verification of a pipelined processor against an unpipelined specification is not directly possible using these techniques. However, given current design practice, FSM verification techniques are more than adequate for a large class of designs through several stages in the design process. Their chief drawback is that they have difficulty handling large circuits. Though small to medium-sized circuits (up to 70 flip-flops) have been verified fairly easily using existing techniques, commercial circuits have a large number of memory elements within their data paths, which gives rise to a huge state space and causes state machine comparison techniques to break down. The techniques presented here address this problem.

2.1 Verification as Containment

Most approaches to FSM verification check input-output behavior while performing forward traversal from specified initial states of the product machine of the two FSMs to be compared [12,34]. They thus check the equivalence of the initial states and all corresponding states reachable from the initial states to verify equivalence of the FSMs. This view is called transformational equivalence. However, such unique initial states are rarely specified for real circuits, especially for circuits with data registers, due to the area cost of making all data flip-flops resettable. The preferred option is to specify some control sequences to load the uninitialized registers with known values prior to executing any operations. In such case, **all** states must be considered reachable. Even if the initial state is given for the specification, its encoding in the implementation may not be available or may be obscured after aggressive optimization. Quite often, during optimizations like retiming, states may be introduced in the implementation such that the behavior in those states does not agree with the specification. The only requirement on such states, usually, is that the implementation resets properly when starting from these states. Furthermore, the implementation is more detailed than the specification and is thus not equivalent to it. In light of all these considerations, a better formulation of the verification problem is to check that every state in the specification has a compatible state in the implementation, or in other words, for **containment** [35] of the specification in the implementation, rather than equivalence of the specification and the implementation. This verifies that the implementation possesses all the functionality defined in the specification. If an initial state is partially defined for the specification, for example in the control circuitry, it represents a set of initial states for all values of the uninitialized flip-flops. Not all these states are equivalent to each other, but it may suffice to check that each state in this set has a compatible state in the implementation for the purpose of verifying the implementation. This is a weaker check than containment.

In the language containment approach [31], the language of the implementation is proved to be contained in the language of the specification, which is a property or task encoded as a non-deterministic automaton. Containment is thus proved in the other direction. However, we believe checking containment of the implementation in the specification is not suitable to our problem because it checks only that all behavior of the implementation is consistent with that of the specification, not that the implementation can perform all the behavior defined in the specification. Furthermore, we deal with machines, not automata, and do not have any notion of starting states or final states. It is important to realize that if we viewed the specification and implementation as non-deterministic automata and were given starting states for both, checking compatibility of the starting states would be equivalent to checking containment of the language of the implementation automaton in that of the specification automaton, which is basically the language containment paradigm of Kurshan [31]. Our procedure goes one step further and ensures that for every possible starting state of the specification automaton, there exists a corresponding starting state in the implementation automaton, such that, for each such pair of starting states, the language of the implementation automaton is contained in the language of the specification automaton.

2.2 Containment of Sequential Circuits

As noted previously, we wish to divorce the notion of equivalence or containment from the existence of initial states. Furthermore, in the design hierarchy, it is more pertinent to verify whether the implementation can mimic the behavior of the specification, rather than vice versa. We now present some standard definitions from the classical theory of finite state machines [35].

Definition 1 *States s_i and s_j of a sequential circuit are said to be* k-**equivalent** *if and only if every possible input sequence of length k causes the circuit to produce the same output sequence when the initial state is s_i and when the initial state is s_j.*

Two states are equivalent if and only if they are k-equivalent for all values of k.

Definition 2 *States s_i and s_j of a sequential circuit are said to be* **equivalent**, *$s_i \Leftrightarrow s_j$, if and only if every possible input sequence causes the circuit to produce the same output sequence when the initial state is s_i as it does when the initial state is s_j.*

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Definition 3 *Sequential circuit* C_b **contains** *sequential circuit* C_a, $C_b \supset C_a$, *if and only if for every state of* C_a *there exists an equivalent state in* C_b.

Definition 4 *Sequential circuit* C_b *is* **equivalent** *to sequential circuit* C_a, $C_b \Leftrightarrow C_a$, *if and only if* C_a *contains* C_b *and* C_b *contains* C_a.

In the application of these concepts to the verification of circuits in the design hierarchy, the circuit specification, and possibly the implementation, may be partially defined. Note that a gate level implementation, by its very nature, is always fully defined. A partially specified machine is a machine which has undefined outputs or undefined state transitions for certain inputs in certain states. To accommodate this, we determine compatibility of states (similar to **state covers** in [36]) in a specification with states in an implementation rather than strict equivalence.

Definition 5 *An input sequence that can be applied to a partially specified circuit* C *in state* s *such that the destination states for all induced transitions are specified is said to be* **applicable** *to circuit* C *in state* s.

Definition 6 *State* s_a *in specification circuit* C_a *is* **compatible** *with state* s_b *in implementation circuit* C_b *if and only if every input sequence that is applicable to* C_a *in state* s_a *is also applicable to* C_b *in state* s_b *and causes the two circuits to produce the same output sequence, with respect to the specified outputs of* C_a, *when applied to* C_a *in initial state* s_a *and to* C_b *in initial state* s_b.

Definition 7 *Specification circuit* C_a *is contained in implementation circuit* C_b, $C_a \subset C_b$, *if and only if every state in* C_a *has at least one compatible state in* C_b. *The circuit* C_b *is then said to correctly implement specification* C_a.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

This notion of implementation implies that there may be some states in circuit C_b which are not compatible with any state in specification C_a. This may well occur often in practice when some particular encoding schemes such as one-hot encoding are used. Even otherwise, optimization of the circuit for performance, especially retiming, may lead to the implementation having more states. In such cases, it is likely that the only requirement on such incompatible states is that they satisfy an initialization sequence to some "good" state. This behavior can be verified in the framework presented here, as will be explained shortly. However, note the following observation.

Observation *If all possible input sequences are applicable to every state in the specification,* i.e.*, the next state is always defined, incompatible states in the implementation are states that are not reachable from any implementation state compatible with some specification state.*

Proof: To see that this is true, note that any state s_a in the specification having a compatible state s_b in the implementation implies that all states reachable from s_a have compatible states in the implementation. Now assume that state s_b in the implementation has a transition on input i to a state s_c that has no compatible state in the specification. However, since all input sequences are applicable to every state in the specification, there exists a state in the specification which is reached from state s_a on input i. This state must necessarily be compatible with its corresponding state s_c in the implementation by the definition of state compatibility. This contradicts our assumption that s_c is not compatible with any specification state. Hence, all states in the implementation reachable from some compatible state are also compatible with some specification state. Consequently, only states unreachable from compatible states may be incompatible. □

On the other hand, if not all specification states have compatible states in the implementation, *i.e.*, if the containment check fails, but initial states for the design are partially or fully specified, it may suffice to check the weaker condition that each initial state in the specification has a compatible state in the implementation.

The concepts of **state covers** and **machine covers** were also used for verifying sequential circuits by Hwang and Newton [37]. However, they were able to verify only small circuits with their explicit state enumeration procedure, which checked only that the initial state of the specification is covered by at least one of a set of candidate initial states in the implementation.

The pairs of compatible states can easily be determined using the algorithm in Figure 2.1 [35]. We first find the set of all pairs of 1-compatible states C_1 in the specification and the implementation, then the set of all pairs of 2-compatible states C_2 as those 1-compatible states all of whose corresponding successor states are also 1-compatible, then C_3 as the set of pairs of 2-compatible states all of whose corresponding successor states are also 2-compatible, and so on till we reach a fixed point $C_{k+1} \equiv C_k$. The set of all pairs of compatible states is then given by $\mathcal{C} \equiv C_k$. By corresponding successor states, we mean next states for the same input. The fixed point is always reached because the state space is finite and the sequence $C_1, C_2, ..,$ is monotonically decreasing. Central to the proof of correctness of this algorithm is the fact that if all k-compatible states have corresponding successor states which are also all k-compatible, or in other words, if all k-compatible states are also $k+1$-compatible, then they are compatible [35].

Consider a Mealy machine defined by the 5-tuple $(I, O, S, \Delta, \Lambda)$ where $I = \{0,1\}^m$ is the input space, $O = \{0,1\}^l$ is the output space, $S = \{0,1\}^n$ is the state space, $\Delta : I \times S \rightarrow S$ is the next-state functional vector, $\Delta = [\delta_1 ... \delta_n]$, and $\Lambda : I \times S \rightarrow O$ is the output functional vector, $\Lambda = [\lambda_1 ... \lambda_l]$. Let $\vec{q}$ represent the vector of present-state variables, $\vec{Q}$ the vector of next-state variables, $\vec{x}$ the input vector and $\vec{y}$ the output vector. Thus, we have $Q_i = \delta_i(\vec{x}, \vec{q}),\ i \in \{1..n\}$ and $y_j = \lambda_j(\vec{x}, \vec{q}),\ j \in \{1..l\}$.

The following concepts and operators are in wide use in the area of formal verification using implicit state enumeration procedures [12,13].

Definition 8 *A* **transition relation** *is a relation T on the inputs I and the set of states S, such that, $T(\vec{x}, s_i, s_j) = 1$ if for input $\vec{x}$ in current state s_i, the next state is s_j.*

Given an FSM with next state functional vector $\Delta(\vec{x}, \vec{q})$, the transition relation of the machine can be written as:

$$T(\vec{x}, \vec{q}, \vec{Q}) = \prod_{i=1}^{l} Q_i \odot \delta_i(\vec{x}, \vec{q}) \tag{2.1}$$

where $\odot$ signifies the XNOR operation.

A set of states in an FSM is represented by its characteristic function, which is a Boolean predicate on the state variables.

Definition 9 *The* **characteristic function** *of a set of states $s \subseteq S$ in an FSM is a Boolean function C on the state variables $\vec{q}$ such that $C(\vec{q})$ is true* *whenever the state represented by that valuation of the state variables belongs to s.*

Manipulation of sets of states and traversing the state space of a state machine implicitly is accomplished through three operators, the Image, PreImage and InverseImage operators.

Definition 10 *Given the characteristic function $C(\vec{q})$ of a set of states s in an FSM with transition relation T,*

$$\begin{aligned} img(T, C) &= \{v \mid \exists x[u \in C \wedge T(x, u, v)]\} \\ pre(T, C) &= \{u \mid \exists x[v \in C \wedge T(x, u, v)]\} \\ inv(T, C) &= \{u \mid \forall x[T(x, u, v) \Rightarrow v \in C]\} \end{aligned} \tag{2.2}$$

Basically, the image of a set of states C is the set of states which can be **reached from** a state in C by one transition, the pre-image is the set of states **from which** a transition **can** go to a state in C, and the inverse image is the set of states **from which** all transitions **must** go to states in C.

The efficiency of the fixed-point algorithm to compute all pairs of equivalent or compatible states increased significantly by using implicit state enumeration procedures and BDDs [26,38]. The characteristic functions of sets of states can be represented as BDDs, and sets of states can be manipulated rather than a single state at a time. In [39], the fixed-point algorithm is used to compute pairs of equivalent states in an FSM and its implementation and the encoding of symbolic FSM states in the implementation is determined. The implicit enumeration procedure uses the transition relation. In [40], all pairs of

1. $i = 1$;
2. $C_1 = \{$pairs of states that are 1-compatible$\}$;
3. repeat
4. $\quad C_{i+1} = C_i$ - $\{$pairs of states in C_i with successors not in $C_i\}$;
5. $\quad i = i + 1$;
6. until $(C_i = C_{i-1})$;

Figure 2.1: Algorithm for computing pairs of compatible states.

inequivalent states in two FSMs are determined by using next state functions instead of transition relations.

Let C_a be the partially specified machine representing the specification and C_b be the (possibly partially specified) implementation. The inputs and outputs of the specification and implementation machines have a one-to-one correspondence. The number of state variables may differ, though we assume $n_a = n_b = n$ in the following derivations for the sake of simplicity and without loss of generality.

Symbolic states are encoded using Boolean variables independent of the encoding in the other description. We also introduce a *dummy* state which acts as the sink for all transitions where the next state is undefined. Encoding N states could introduce $(2^{\lceil log_2 N \rceil} - N)$ extraneous states. Let $INVALID(\vec{q}_a)$ be the characteristic function of the set of extraneous states introduced as a result of encoding the symbolic states in the specification, and let the vector Boolean functions $\eta_a : I \times S \rightarrow O$ and $\eta_b : I \times S \rightarrow O$ represent the transitions with unspecified outputs in circuits C_a and C_b respectively.

To find the set of pairs of compatible states in specification C_a and implementation C_b, we first find the set C_1 of pairs of 1-compatible states as those states which produce the same outputs for all inputs applicable to C_a when the outputs of C_a are specified (step 2). The set of state pairs whose corresponding next-states belong to the set C_k of pairs of k-compatible states can be seen to be the **inverse image** (Equation 2.2) of the set C_k. At each step in the fixed-point computation, the set of pairs of $k+1$-compatible states is computed by taking the inverse image of the set C_k, restricted to the set C_k (step 4). Finally, the set $\mathcal{C}$ of all pairs of compatible states is the set C_k when the fixed point is reached. The characteristic functions of these sets are the following:

$$\begin{aligned}
C_1(\vec{q}_a, \vec{q}_b) &= \forall \vec{x} : \bigwedge_{i=1}^{l} [\ ((\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b))\ \wedge\ \neg\eta_{b_i}(\vec{x}, \vec{q}_b))\ \vee\ \eta_{a_i}(\vec{x}, \vec{q}_a)\ ] \\
C_{k+1}(\vec{q}_a, \vec{q}_b) &= C_k(\vec{q}_a, \vec{q}_b) \bigwedge \forall \vec{x} : C_k(\vec{Q}_a, \vec{Q}_b) \qquad (2.3) \\
\mathcal{C}(\vec{q}_a, \vec{q}_b) &= C_k(\vec{q}_a, \vec{q}_b) \qquad if\ C_{k+1} = C_k
\end{aligned}$$

where $\odot$ is the Boolean equivalence (XNOR) operator.

To prove containment of specification circuit C_a in implementation circuit C_b, there must exist at least one compatible state in C_b for each valid state in C_a,

$$\forall \vec{q}_a, \exists \vec{q}_b :\ \mathcal{C}(\vec{q}_a, \vec{q}_b)\ \vee\ INVALID(\vec{q}_a)$$

Determination of the set $INVALID$ from a high level specification is fairly straightforward since we know the state encoding; it includes the set of states which do not exist in the specification but are created as a result of translating the symbolic specification into a Boolean specification. It must be noted that $INVALID$ is not the set of unreachable states, since we do not have knowledge of any initial state. A state belongs to $INVALID$ if there does not exist a single applicable input vector for that state.

Enumerating the states in the set $\mathcal{C}$ gives the encoding in the implementation of every state in the specification. Incompatible states in the implementation can also be identified. The characteristic function of the set κ of states in C_b not compatible with any state in C_a is given by

$$\kappa(\vec{q}_b) \;=\; \neg\; \exists \vec{q}_a \;:\; \mathcal{C}(\vec{q}_a, \vec{q}_b)$$

To signify absence of incompatible behavior in circuit C_b, κ must be the empty set. However, it may frequently be the case that the designer has introduced other states to facilitate a more efficient implementation and requires that they satisfy only the initialization property.

Computing all pairs of compatible states in the specification and implementation also enables us to verify that a given reset sequence actually does reset the implementation. Knowledge of the reset sequence is not an unreasonable assumption, since the reset sequence for the specification has been specified by the designer. To verify resettability of an implementation with a reset sequence $\vec{x}_1 ... \vec{x}_k$, a sequence of dummy states $D_i (i = 1..k)$, with unspecified outputs, is added to the specification with transitions $D_i \stackrel{\vec{x}_i}{\rightarrow} D_{i+1}$ and $D_k \stackrel{\vec{x}_k}{\rightarrow} \vec{q}_{reset}$. This can result in increasing the number of state variables in the specification by at most one. The implementation is resettable with the given input sequence if, after computing the set of pairs of compatible states, D_1 is compatible with all states in the implementation, *i.e.*,

$$\forall \vec{q}_b \;:\; \mathcal{C}(D_1, \vec{q}_b)$$

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits," Ph.D. Dissertation, The University of Texas at Austin, December 1995.

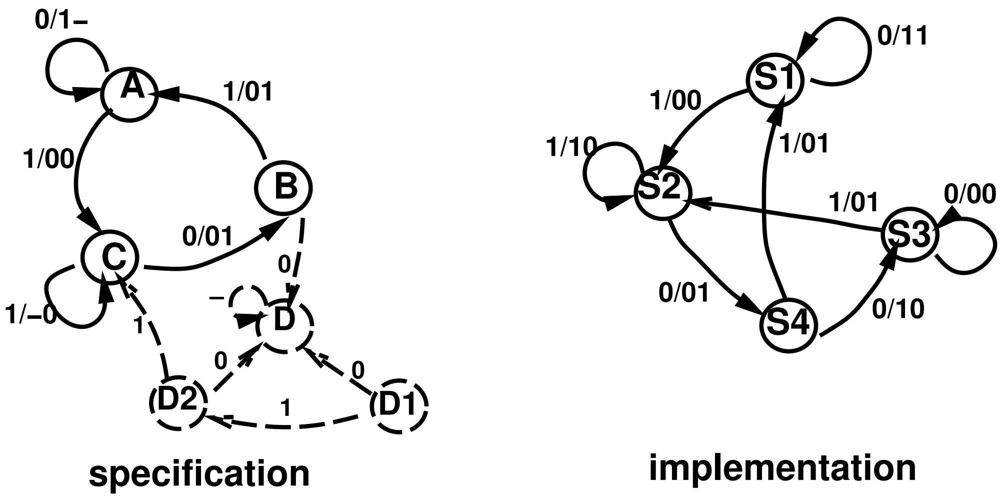


Figure 2.2: Containment and checking of reset circuitry.

To clarify the ideas of containment verification and checking of reset circuitry presented above, consider the following simple example (Figure 2.2) of a design with a single input and two outputs. The specification has three states A, B and C. The reset state is state C with reset sequence 11. A dummy state D is added to the specification to act as a sink for undefined transitions (input 0 for state B). The implementation has four states and is fully specified. This could represent the most frequent case of the implementation being a gate level description. Performing verification gives us the following pairs of compatible states $(\langle A, S1\rangle, \langle B, S4\rangle, \langle C, S2\rangle)$, which proves that the specification is indeed contained in the implementation. However, state $S3$ in the implementation is not compatible with any state in the specification. To verify that the implementation can be reset from $S3$ with reset sequence 11, two states $D1$ and $D2$ with associated transitions are added to the specification. All additions to the specification are shown with dotted lines in Figure 2.2. Recomputing the pairs of compatible states confirms that all implementation states are compatible

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

with $D1$ and that the implementation is reset by input sequence 11.

Computation of C_{k+1} or E_{k+1} in the above algorithm, to find the set of state pairs whose successor states are k-compatible by finding the inverse image of C_k, has complexity exponential in the number of state variables in the two circuits, because it involves the expensive composition operation. Attempts to reduce this complexity have been made, for example, by recognizing that the the final result of the composition operation is restricted to the set of k-equivalent states and using this constraint to simplify the composed functions [41], by attempting to simplify both the constraints and the composed functions together [42] or by computing E_{k+1} for subsets of the k-equivalent states at a time [43]. In contrast to these low level approaches which seek to improve the computation operation, the techniques presented in the next section attack this problem by using information about relationships between partial state encodings in the two machines. Consequently, these techniques can be used in tandem with all these low-level techniques to get even better performance.

2.3 Strong Containment

In this section, we present an extension of containment, called **strong containment**, which is a conservative approach to containment and enables us to verify circuits with large data paths. Most approaches to implementation verification of large designs separate the control and data path and verify the control units and the data path units separately [44]. Not only does this require exact correspondence between the functional units in the implementation and the specification, but it also ignores verification of the composition of the individually verified units. The approach presented here is based on the far less

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

restrictive assumption that registers in the data paths of two descriptions of the same design exhibit a one-one correspondence, even though the encoding of the control states may differ. Information about register correspondence is used to abstract those registers out of the computation. This technique enables verification of large circuits against specifications without any structural information, as in scheduled control/data-flow graphs prior to hardware allocation, or two implementations obtained with different allocation schemes. If the assumption about correspondence is wrong, the verification fails. The correspondence information has either to be supplied by the designer or by the synthesis tool. The number of states in the control path is relatively small for a large class of circuits, and utilizing data register correspondence significantly extends the domain of applicability of automatic state machine comparison techniques.

Thus, given some knowledge of the state encodings in the specification and implementation, a stronger relationship than containment can be formulated between the two circuits, resulting in significantly lower complexity of the verification. We define the notion of strong compatibility on the set of compatible states.

Definition 11 *State* $\vec{q}_a = (q_{a_1}...q_{a_n})$ *of specification circuit* C_a *is* **strongly** k**-compatible** *with state* $\vec{q}_b = (q_{b_1}...q_{b_n})$ *of implementation circuit* C_b *if they are* k*-compatible,* d $(d \in \{1..n\})$ *bits of* $\vec{q}_a$ *have the same values as* d *bits of* $\vec{q}_b$*, and all states in* C_a *reachable from* $\vec{q}_a$ *by an input sequence of length* $\leq k$ *also have identical values for the same* d *bits as corresponding states in* C_b *reachable from* $\vec{q}_b$*.*

As we have seen, two states being k-compatible implies that all their corresponding successor states are $k-1$-compatible. It follows that all corresponding successor states of strongly k-compatible states in the specification and implementation are strongly $k-1$-compatible. Strongly compatible states are states which are strongly k-compatible for all values of k.

Definition 12 *State* $\vec{q}_a = (q_{a_1}...q_{a_n})$ *of specification circuit* C_a *is* **strongly compatible** *with state* $\vec{q}_b = (q_{b_1}...q_{b_n})$ *of implementation circuit* C_b *if they are compatible,* d $(d \in \{1..n\})$ *bits of* $\vec{q}_a$ *have the same values as* d *bits of* $\vec{q}_b$*, and all states reachable from* $\vec{q}_a$ *also have identical values for the same* d *bits as corresponding states reachable from* $\vec{q}_b$*.*

A direct implication of this definition is that corresponding successor states of strongly compatible states are also strongly compatible. Consequently, a **one-to-one functional correspondence** exists between the d state variables in the specification and implementation.

Definition 13 *Specification circuit* C_a *is* **strongly contained** *in implementation circuit* C_b *if, for every state* $\vec{q}_a$ *in* C_a*, there exists at least one strongly compatible state* $\vec{q}_b$ *in* C_b *such that the* d *bits which have the same values are the same bits for all states.*

Two circuits are strongly equivalent if each is strongly contained in the other. In the simplest case, two circuits with the same state encoding $(d = n)$ are strongly equivalent if states with the same encoding are indeed equivalent.

Observation *Sequential circuit* C_a *is contained in sequential circuit* C_b *if*

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Given: The d state variables with identical encoding.

1. $i = 1$;
2. $SC_1 = \{$pairs of states that are strongly 1-compatible$\}$;
3. repeat
4. $\quad SC_{i+1} = SC_i$ - $\{$pairs of states in SC_i with successors not in $SC_i\}$;
5. $\quad i = i + 1$;
6. until $(SC_i = SC_{i-1})$;

Figure 2.3: Algorithm for computing pairs of strongly compatible states.

C_a is strongly contained in C_b. In other words, strong containment implies containment, but not vice versa.

That strong containment implies containment is obvious since strong compatibility of states implies compatibility by definition. However, if there is some specification state for which there exists no strongly compatible state in the implementation for any value of $d > 0$ and strong containment is not satisfied, there may still exist compatible states in the implementation to satisfy the containment relation.

Given the d state variables, the algorithm to find pairs of compatible states is now modified (Figure 2.3) to determine the set of pairs of strongly compatible states $\mathcal{SC}$. It first computes the set of pairs of strongly 1-compatible states SC_1 as those states, $\vec{q}_a$ and $\vec{q}_b$, which have the same outputs for all applicable inputs, have identical values for d variables, and all of whose corresponding successor states have identical values for the same d variables. The pairs of strongly k+1-compatible states, SC_{k+1}, are those pairs of strongly k-compatible states all of whose corresponding successor states are also strongly k-compatible for the same d variables. The set $\mathcal{SC}$ is the fixed point $SC_{k+1} = SC_k$. For the sake of simplicity, we will assume both machines are completely specified. We will also assume, without loss of generality, that the d variables with identical encoding are consecutive variables in $\vec{q}$. The vectors $\vec{q}_a$ and $\vec{q}_b$ can now each be written as the concatenation of two vectors, $\vec{q}_a \equiv \vec{q}_a^{\,s} \cdot \vec{q}_a^{\,d}$ and $\vec{q}_b \equiv \vec{q}_b^{\,s} \cdot \vec{q}_b^{\,d}$ $(n = s+d)$[1]. The characteristic functions of these sets of strongly k-compatible states are given below,

$$
\begin{aligned}
SC_1(\vec{q}_a^{\,s} \cdot \vec{q}_a^{\,d}, \vec{q}_b^{\,s}) &= \forall \vec{x} : \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b^{\,s} \cdot \vec{q}_a^{\,d})) \bigwedge \\
&\quad\ \forall \vec{x} : \bigwedge_{j=s+1}^{n} (\delta_{a_j}(\vec{x}, \vec{q}_a) \odot \delta_{b_j}(\vec{x}, \vec{q}_b^{\,s} \cdot \vec{q}_a^{\,d})) \\
SC_{k+1}(\vec{q}_a, \vec{q}_b^{\,s}) &= SC_k(\vec{q}_a, \vec{q}_b^{\,s}) \bigwedge \forall \vec{x} : SC_k(\vec{Q}_a^{\,s} \cdot \vec{Q}_a^{\,d}, \vec{Q}_b^{\,s}) \qquad (2.4) \\
\mathcal{SC}(\vec{q}_a, \vec{q}_b^{\,s}) &= SC_k(\vec{q}_a, \vec{q}_b^{\,s}) \qquad\qquad if\ SC_{k+1} = SC_k
\end{aligned}
$$

The second term in the characteristic function of SC_1 ensures that the d state variables for all corresponding successor states have the same values. At each step in the fixed-point computation, variables $\vec{q}_b^{\,d}$ are replaced with $\vec{q}_a^{\,d}$. This can be done even while computing the inverse image of SC_k (we have $Q_{b_j} = \delta_{b_j}(\vec{x}, \vec{q}_b^{\,s} \cdot \vec{q}_a^{\,d})\ j \in \{1..n\}$), because we want only those states from the inverse image which are themselves strongly k-compatible and have identical values for the d variables, *i.e.*, the inverse image of SC_k is restricted to the set SC_k itself. The characteristic function of set SC_k now depends on $(2s + d)$ variables $(q_{a_1}, .., q_{a_s}, q_{a_{s+1}}, .., q_{a_n}, q_{b_1}, .., q_{b_s})$, rather than $2n$ variables $(q_{a_1}, .., q_{a_n}, q_{b_1}, .., q_{b_n})$, thus reducing the complexity of the inverse image computation.

---

[1] we use the notation $\vec{y}^{\,n}$ to denote a vector $\vec{y}$ of length n

To prove the correctness of the above algorithm, we need the following lemma.

Lemma 1 *If all strongly k-compatible states $\vec{q}_a$ of specification C_a and $\vec{q}_b$ of implementation C_b are also strongly $k+1$-compatible, then they are strongly compatible.*

Proof: We first prove that $\vec{q}_a$ and $\vec{q}_b$ are strongly $k+2$-compatible by contradiction. Assume they are not strongly $k+2$-compatible. However, they are strongly 1-compatible with the same values for d variables. This implies that there must be some successor states which are not strongly $k+1$-compatible. But $\vec{q}_a$ and $\vec{q}_b$ are strongly $k+1$-compatible and consequently all their successor states must be strongly k-compatible. By the statement of the lemma, all the successor states are then also strongly $k+1$-compatible, which contradicts our previous statement. Hence, our assumption is wrong and $\vec{q}_a$ and $\vec{q}_b$ are strongly $k+2$-compatible. We can apply the same argument for $k+3$ and so on. Hence, $\vec{q}_a$ and $\vec{q}_b$ are strongly k-compatible for all k and hence strongly compatible. □

Theorem 1 *The algorithm presented in Figure 2.3 correctly computes the set of all pairs of strongly compatible states in the specification and the implementation.*

Proof: To prove that the set of state pairs given by the algorithm are indeed strongly compatible, and consequently compatible, we must prove that

1) At each step in the computation, the set SC_k is exactly the set of all pairs of strongly k-compatible states, and

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

2) If SC_k is the set of all pairs of states which are both strongly k-compatible and strongly $k+1$-compatible, then SC_k is exactly the set of all pairs of strongly compatible states.

We will prove statement 1 by induction on k. It clearly holds for $k = 1$. Assuming it holds for some k, the set of strongly $k + 1$-compatible states is computed as those strongly k-compatible states whose successor states are also strongly k-compatible. These states are indeed strongly $k + 1$-compatible because they are at least strongly 1-compatible and all their successor states are strongly k-compatible. Also, there are no strongly $k + 1$-compatible states missing from this set, because all strongly $k + 1$-compatible states are strongly k-compatible too, and the only reason any states are incorrectly missing from SC_{k+1} would be that they are absent in SC_k too, which contradicts our induction hypothesis. To prove statement 2, we note that the algorithm always terminates because the sequence $SC_1, SC_2, ..$ is a monotonically decreasing sequence on a finite state space $(\{0,1\}^s \times \{0,1\}^s \times \{0,1\}^d)$. When it does terminate $(SC_k = SC_{k+1})$, the set SC_k is exactly the set of all pairs of strongly k-compatible states, and they are also strongly $k + 1$-compatible. Statement 2 thus follows from lemma 1, and SC_k is exactly the set of all pairs of strongly compatible states. $\square$

Now that we have the set of all pairs of strongly compatible states in C_a and C_b, specification circuit C_a is strongly contained, and consequently contained, in implementation circuit C_b if

$$\forall \vec{q}_a, \exists \vec{q}_b \;:\; \mathcal{SC}(\vec{q}_a, \vec{q}_b^{\,s}) \;\vee\; INVALID(\vec{q}_a).$$

Though proving strong containment is a conservative approach, it is in

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

keeping with the stated intent of the designer in requiring states with corresponding encodings to be compatible. We will now point out how the regularity of data can be exploited to further reduce the complexity of our verification algorithm for circuits with data paths.

2.4 Abstraction of data registers

As noted earlier, state machine comparison techniques can be applied to verify designs after the scheduling phase in the design process. Given that the implementation retains precise input-output behavior with respect to the specification and no behavioral transformations are done, corresponding registers in the data path of the specification and implementation have the same bit values on every clock cycle independent of the control state encoding. The specification can now be proved to be strongly contained in the implementation by identifying corresponding data registers. Algorithmic variables which will be synthesized as data registers can also be identified in behavioral specifications.

Given that the corresponding variables are data registers, we can exploit the regularity of data by introducing the concept of pairs of *meta* states, $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$, where each *meta* state pair actually represents 2^d state pairs (for all possible values of the missing d state variables). In the following, when we refer to a *meta* state pair, we mean all 2^d pairs of states represented by that pair, and successor states of *meta* states include successor states of all 2^d states. Computation of pairs of strongly compatible states is now replaced by computation of a subset of the pairs of strongly compatible states (Figure 2.4). At each step, we now compute a subset, $\tilde{SC}_{k+1}$, of the strongly $k+1$-compatible state pairs. $\tilde{SC}_{k+1}$ is the set of pairs of *meta* states, $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$, such

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Given: The d state variables with identical encoding.

1. $i = 1$;
2. $\tilde{SC}_1 = \{$pairs of $meta$ states that are strongly 1-compatible$\}$;
3. repeat
4. $\quad \tilde{SC}_{i+1} = \tilde{SC}_i$ - $\{$pairs of $meta$ states in $\tilde{SC}_i$ with successors not in $\tilde{SC}_i\}$;
5. $\quad i = i + 1$;
6. until $(\tilde{SC}_i = \tilde{SC}_{i-1})$;

Figure 2.4: Algorithm for computing a subset of the pairs of strongly compatible states.

that all 2^d pairs of states represented by each $meta$ state pair are strongly $k+1$-compatible and all successor states belong to the set of strongly k-compatible $meta$ states, $\tilde{SC}_k$. Equation 2.4 is now modified to give the characteristic functions of subsets of pairs of strongly compatible states as follows:

$$
\begin{aligned}
\tilde{SC}_1(\vec{q}_a^{\,s}, \vec{q}_b^{\,s}) &= \forall \vec{q}_a^{\,d}, \forall \vec{x} : \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b^{\,s} \cdot \vec{q}_a^{\,d})) \bigwedge \\
&\quad \bigwedge_{j=s+1}^{n} (\delta_{a_j}(\vec{x}, \vec{q}_a) \odot \delta_{b_j}(\vec{x}, \vec{q}_b^{\,s} \cdot \vec{q}_a^{\,d})) \bigvee INVALID(\vec{q}_a^{\,d}) \\
\tilde{SC}_{k+1}(\vec{q}_a^{\,s}, \vec{q}_b^{\,s}) &= \tilde{SC}_k(\vec{q}_a^{\,s}, \vec{q}_b^{\,s}) \bigwedge \forall \vec{q}_a^{\,d}, \forall \vec{x} : (\tilde{SC}_k(\vec{Q}_a^{\,s}, \vec{Q}_b^{\,s}) \vee INVALID(\vec{q}_a^{\,d})) \\
\tilde{\mathcal{SC}}(\vec{q}_a^{\,s}, \vec{q}_b^{\,s}) &= \tilde{SC}_k(\vec{q}_a^{\,s}, \vec{q}_b^{\,s}) \qquad\qquad if\ \tilde{SC}_{k+1} = \tilde{SC}_k \qquad (2.5)
\end{aligned}
$$

By universally quantifying over $\vec{q}_a^{\,d}$, $\tilde{SC}_1(\vec{q}_a^{\,s}, \vec{q}_b^{\,s})$ now represents pairs of $meta$ states, $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$, of which each pair actually represents 2^d strongly 1-compatible state pairs. The quantification is done over all valid values of the data registers. Again, valid states are not synonymous with reachable states. For example, an integer which takes values 0-5 in the specification may be represented as a bit vector of length 3. The values 6,7 are then invalid values.

For general purpose registers as found in microprocessors, there are usually no invalid values. Registers demarcated for specific functions in ASICs may require determination of invalid values. The quantification gives a conservative verification result if it checks for invalid data values, but never a false positive.

The quantification might eliminate some *meta* state pairs because all 2^d state pairs for those *meta* states are not strongly 1-compatible. Thus, the set $\tilde{SC}_1$ now contains not all strongly 1-compatible state pairs, but a subset of the strongly 1-compatible state pairs, represented by strongly 1-compatible pairs of *meta* states. The inverse image of set $\tilde{SC}_k$, restricted to the set $\tilde{SC}_k$, gives those strongly k-compatible state pairs all of whose successor states are such that their respective *meta* states are strongly k-compatible. Again, universally quantifying over valid $\vec{q}_a^{\,d}$ gives the corresponding *meta* states, and so on. The characteristic function for set $\tilde{SC}_k$ now depends on only variables $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$, and the complexity of computation of its inverse image is greatly reduced. When the fixed point is reached, the set $\tilde{SC}_k$ includes pairs of *meta* states which are both strongly k-compatible and strongly $k+1$-compatible. The algorithm always terminates because the sequence $\tilde{SC}_1, \tilde{SC}_2, ..$ is monotonically decreasing on a finite space $(\{0,1\}^s \times \{0,1\}^s)$.

Theorem 2 *The pairs of states computed by the algorithm presented in Figure 2.4 are strongly compatible.*

The proof of this theorem is similar to the proof presented in the last section. The difference is that at each step in the algorithm in Figure 2.4, we are computing only a subset of the pairs of strongly k-compatible states rather than the entire set of strongly k-compatible states as in the algorithm in Figure

2.3. Consequently, the lemma required to complete the proof must be modified accordingly.

Lemma 2 *If all pairs of meta states, $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$, belonging to set $\tilde{SC}_k$ also belong to set $\tilde{SC}_{k+1}$, $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$ are strongly compatible.*

Proof: We can easily prove by induction that, for all k, the state pairs belonging to set $\tilde{SC}_k$ are indeed strongly k-compatible, because they are strongly 1-compatible, and all their successor states belong to the set $\tilde{SC}_{k-1}$.

We now prove that *meta* states $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$ are strongly $k+2$-compatible by contradiction. Assume they are not strongly $k+2$-compatible. But we know that they are at least strongly 1-compatible, which implies some successor state pairs are not strongly $k+1$-compatible. However, all successor states of states in $\tilde{SC}_{k+1}$ belong to $\tilde{SC}_k$. Hence, by the statement of the lemma, all successor states of $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$ also belong to $\tilde{SC}_{k+1}$. Hence, all successor state pairs are strongly $k+1$-compatible, which leads to a contradiction. Hence, *meta* states $\vec{q}_a^{\,s}$ and $\vec{q}_b^{\,s}$ are strongly $k+2$-compatible and similarly, strongly k-compatible for all finite k. $\square$

Specification circuit C_a is strongly contained, and consequently contained, in implementation C_b if, for every valid state in C_a, there exists a strongly compatible state in C_b,

$$\forall \vec{q}_a, \exists \vec{q}_b \ : \ \tilde{\mathcal{SC}}(\vec{q}_a^{\,s}, \vec{q}_b^{\,s}) \ \vee \ INVALID(\vec{q}_a)$$

Implementation of these techniques involves, quite simply, taking the inputs to the delay elements of corresponding data registers in the two circuits

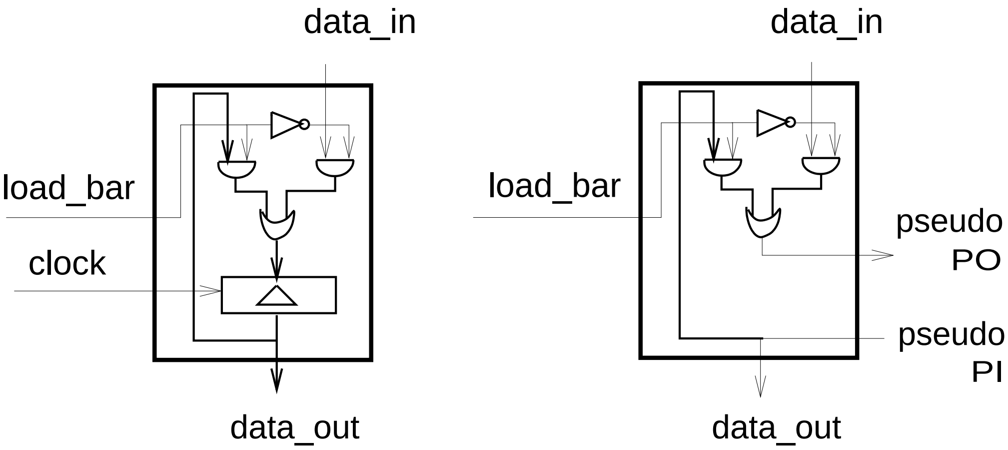


Figure 2.5: Register before and after abstraction.

as primary outputs and the corresponding outputs of the delay elements as primary inputs, dropping the delay elements, and applying the algorithm for computing pairs of compatible states. A single bit of a data register modified for proving strong containment is shown in Figure 2.5.

The existence of such a set of pairs of strongly compatible *meta* states is a special characteristic of circuits with data paths, where the d variables with identical values are data registers and the *meta* states are strongly compatible for all values in the data registers because of the **regularity of data**. To see that such *meta* states may not exist in general, consider the FSM and two implementations shown in Figure 2.6. The FSM has no inputs and one output. State variables q_{a_0} and q_{b_0} take the same values on every clock cycle. To prove strong equivalence, we make these variables pseudo-inputs and the corresponding next state variables pseudo-outputs, resulting in the modified machines shown in Figure 2.6. These machines have one input, one output and two (*meta*) states. As can be seen, there are no equivalent *meta* states in the two machines, the reason being there are no *meta* states equivalent for **all** values of q_{a_0} and q_{b_0}.

It is obvious that correspondence between all data registers is not necessary to prove strong containment. Most important, proving strong containment does not require the encoding of control states to be the same in the two descriptions! Thus, it is far more powerful than the simplistic method of using purely combinational techniques to verify two circuits which have the same encoding for all state variables. If the correspondence information is erroneous, the verification fails because, at each step, the set of pairs of strongly k-compatible states only includes those states all of whose successor states also have identical values for corresponding variables.

Techniques to exploit functional dependencies between variables in a circuit have also been developed by Hu and Dill [32]. They eliminate certain variables from the computation which are difficult to order in the BDDs by substituting the functions for those variables. These functions are provided by the user. This improves performance by eliminating the need to order those variables and reducing BDD sizes by reducing the number of BDD variables. This corresponds to the notion of strong containment presented in the last section. However, the further abstraction of data registers provides us with the maximum gain in efficiency. Hu and Dill look for functional dependencies between variables in the same circuit, whereas this work exploits dependencies between variables in two different implementations. They also appear to require more knowledge of the circuit on the part of the user to produce the requisite functional dependencies than this approach.

ps	ns	o/p
A	B	1
B	C	0
C	D	1
D	A	1

FSM

	qa1	qa0
A	0	0
B	0	1
C	1	0
D	1	1

impln. (1)

	qb1	qb0
A	0	0
B	1	1
C	1	0
D	0	1

impln. (2)

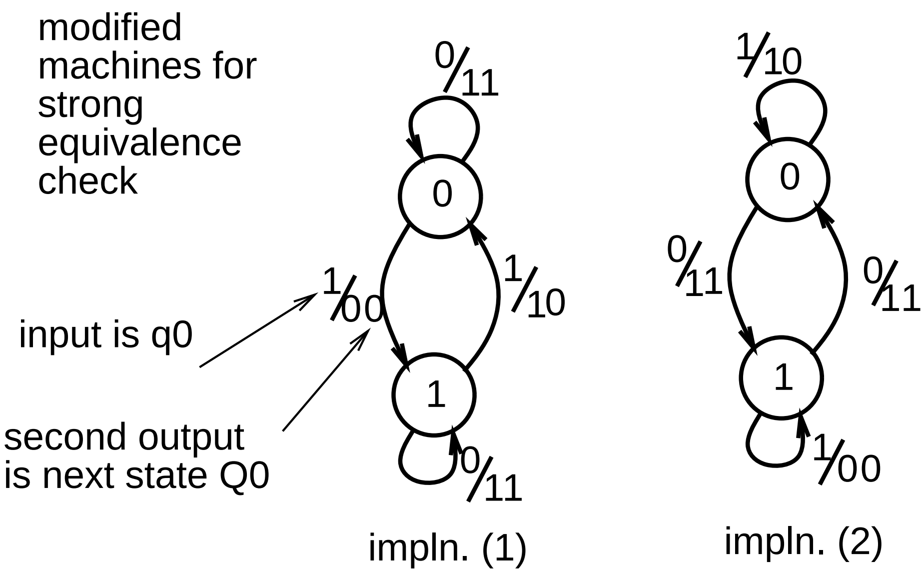


Figure 2.6: Counterexample for strong equivalence.

Table 2.1: Statistics of example circuits.

example number	*circuit*	*# inputs*	*# outputs*	*# states*	*Corresponding data registers*
1	Sequencer	6	17	109	-
2	CPU	9	20	10^{12}	39
3	AM2901	51	14	10^{20}	64
4	AM2910	22	16	10^{30}	84

2.4.1 Experimental Results

These verification techniques have been implemented in a C++ program which uses the OBDD package from CMU [45] and runs on an IBM RS 6000. They have been used to verify implementations of several industry-level circuits against their VHDL specifications at different levels of abstraction. The top level behavioral description is a single VHDL process with operations scheduled into clock periods or states, abstract data types like integers and no structural information. The middle level is a behavioral RTL description with the design partitioned into functional blocks. Details of the compilation of these descriptions into finite state machines are given in Chapter 5. All bi-directional buses have been split into two uni-directional buses. Tri-state buses have been made 2-valued, and their enable signals are checked during the verification. The values on drivers for such a bus are checked only when that driver is enabled. We have not encountered buses with multiple drivers in the designs we have verified. Multiple drivers can be handled similarly by introducing additional logic to verify correct mutually exclusive enabling of drivers and correctness of driven values at relevant times.

Table 2.2: Verification results on an IBM RS 6000 with 128Mb RAM.

no.	*Spec. type*	*Impln. type*	*# BDD nodes*	*Time to build BDDs*	*Time to find* SC_1	*Time to find* $\mathcal{SC}$	*Total time*
1	RTL	Gate	851	1.37s	1.81s	80.91s	84.49s
	RTL	Gate	855	1.32s	1.78s	84.46s	87.92s
2	Behavioral	RTL	2466	1.33s	0.87s	0.11s	2.41s
	RTL	Gate	2623	2.35s	0.74s	0.34s	3.44s
	Behavioral	Gate	2442	1.91s	1.02s	0.08s	3.03s
3	Behavioral	RTL	574390	380.71s	12.89s	30.28s	423.88s
	RTL	Gate	574315	367.03s	12.82s	30.05s	409.90s
	Behavioral	Gate	574317	365.64s	12.38s	30.18s	408.21s
4	Behavioral	RTL	12777	30.71s	3.33s	0.64s	34.69s
	RTL	Gate	5322	3.45s	1.74s	0.24s	5.43s
	Behavioral	Gate	12653	30.36s	3.35s	0.62s	34.35s

Table 2.1 shows four such circuits with the number of inputs, outputs, and state space. The last column shows the number of data registers for which correspondence information was derived from the circuit descriptions to prove strong containment. The **sequencer** is a control machine without a data path and no correspondence information could be extracted. The **CPU** is a simple 8-bit processor with a small instruction set of six opcodes. The **AM2910** and **AM2901** are a microprogram address sequencer and an ALU bit-slice, respectively, each with an on-chip RAM.

Table 2.2 gives the times to verify these circuits on an IBM RS 6000 for all possible combinations of specification and implementation. The sequencer has only a register transfer level specification. Two different gate level implementations were verified against the same RTL description. Verification was done by proving containment rather than strong containment. The CPU

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

has both a behavioral and a register transfer level specification. The gate level implementation has roughly 500 gates. Two errors were found in the implementation which prevented it from entering the halt state on encountering an illegal opcode. These errors were missed by the simulation vectors. The VHDL descriptions for the AM2901 and AM2910 were obtained from the MCNC high-level synthesis benchmark suite and were synthesized using the Viewlogic synthesis tool, giving implementations with approximately 1000 gates each. Discrepancies were found between the top level description and the functional block description for both circuits. In Table 2.2, the different columns show the number of nodes in the OBDDs, the times for building the BDDs, for computing pairs of strongly 1-compatible states, pairs of strongly compatible states and the total verification time, which includes the time to do the containment check. The BDD sizes are not optimal since the ordering heuristics used depend on structural information [46] which is absent in the VHDL specifications. Incorporation of better ordering algorithms [47,48] could improve the performance. As can be seen, proving strong containment requires orders of magnitude less time than required by conventional methods [12,39].

2.5 Containment by Partitioning Outputs

Though verification of circuits with large data paths has been made tractable by exploiting data register correspondence, the number of outputs to be verified is sometimes prohibitively large, especially in applications like signal processing. Furthermore, proving strong containment requires introduction of several additional primary inputs and outputs to the circuit, corresponding to the abstracted data registers. Ordering of variables in the BDDs representing output functions and next state functions or transition relations, and its effect on BDD size is a well-known problem. Choosing a single "best" variable ordering for the entire set of output functions in a circuit with a large input-output set will, more often than not, lead to an explosion in size for at least some of the outputs. The solution is to use a different variable ordering for different output functions, or for different sets of outputs. In this section, we will present such a technique for effectively dealing with large circuits by partitioning the outputs into disjoint sets.

Definition 14 *State s_a of specification circuit C_a and state s_b of implementation circuit C_b are* i**th-output compatible** *if and only if every input sequence that is applicable to C_a in s_a is also applicable to C_b in s_b, and the two circuits produce the same sequence of values for output i, with respect to the specified values in C_a, when C_a starts from state s_a and C_b starts from state s_b.*

Theorem 3 *The complete set of pairs of compatible states, $\mathcal{C}(\vec{q}_a, \vec{q}_b)$, for a specification C_a and an implementation C_b is given by the intersection of the sets of pairs of compatible states, $C^i(\vec{q}_a, \vec{q}_b)$, for each output i considered separately,*

$$\mathcal{C}(\vec{q}_a, \vec{q}_b) \;=\; \bigcap_i \; C^i(\vec{q}_a, \vec{q}_b) \; .$$

In other words, the set of pairs of compatible states, $\mathcal{C}$, for specification C_a and implementation C_b, each with l outputs, is given by the intersection of the sets of pairs of ith-output compatible states for every i from $1..l$. Verification can now be performed by considering the outputs one at a time, using different variable orderings, and computing the sets of pairs of ith-output compatible states, $\mathcal{C}^1 ... \mathcal{C}^l$. The implementation contains the specification if and only if for every state $\vec{q}_a$ in C_a, there exists a state $\vec{q}_b$ in C_b, such that $\vec{q}_a$ and $\vec{q}_b$ both belong to the set $\mathcal{C}^1 \cap \mathcal{C}^2 \cap \mathcal{C}^3 \cap \ldots \cap \mathcal{C}^l$.

Proof: Consider two sequential circuits C_a, the specification, and C_b, the implementation. The characteristic function of the set of 1-compatible state pairs (Equation 2.3) can be written as

$$C_1(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} c_1^i(\vec{q}_a, \vec{q}_b), \tag{2.6}$$

where each $c_1^i(\vec{q}_a, \vec{q}_b)$ is the characteristic function for the set c_1^i of pairs of 1-compatible states for output i and is given by

$$c_1^i(\vec{q}_a, \vec{q}_b) = \forall \vec{x} : (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b)). \tag{2.7}$$

Similarly, $c_k^i(\vec{q}_a, \vec{q}_b)$ represents the set of pairs of k-compatible states for output i. Following the derivation for $C_{k+1}(\vec{q}_a, \vec{q}_b)$ presented earlier (Equation 2.3), each successive $c_k^i(\vec{q}_a, \vec{q}_b)$ can be computed as

$$c_{k+1}^i(\vec{q}_a, \vec{q}_b) = c_k^i(\vec{q}_a, \vec{q}_b) \bigwedge \forall \vec{x} : c_k^i(\vec{Q}_a, \vec{Q}_b). \tag{2.8}$$

Using induction, we can prove that each function $C_k(\vec{q}_a, \vec{q}_b)$ can be written in the form $\bigwedge_{i=1}^{l} c_k^i(\vec{q}_a, \vec{q}_b)$. The base case for $k = 1$ is trivial (Equation 2.6). Assume that for any k,

$$C_k(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} c_k^i(\vec{q}_a, \vec{q}_b). \tag{2.9}$$

Now, from the characteristic function for C_{k+1} (Equation 2.3) and our induction hypothesis (Equation 2.9), we have

$$C_{k+1}(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} c_k^i(\vec{q}_a, \vec{q}_b) \ \bigwedge \ \forall \vec{x} : \bigwedge_{i=1}^{l} c_k^i(\vec{Q}_a, \vec{Q}_b).$$

This can be rewritten as

$$C_{k+1}(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} [\ c_k^i(\vec{q}_a, \vec{q}_b)\ \bigwedge\ \forall \vec{x} : c_k^i(\vec{Q}_a, \vec{Q}_b)\ ].$$

Therefore, using Equation 2.8,

$$C_{k+1}(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} c_{k+1}^i(\vec{q}_a, \vec{q}_b), \tag{2.10}$$

which proves that each $C_k(\vec{q}_a, \vec{q}_b)$ can be written in the form $\bigwedge_{i=1}^{l} c_k^i(\vec{q}_a, \vec{q}_b)$.

Now it remains to prove that each $c_k^i(\vec{q}_a, \vec{q}_b)$ is independent of other outputs. This is easily evident, by induction, from the predicates $c_1^i(\vec{q}_a, \vec{q}_b)$ (Equation 2.7) and $c_{k+1}^i(\vec{q}_a, \vec{q}_b)$ (Equation 2.8), both of which depend only on the ith output.

The set of pairs of compatible states $\mathcal{C}$ is the set C_k when the fixed point $C_{k+1} = C_k$ is reached, or equivalently, $\forall i (i = 1..l),\ c_{k+1}^i(\vec{q}_a, \vec{q}_b) = c_k^i(\vec{q}_a, \vec{q}_b) = \mathcal{C}^i(\vec{q}_a, \vec{q}_b)$, where $\mathcal{C}^i(\vec{q}_a, \vec{q}_b)$ is the characteristic function for the set of pairs of ith-output compatible states. Thus, the fixed point is reached only when the set of pairs of compatible states is found for each output. Since $C_k(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} c_k^i(\vec{q}_a, \vec{q}_b)$, it follows that, at the fixed point, the characteristic function for the set of compatible states is given by

$$\mathcal{C}(\vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} \mathcal{C}^i(\vec{q}_a, \vec{q}_b). \tag{2.11}$$

Finally, the specification is contained in the implementation if and only if

$$\forall \vec{q}_a,\ \exists \vec{q}_b : \bigwedge_{i=1}^{l} \mathcal{C}^i(\vec{q}_a, \vec{q}_b). \tag{2.12}$$

Thus, each $\mathcal{C}^i(\vec{q}_a, \vec{q}_b)$ can be computed independently using different variable orderings. $\square$

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Corollary *The intersection of sets of compatible state pairs for disjoint subsets of the output set of two sequential circuits gives the complete set of pairs of compatible states for the two circuits.*

We believe this result is indispensable for the verification of very large circuits with several inputs and outputs where building the BDDs is a problem. Each set $\mathcal{C}^i$ depends on state variables only, and using a common ordering for their intersection does not pose a problem. However, partitioning outputs does not appreciably improve the complexity of the inverse image computation itself and is therefore of marginal utility for smaller circuits.

The partitioning of outputs into subsets has been automated in the verification tool. This is achieved by building the BDDs for each successive output until the size of the BDDs increases beyond three times the BDD size before building the BDD for that output. A new group is then formed, starting with that particular output, and the same procedure is followed. More sophisticated partitioning techniques as in current variable ordering algorithms can also be used; for example, putting outputs with maximal overlapping cones together.

2.5.1 Experimental Results

Two of our experimental circuits, a microprocessor and a signal processing chip, required register correspondence as well as output partitioning to facilitate their verification. Their statistics are shown in Table 2.3. It is not possible to verify these circuits without resorting to strong containment, since there is no clear distinction between control and data path in the net lists. In both cases, we were able to extract correspondence information for most of the data registers. Of course, it is not necessary to specify correspondence between all

Table 2.3: Statistics of example circuits.

example number	*circuit*	*# inputs*	*# outputs*	*# states*	*Corresponding data registers*
5	Viper	32	53	10^{75}	248
6	Chip1	73	224	10^{160}	527

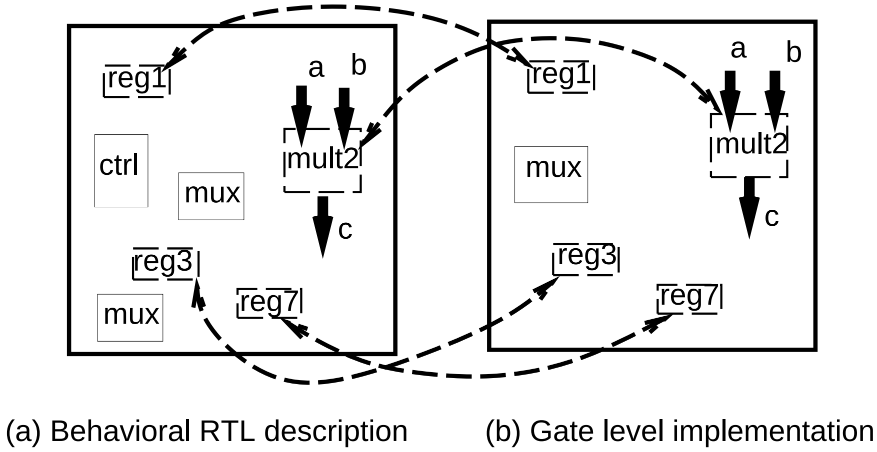


Figure 2.7: Correspondences in Chip1.

data registers to prove strong containment.

The **Viper** is a 32-bit microprocessor [49] designed for safety-critical applications. It is non-pipelined and has no interrupts. In our experiment, the specification is a scheduled behavioral description. The net list, synthesized by us, has 251 flip-flops and approximately 5000 gates. The 1M on-chip memory from the original description is omitted. A more complete description can be found in Chapter 5. **Chip1** is a signal-processing chip from LSI Logic with the specification at the register transfer level. The gate level has approximately 30,000 gates and 530 flip-flops. The net list has merged data path and control and a distinct control unit cannot be identified. However, the registers in the

Table 2.4: Verification of Chip1 using partitioned outputs.

Chip1		*Output partitions*			*Total for all partitions*	*All outputs together*
		Set1	*Set2*	*Set3*		
# BDD nodes		13.2K	28.5K	47.1K	47.1K (max)	876.8K
Time	*BDDs*	60.63s	181.25s	309.74s	676.02s	3.1h
	SC_1	13.8s	15.20s	88.18s	250.45	failed
	$\mathcal{SC}$	0.29s	0.33s	0.33s	1.58s	
Total verf. time		74.74s	196.80s	398.27s	15.5m	

RTL description and net list correspond (as shown in Figure 2.7) and they can be abstracted away to prove strong containment. This chip also has six multipliers. As is well known, multipliers have an exponential OBDD representation and require some other representation [50] to make their verification tractable. However, the multipliers in the net list and VHDL description of Chip1 also correspond, and we were able to remove them from the circuit and verify the chip down to the interface with the multipliers. In other words, we verified the correctness of the inputs to the multipliers while implicitly checking the design for all possible values at the multiplier outputs (In Figure 2.7, a and b are taken as pseudo outputs and c as pseudo input). This is again a conservative approach because we may be checking values which will not occur at the multiplier outputs during normal operation. However, both the Viper and Chip1 net lists were verified to correctly implement their specifications.

Abstraction of data path registers resulted in 601 primary inputs and 751 primary outputs for the signal-processing chip, and 283 primary inputs and 304 primary outputs for the Viper, leading to huge sizes for the OBDDs using a single variable ordering. Consequently, the verification did not complete.

Table 2.5: Verification of Viper using partitioned outputs.

Viper		*Output partitions*			*Total for all partitions*	*All outputs together*
		Set1	*Set2*	*Set3*		
# BDD nodes		5166	30.7K	74.3K	74.3K	failed
Time	*BDDs*	2.84s	236.57s	92.46s	331.87s	
	SC_1	0.84s	7.51s	17.25s	25.6s	
	$\mathcal{SC}$	0.38s	4.35s	0.54s	5.27s	
Total verf. time		4.06s	248.43s	110.25s	362.78s	

Partitioning the outputs into five sets for Chip1 resulted in a verification time of only 15 minutes. The outputs of the Viper were partitioned into three sets, and verification required approximately 6 minutes. Tables 2.4 and 2.5 show the results for verification by splitting the outputs and also by considering all outputs together. The BDD sizes shown are the combined BDD sizes for all outputs being considered at the time. The times to build the BDDs and find the set of strongly 1-compatible states and the set of strongly compatible states are shown. The verification for the case with all outputs considered together failed after 4 hours for Chip1 and failed even to build the BDDs for the Viper. Verification was also attempted using the recent innovation of dynamic variable ordering, where the ordering of the variables is changed dynamically during runtime whenever the BDD sizes cross a threshold [47]. This technique has been shown to handle large circuits, but is very slow due to the frequent re-orderings required. The verification was successful using dynamic ordering without output partitioning, but required 6 hours to verify Chip1 and 21 minutes to verify the Viper.

2.6 Extension to Handle Retiming Transformations

We have seen that verification of circuits which have data paths of practical size, through several stages in the design process, is tractable by using data register correspondences. However, retiming [51,52] is one optimizing transformation applied late in the design cycle that, while preserving the input-output behavior so that FSM verification techniques are still applicable, destroys the simple one-to-one relationship between data registers in a circuit and its retimed counterpart. This section presents an extension of the concept of strong containment which uses mapping functions between data registers in the two circuits for the verification of large circuits even after retiming transformations. Mapping functions do not have to be generated for all registers to use these techniques. The required mapping functions can either be generated automatically for a given retiming and for a specified set of flip-flops, using an algorithm like the one given in Section 2.7.1, or in the case of manual retiming, can be easily provided by the designer. Combinational optimization techniques (e.g., resynthesis) used on the logic between registers in tandem with retiming do not change the mapping between registers. Thus, these verification techniques can still be used to verify the correctness of the final implementation.

It is common practice to use heuristics to manually retime circuit implementations to improve performance or area, based on the designer's experience and without recourse to a retiming tool. It is then important to determine whether these changes have affected the functionality of the circuit. Even when incorporated in a synthesis tool, retiming is often used in conjunction with resynthesis and other optimizing transformations on the net list to find an implementation that best suits a designer's requirements. In this case, it is the execution of the retiming transformation or of other optimizing transformations that may be incorrect. An independent verification is thus desirable to ensure circuit correctness. A technique for verifying temporal behavior of retimed circuits has been presented in [53]. But their assumption of logical equivalence of the circuits, proven using existing methods, holds only for moderately sized circuits. An algorithm for finding the corresponding initial state after a legal retiming, assuming correct implementation of the retiming and when the initial state satisfies some reachability condition, is presented in [54]. However, as we have seen, initial states are often left unspecified, and proving equivalence of initial states is not a desirable approach to verification.

Before we move on to verifying retimed circuits, the following section gives a brief description of the retiming technique, the transformations involved and the implications.

2.6.1 Retiming Sequential Circuits

Retiming is an optimization technique for sequential circuits that moves registers in the circuit across blocks of combinational logic, while keeping the functional elements and circuit structure intact, in order to minimize the longest combinational path between flip-flops or minimize the number of registers in the circuit, or both. The relationship between a retimed register and corresponding registers in the original circuit is now dependent on the function of the logic block across which the register was moved during retiming.

A **legal** retiming of a sequential circuit preserves its functional behavior, that is, the finite state machines (FSMs) corresponding to the initial and final circuits are functionally equivalent [51]. Here functional equivalence means that for any state s entered by the original machine after being run for **a sufficiently long period of time** which is a function of the retiming, there exists a state s_r in the retimed machine such that the two machines exhibit the same behavior when started in these two states, *i.e.*, states s and s_r are **functionally equivalent**. This implies that not **all** states in the original machine will have equivalent states in the retimed machine. When retiming moves registers backward across logic blocks, some states in the original circuit may not have equivalent states in the retimed circuit. Correspondingly, when registers are moved forward across logic, some states in the retimed circuit may not be equivalent to any state in the original circuit.

The example in Figure 2.8 illustrates these points. State 00 in circuit(1) has no equivalent state in circuit(2). It may be necessary to determine such states and, in case of circuits with meaningful initial states, determine whether the retimed circuit has equivalent initial states. Also, if the original circuit was resettable, it is necessary to determine whether the retimed circuit is also resettable by the original initialization sequence. We present the important result that the correct implementation of a legal retiming ensures that the initialization sequence for the retimed circuit is a simple extension of the original initialization sequence.

We briefly discuss the transformation of a synchronous sequential circuit resulting from retiming. A more detailed explanation can be found in [51,52]. For the purpose of retiming, a sequential circuit is transformed into a directed graph G, with the nodes v of the graph representing combinational logic blocks and directed edges e being the interconnections between the blocks. Each edge is weighted with a register count $w(e)$, which is the number of registers along

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

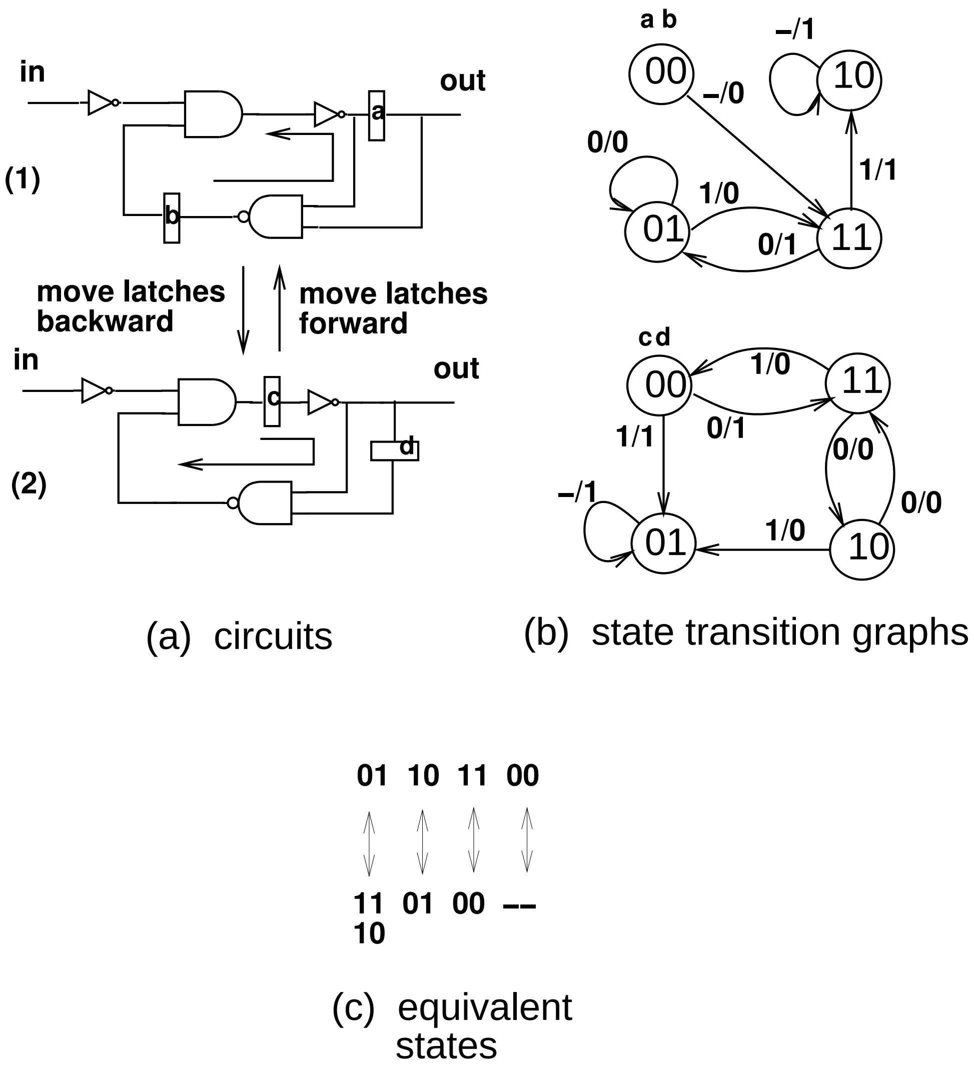


Figure 2.8: A legal retiming.

the connection. A special node called the host node v_h represents the external interface to the circuit. The graph for a synchronous circuit has a non-negative edge weight $w(e)$ for each edge and has no directed cycles with all edges of zero weight.

A retiming of a circuit is an integer-valued labeling r of the vertices of the corresponding graph G to transform it into a new graph G_r, where the new edge weight $w_r(e)$ for an edge $u \xrightarrow{e} v$ is given by

$$w_r(e) \;=\; w(e) \;+\; r(v) \;-\; r(u) \qquad .$$

A legal retiming is a vertex labeling that enforces the condition $w_r(e) \geq 0$ for every edge in G_r. The vertex labels are chosen in a manner that attempts to minimize some pre-defined cost function, such as the clock period or the number of registers. A positive label on a node v, called a positive retiming $r(v) = n > 0$, results in n registers being moved backward across the combinational logic block represented by that node, or, in other words, n registers are deleted from each output of v, and n registers are added to each input. A negative label, called a negative retiming $r(v) = -n < 0$, results in n registers being moved forward across the logic block represented by that node, or, in other words, n registers are deleted from each input of v, and n registers are added to each output. While it is fairly easy to check whether a vertex labeling is a legal retiming, verification of the final circuit is aimed at uncovering errors in the transformations performed on the circuit while implementing the retiming.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits," Ph.D. Dissertation, The University of Texas at Austin, December 1995.

2.6.2 Extending the Notion of Strong Containment

We can exploit the fact that retiming transformations do not change the structure of the circuit and result in simple functional relationships between registers in the original circuit and registers in the retimed circuit. We extend the notion of strong containment presented earlier to account for such complex mappings between the state encodings in the two circuits rather than simple one-to-one correspondence.

Definition 15 *State $s_a = \vec{q}_a = (q_{a_1}...q_{a_n})$ of specification circuit C_a and state $s_b = \vec{q}_b = (q_{b_1}...q_{b_n})$ of implementation circuit C_b are* **strongly k-compatible** *if they are k-compatible and there exists $\Gamma = [\gamma_1(\vec{q}_a)...\gamma_r(\vec{q}_a)]$, a functional vector $(\{0,1\}^n \rightarrow \{0,1\}^r)$ defining a set $S \subseteq S_b$, such that $\vec{q}_b^{\,r} = \Gamma(\vec{q}_a)$, and corresponding successor states of s_a in C_a and s_b in C_b, for all states reachable by an input sequence of length $\leq k$, also satisfy the functions γ_i for the same r bits in $\vec{q}_b$.*

All corresponding successor states of strongly k-compatible states are strongly $k-1$-compatible. Strongly compatible states are states which are strongly k-compatible for all values of k.

Definition 16 *State $s_a = \vec{q}_a = (q_{a_1}...q_{a_n})$ of specification circuit C_a and state $s_b = \vec{q}_b = (q_{b_1}...q_{b_n})$ of implementation circuit C_b are* **strongly compatible** *if they are compatible and there exists $\Gamma = [\gamma_1(\vec{q}_a)...\gamma_r(\vec{q}_a)]$, a functional vector $(\{0,1\}^n \rightarrow \{0,1\}^r)$ defining a set $S \subseteq S_b$, such that $\vec{q}_b^{\,r} = \Gamma(\vec{q}_a)$, and corresponding successor states of s_a in C_a and s_b in C_b, for all reachable states, also satisfy the functions γ_i for the same r bits in $\vec{q}_b$.*

Thus Γ defines a **functional relationship** between r state variables in implementation C_b and at most n state variables in specification C_a. In the simplest case, with Γ given by $\gamma_1 = q_{a_1}, \gamma_2 = q_{a_2}, .., \gamma_r = q_{a_r}$, it reduces to the one-to-one correspondence required in the earlier definition. If Γ is undefined for all $\vec{q}_b$, strong containment reduces to containment.

For the sake of simplicity, assume that Γ is defined for all state variables of circuit C_b. The predicate for computing the set of strongly 1-compatible states in C_a and C_b can be written as:

$$\tilde{SC}_1 = \forall \vec{x} : [\ (\Lambda_a \odot \Lambda_b) \bigwedge (\vec{q}_b \equiv \Gamma(\vec{q}_a)) \bigwedge (\vec{Q}_b \equiv \Gamma(\vec{Q}_a))\ ]$$

,

$$\tilde{SC}_1(\vec{q}_a) = \forall \vec{x} : [\ \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b)) \ \bigwedge \ \bigwedge_{j=1}^{n} (\delta_{b_j}(\vec{x}, \vec{q}_b) \odot \gamma_j(\vec{Q}_a))\ ].$$

Using composition, we have

$$\begin{aligned} \tilde{SC}_1(\vec{q}_a) = \forall \vec{x} : [\ & \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \Gamma(\vec{q}_a))) \bigwedge \\ & \bigwedge_{j=1}^{n} (\delta_{b_j}(\vec{x}, \Gamma(\vec{q}_a)) \odot \gamma_j(\Delta_a(\vec{x}, \vec{q}_a)))\ ]. \end{aligned} \tag{2.13}$$

$\tilde{SC}_1$ is a function of only $\vec{q}_a$ variables and is the set of states in C_a which have strongly compatible states in C_b. Therefore, to prove strong containment of C_a in C_b, it is sufficient to check whether $\tilde{SC}_1$ is a tautology.

Now consider the general case where Γ is defined only for r of the state variables ($\vec{q}_b^{\,r}$) of C_b over a subset of state variables ($\vec{q}_a^{\,g}$) of C_a. Let $\vec{q}_b \equiv \vec{q}_b^{\,s} \& \vec{q}_b^{\,r}$ and $\vec{q}_a \equiv \vec{q}_a^{\,h} \& \vec{q}_a^{\,g}$ with Γ $(\{0,1\}^g \rightarrow \{0,1\}^r)$. In this case, $\tilde{SC}_1$ can be written as:

$$\tilde{SC}_1 = \forall \vec{x} : [\ (\Lambda_a \odot \Lambda_b) \bigwedge (\vec{q}_b^{\,r} \equiv \Gamma(\vec{q}_a^{\,g})) \bigwedge (\vec{Q}_b^{\,r} \equiv \Gamma(\vec{Q}_a^{\,g}))\ ],$$

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

$$\tilde{SC}_1(\vec{q}_a, \vec{q}_b^{\,s}) = \forall \vec{x} : [\ \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b^{\,s}, \vec{q}_b^{\,r}) \bigwedge \bigwedge_{j=1}^{r} (\delta_{b_j}(\vec{x}, \vec{q}_b^{\,s}, \vec{q}_b^{\,r}) \odot \gamma_j(\vec{Q}_a^{\,g}))\ ].$$

Using composition,

$$\tilde{SC}_1(\vec{q}_a, \vec{q}_b^{\,s}) = \forall \vec{x} : [\ \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b^{\,s}, \Gamma(\vec{q}_a^{\,g})) \bigwedge \bigwedge_{j=1}^{r} (\delta_{b_j}(\vec{x}, \vec{q}_b^{\,s}, \Gamma(\vec{q}_a^{\,g})) \odot \gamma_j(\Delta_a^g(\vec{x}, \vec{q}_a))\ ]. \tag{2.14}$$

$\tilde{SC}_1$ is now a function of $\vec{q}_a$ and $\vec{q}_b^{\,s}$, but not $\vec{q}_b^{\,r}$. Thus, we have again reduced the number of variables on which $\tilde{SC}_k$ depends and reduced the complexity of the inverse image computation. And again, we can go further by observing that universally quantifying over $\vec{q}_a^{\,g}$ gives *meta* state pairs, $\langle \vec{q}_a^{\,h}, \vec{q}_b^{\,s} \rangle$, each of which represents 2^g strongly 1-compatible pairs of states, $\langle \vec{q}_a, \vec{q}_b \rangle$, such that r bits of $\vec{q}_b$ and g bits of $\vec{q}_a$ satisfy Γ, and those r and g bits of the corresponding next states also satisfy Γ. Note that the state pairs may not include all possible values for the r bits in $\vec{q}_b$. Again, the g variables are data registers and, as in the earlier data abstraction technique, the existence of such *meta* state pairs is chiefly due to the regularity of data.

$$\tilde{SC}_1(\vec{q}_a^{\,h}, \vec{q}_b^{\,s}) = \forall \vec{q}_a^{\,g}, \forall \vec{x} : [\ \bigwedge_{i=1}^{l} (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b^{\,s}, \Gamma(\vec{q}_a^{\,g})) \bigwedge \bigwedge_{j=1}^{r} (\delta_{b_j}(\vec{x}, \vec{q}_b^{\,s}, \Gamma(\vec{q}_a^{\,g})) \odot \gamma_j(\Delta_a^g(\vec{x}, \vec{q}_a))\ ]. \tag{2.15}$$

We proceed with the fixed point computation to find the set of strongly compatible states. But now $\tilde{SC}_k$ has a substantially smaller number of variables. We compute $\tilde{SC}_{k+1}$ as those pairs of states in $\tilde{SC}_k$ whose corresponding next states, for all values of $\vec{q}_a^{\,g}$, also belong to $\tilde{SC}_k$.

$$\tilde{SC}_{k+1} = \tilde{SC}_k(\vec{q}_a^{\,h}, \vec{q}_b^{\,s}) \bigwedge \forall \vec{q}_a^{\,g}\ \forall \vec{x} : \tilde{SC}_k(\vec{Q}_a^{\,h}, \vec{Q}_b^{\,s}).$$

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits," Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Using composition again,

$$\tilde{SC}_{k+1}(\vec{q}_a^{\,h}, \vec{q}_b^{\,s}) = \tilde{SC}_k \bigwedge \forall \vec{q}_a^{\,g}\ \forall \vec{x} : \tilde{SC}_k(\Delta_a^h(\vec{x}, \vec{q}_a), \Delta_b^s(\vec{x}, \vec{q}_b^{\,s}, \Gamma(\vec{q}_a^{\,g}))). \quad (2.16)$$

$$\tilde{\mathcal{SC}} = \tilde{SC}_k \quad if \tilde{SC}_{k+1} = \tilde{SC}_k$$

To prove strong containment of circuit C_a in C_b, we must show

$$\forall \vec{q}_a, \exists \vec{q}_b : \tilde{\mathcal{SC}}(\vec{q}_a^{\,h}, \vec{q}_b^{\,s}). \quad (2.17)$$

The algorithm terminates because the sequence $\tilde{SC}_1, \tilde{SC}_2, ..,$ is a monotonically decreasing sequence. Traversal of $\tilde{\mathcal{SC}}$, along with use of the mapping functions Γ for $\vec{q}_a^{\,g}$ and $\vec{q}_b^{\,r}$, will give all pairs of strongly compatible states in C_a and C_b for the functions Γ, while $\neg\tilde{\mathcal{SC}}$ will give all states in C_a which do not have strongly compatible states in C_b. The proof that this algorithm is correct is very similar to the proof for strong containment using one-to-one correspondences presented in Section 2.4 and is not discussed here.

2.7 Application to Retimed Circuits

Application of the above techniques to retimed circuits requires certain proof obligations which we will now consider. The existence of appropriate mapping functions depends on the nature of the retiming. In order to apply the above techniques to the verification of retimed circuits, we will consider the cases of negative and positive retiming of circuits separately before considering the general case. We will assume node labels in the retiming r take the values -1, 0 or 1. Retimings with labels $r(v) < -1$ or $r(v) > 1$ can be decomposed into a sequence of negative or positive retimings of node v by one unit. However, a sequence of retiming transformations requires consecutive retimings to be

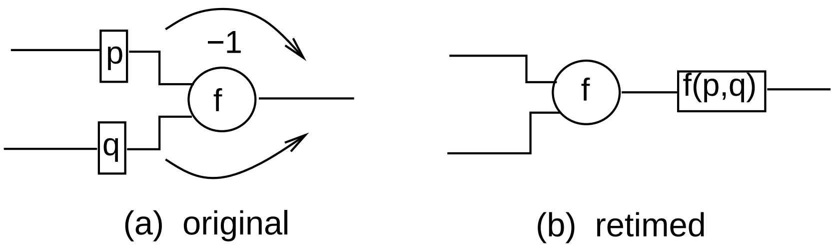


Figure 2.9: Negative retiming of a combinational logic node.

consistent, *i.e.* a node label cannot change sign from one retiming to another, for the mapping functions to be valid. This precludes use of this technique for certain illegal retimings like peripheral retiming [55].

Consider the negative retiming of a combinational logic node, $r(v) = -1$, which results in registers being moved forward across the node. The values taken by the registers created at the outputs are the image by the logic function f at the node v of the values taken by the registers at the inputs of v before retiming, as shown in Figure 2.9. Consequently, if all nodes in the circuit are retimed by -1 or 0, there exists a straightforward mapping from states in the original circuit to states in the retimed circuit that is given by the logic functions at the retimed nodes. These mappings between state variables in the original circuit and the retimed circuit can be used in our verification technique to prove that the original circuit is strongly contained in the retimed circuit, though the two circuits may not be equivalent. For example, if circuit (2) is the original circuit and circuit (1) is the retimed circuit in Figure 2.8, the mapping functions are $a = \neg(d \wedge \neg c)$ and $b = \neg c$.

In case of positive retiming of a node, $r(v) = 1$, registers are moved backward over the logic node, and the values taken by them are given by the reverse image by the function f of the values taken by the registers at the outputs of the node before retiming. Mapping functions are now computed by considering the retimed circuit C_b as the original circuit and reversing the node labels in sign so that they are now all lesser than or equal to zero. Mapping functions can now be computed for all registers in the original circuit C_a in terms of registers in the retimed circuit C_b (Figure 2.9). We now have $\vec{q}_a^{\,r} = \Gamma(\vec{q}_b^{\,g})$, with Γ defining a set $S \subseteq S_a$. Since we have mapping functions for state variables in the original circuit C_a in terms of state variables in the retimed circuit C_b, the predicates for sets of strongly k-compatible states will depend on $\vec{q}_b^{\,n}$ and $\vec{q}_a^{\,s}$. In order to check for containment of C_a in C_b, we must perform the universal quantification in Equations 2.13 – 2.16 over variables $\vec{q}_a^{\,r}$ of C_a, not $\vec{q}_b^{\,g}$ of C_b. This is accomplished in the following manner after substitution of $\vec{q}_a^{\,r}$ by the functions $\Gamma(\vec{q}_b^{\,g})$ for each predicate $\tilde{SC}_k$,

$$\tilde{SC}_k(\vec{q}_a, \vec{q}_b^{\,h}) = \exists \vec{q}_b^{\,g} : [\ \tilde{SC}_k(\vec{q}_a^{\,s}, \vec{q}_b^{\,n}) \bigwedge (\bigwedge_{i=1}^{r} (q_{a_i} \odot \gamma_i(\vec{q}_b^{\,g}))\ )\ ] \tag{2.18}$$

Universal quantification is then done over variables $\vec{q}_a^{\,g}$ for these modified predicates, and containment of the original circuit in the retimed circuit is checked in Equation 2.17.

Even in case of a legal retiming implemented correctly, C_a may not be contained in C_b. It is easy to characterize states in C_a which possibly do not have compatible states in the retimed circuit.

Definition 17 *A k-realizable state is a state which can be reached from some state in k or more transitions. A state reachable from itself is k-realizable for all values of k.*

Definition 18 *A state which is not k-realizable for any k is an unrealizable state.*

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

The reverse image by the mapping functions Γ for state s_a in C_a can be empty only if s_a cannot be reached from any state and is an unrealizable state. This is apparent from the fact that a state s does not have a corresponding state in the retimed circuit only if there are no values of the inputs and no values on the wires of the circuit and in the registers which can result in the values of s being loaded into the registers on the next clock, which means state s is an unrealizable state. **This is only a necessary condition, not a sufficient one**. A sequence of k positive retimings of a node by $+1$, *i.e.* moving k registers backward over the node, can result in a state s not having a corresponding state in the retimed circuit only if s cannot be reached from any state in k or more transitions. The following theorem is essentially the same as Theorem 3.1 in [54] and follows directly from the observations above.

Theorem 4 *A k-realizable state s will always have an equivalent state in the retimed circuit if the maximum positive value by which any node in the circuit is retimed is k, and the retiming is a legal retiming.*

In Figure 2.8, state 00 in circuit(1) is unrealizable and has no equivalent state in circuit(2). All other states are realizable and have equivalent states in circuit(2).

In the general case, when a retiming may have node labels with both positive and negative values, Theorem 4 still holds [54]. The mapping functions Γ can now be split into two sets, those expressing certain state variables in C_a in terms of certain state variables in C_b and those expressing certain other state variables in C_b in terms of certain other state variables in C_a. Note that the set of variables contributing to the domains of functions in Γ and the set of

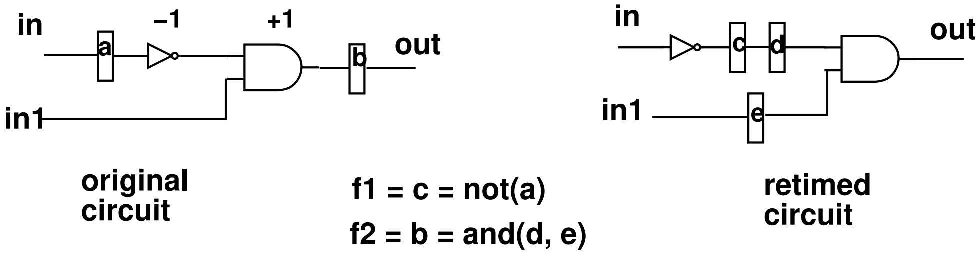


Figure 2.10: Mapping functions for a "mixed" retiming.

variables contributing to ranges will always be mutually exclusive for a given retiming. For example, a retiming with both -1 and +1 labels has mapping functions as shown in Figure 2.10. For ease of manipulation, it is advisable to use only one set of mapping functions (for negative node labels) and ignore the other set.

One pitfall which results in false negatives, and which should be avoided while quantifying over variables $\vec{q}_a^{\,g}$ in Equation 2.16, is neglecting to compose **all** variables in $\vec{q}_b^{\,s}$ which are related to variables in $\vec{q}_a^{\,g}$ by some mapping function. This will become clear when we consider the implications of the quantification operation in Equation 2.16. Recall that we check 1-compatibility of only those states with encodings $\vec{q}$ satisfying Γ, verify that the corresponding next states $\vec{Q}$ also satisfy Γ, and quantify over the current-state variables $\vec{q}$. This implies that we are essentially considering the register outputs $\vec{q}$ as pseudo-primary inputs to the circuit rather than as state variables, and the register inputs $\vec{Q}$ as pseudo-primary outputs rather than as next-state variables. When dropping registers from the circuit in this manner, we should take care to keep the temporality [53] of all outputs in the circuits being verified the same, *i.e.* the number of flip-flops on any path from input to output in the graphs G and G_r is now reduced but should be the same for both graphs. For example,

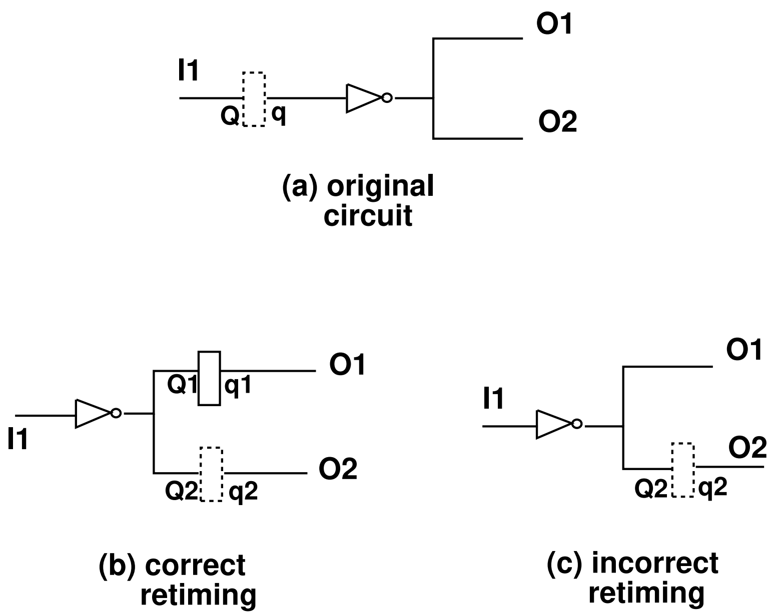


Figure 2.11: Retaining temporal equivalence.

if circuit(a) in Figure 2.11 is the original circuit and circuit(b) is the retimed circuit, both q_1 and q_2 in circuit(b) must be composed while computing $\tilde{SC}_1$. Composing only q_2 will destroy the temporal equivalence of output O_1 in circuit(a) and circuit(b) as implied by dropping the dotted flip-flops in circuit(a) and circuit(b) in Figure 2.11. Verification will then fail even though the actual circuits are logically equivalent, resulting in a false negative. For retimed circuits, this can be avoided by generating mapping functions for all flip-flops in the **dependency set** of a register, as we will see in Section 2.7.1.

However, if the retiming itself is incorrect and $q1$ has been erroneously omitted in the retimed circuit (circuit(c) in Figure 2.11), verification of the incorrectly retimed circuit would be attempted using the mapping function for $q2$ only. Now the two circuits, circuit(a) and circuit(c), are temporally equivalent after dropping the dotted flip-flops during verification. However, the

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

verification will prove they are not input-output equivalent. This is because even after dropping registers q and $q2$, $O1$ is a function of pseudo-primary input q in circuit(a), but a function of primary input $I1$ in circuit(c), *i.e.*, the modified circuits are not logically equivalent though they are temporally equivalent.

2.7.1 Generation of Mapping Functions

Mapping functions are generated for a particular retiming incrementally by considering a single node at a time and composing the functions generated. In general, mapping functions are generated only for a subset of the state variables, usually those representing data registers. Retimings with $r(v) > 1$ or $r(v) < -1$ can be decomposed into a sequence of retimings with labels of -1, 0 or 1. For a node v which has a positive label $r(v) > 0$, we generate mapping functions expressing deleted state variables on the outputs of v as functions of newly created state variables on the inputs of v, *i.e.*, effectively reversing the sign of $r(v)$.

We first define certain terms useful for the understanding of the algorithm.

Definition 19 *The* **distance** *of a register q on edge $u \xrightarrow{e} v$ from node u is one more than the number of registers between q and u on the edge e.*

For a given retiming r, the **retiming of a register** q on edge $u \xrightarrow{e} v$ means moving q either forward or backward, depending on the labels $r(u)$ and $r(v)$, as well as deleting and creating registers in the circuit as required to accomplish this movement, and further doing the same for each newly created register. Register q is said to be retimed when none of the newly created registers needs to be moved any further. Note that all nodes in the circuit may not need to be retimed for q to be retimed.

Definition 20 *The* **dependency set** *of register q is made up of all registers deleted from the original circuit and all registers created in the retimed circuit when q is retimed.*

As we saw earlier, we need to retain the temporal equivalence between the original circuit and the retimed circuit during verification and the attendant dropping of registers by quantification. Two circuits are temporally equivalent if the temporality of each output is the same for both circuits along each path from a primary input to a primary output in the corresponding circuit graphs. Note that retiming does not change topology of the graph.

Theorem 5 *Deleting all registers in the dependency set of a register q for a given retiming r from the original circuit and the retimed circuit preserves the temporal equivalence of the two circuits.*

Proof: Decompose r into a sequence of retimings $r_1, r_2, .., r_n$, such that each r_i has a single node with label -1 or +1 and label 0 for all other nodes. Let the circuit obtained after each retiming r_i be C_i. Then C_n is the circuit obtained after applying retiming r to the original circuit. For each retiming r_i, it is obvious from the definition of retiming that removing the registers from C_{i-1} which are deleted by retiming r_i and removing the registers from C_i which are created as a result of r_i preserves the temporal equality of C_{i-1} and C_i. This holds true for each r_i and hence also for their composition ($\cup_i\ r_i = r$). $\square$

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Generate$(q,\ e,\ d_u,\ d_v,\ r)$
{To generate mapping functions for the dependency set of register q on edge $u \xrightarrow{e} v$. The distance of q from u is d_u and from v is d_v.}

1. If $r(v) \neq 0$ and $r(v) + d_v \leq 0$
   check each input of v
   else go to step 3.
   If any in-edge $e1$ to node v does not have a register
   Back_Gen$(e1)$ and go to step 1
   else continue.
2. Retime v by one unit. Generate a new register for each output of v and compute the mapping function (Figure 2.9).
   Increment $r(v)$ by one and compute new weights w_r.
   If q has not been moved yet
   go to step 1
   else go to step 5.
3. If $r(u) \neq 0$ and $r(u) - d_u \geq 0$
   check each output of u
   else exit.
   If any outedge $e1$ from node u does not have a register
   Forward_Gen$(e1)$ and go to step 3
   else continue.
4. Retime u by one unit. Generate a new register for each input of u and compute the mapping function for each output register (Figure 2.9).
   Decrement $r(u)$ by one and compute new weights w_r.
   If q has not been moved yet
   go to step 3
   else go to step 6.
5. For each outedge $e1$ from v with newly created register $q1$
   Generate$(q1,\ e1,\ 0,\ w_r(e1),\ r)$.
6. For each in-edge $e1$ to u with newly created register $q1$
   Generate$(q1,\ e1,\ w_r(e1),\ 0,\ r)$.

Figure 2.12: Algorithm for generating mapping functions.

Consequently, using mapping functions for the entire dependency set of a register ensures correctness of the verification technique. When a circuit is retimed manually, mapping functions can be generated by keeping track of all registers deleted and all newly created registers while retiming a particular register. For a given retiming r, an algorithm which can be used to generate the mapping functions for the dependency set of a register q is given in Figure 2.12. Generation of mapping functions is independent of the actual retiming transformations. It is conceivable that errors in the transformations affecting temporal behavior may be uncovered during generation of the mapping functions themselves. However, absence of such errors does not imply logical equivalence, especially after other optimizing transformations have been applied, and further verification is a must.

Step 1 and step 3 in the algorithm check whether the register being retimed needs to be moved. If so, the register is moved, and mapping functions are generated as appropriate. Other nodes may have to be retimed to accomplish this movement if some edges do not have the required registers. The subroutine *Back_Gen*(e) creates a new register on an edge $u \xrightarrow{e} v$ with $w(e) = 0$ and $r(u) < 0$ and generates corresponding mapping functions by retiming node u. If necessary, it calls itself recursively on in-edges to u. Similarly, the subroutine *Forward_Gen*(e) creates a new register on an edge $u \xrightarrow{e} v$ with $w(e) = 0$ and $r(v) > 0$ and generates corresponding mapping functions by retiming node v. If necessary it also calls itself recursively on outedges from v. The result of this algorithm is a list of functions expressing mappings between registers in the dependency set of q. In the worst case, all nodes in the circuit may need to be retimed if the dependency set of q contains all the registers.

Table 2.6: Verification results for example circuits on an IBM RS6000.

example	*Original Circuit*		*Retimed Circuit*		*# BDD nodes*	*CPU time*
	Cr. path	*# FFs*	*Cr. path*	*# FFs*		
ex1	15	6	14	8	956	0.68s
ex2	19	6	13	7	3260	1.50s
ex3	9	14	8	15	4003	1.11s
ex4	17	32	15	18	1986	1.28s
ex5	-	558	-	530	68.7K	17.42m
ex6	34	63	17	112	1607	1.20s

2.7.2 Experimental Results

These techniques were used to verify some circuits of practical size after retiming the original circuit for reducing area or clock period (Table 2.6). The last two columns show the maximum number of BDD nodes and the CPU time on an RS 6000 required for verification. In these experiments, the circuits were retimed manually, and mapping functions were generated by symbolically simulating the logic nodes.

ex1, ex2, ex3 are FSMs from the MCNC suite which were retimed to reduce the length of the combinationally longest sensitizable path. Mapping functions were used for approximately 50% of the flip-flops during verification.

ex4 is a two-stage adder for adding four 8-bit numbers, composed of three 8-bit full adders. Each full adder is a ripple carry adder. It is split into two pipeline states after retiming. Mapping functions were used for all registers. Our attempts at verification without mapping functions failed for this example, as also for the next two examples.

ex5 is a signal-processing chip from LSI Logic (Chip1 from Section 2.5.1), synthesized from a VHDL description using a high-level synthesis tool. The implementation has merged data path and control with more than 30,000 gates, with multipliers, adders and subtractors being the primary functional units. An optimized implementation with 530 flip-flops was verified against an implementation with 558 flip-flops. Verification was performed down to the I/O of the multipliers, using mapping functions for the data registers but not for registers in the control path. The optimized implementation was also verified against its RTL description, again using mapping functions for data registers, which took 15.5 minutes. The verification for this chip was, of course, performed by partitioning outputs.

ex6 is a 32 bit serial-parallel multiplier [56]. A 4 bit version is shown in Figure 2.13(a). The nodes v_1 through v_4 are summand-adders, each with a delay of 2 units, and the nodes v_5 through v_8 are broadcast nodes each with a delay of 1 unit. The nodes v_9 through v_{12} each has a delay 0.5 units. The circuit has 8 flip-flops and a clock period of 6 units. It can be retimed to obtain a circuit with 14 flip-flops and a clock period of 3 units shown in Figure 2.13(b). The 32 bit multiplier has 63 flip-flops and a clock period of 34 units. It was retimed to reduce the clock period to 17 units while increasing the flip-flop count to 112. Verification took 1.2 seconds with all but two flip-flops in the original circuit mapped to flip-flops in the retimed circuit, and 27.8 minutes when 75% of the flip-flops in the original circuit were mapped.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

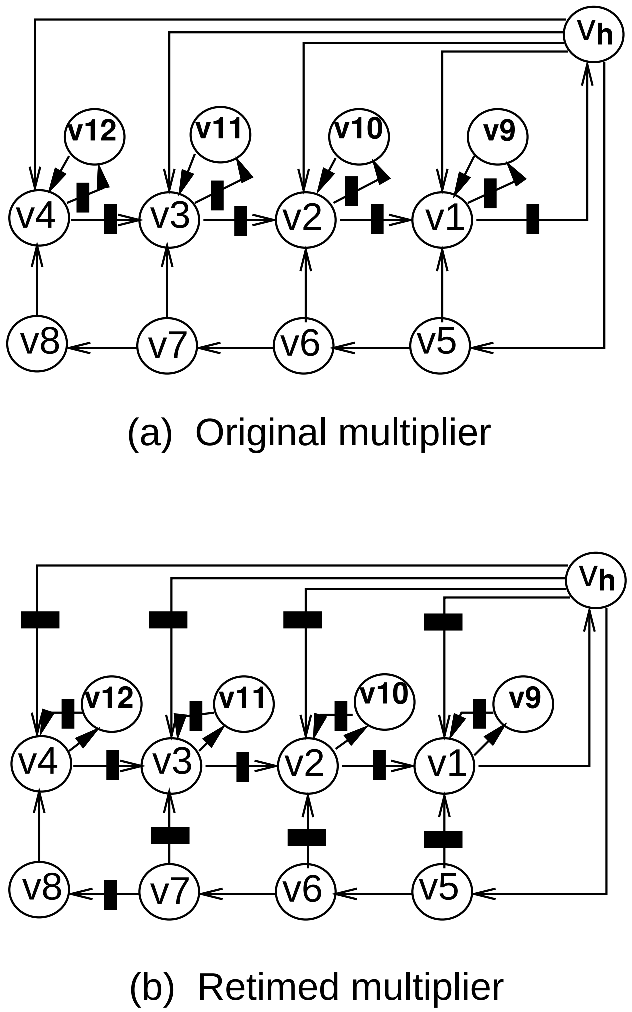


Figure 2.13: Graph for a 4x4 bit-serial multiplier.

2.8 Reset State and Initialization Sequence After Retiming

As a consequence of Theorem 4, a state reachable from itself always has an equivalent state in a legally retimed circuit. A reset state is a state which can be reached from all states by application of the same input sequence, irrespective of the starting state. Such a sequence is called a reset sequence or initialization sequence. Thus, a reset state is k-realizable for all k. A reset state

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

will always have an equivalent state in the retimed circuit if the retiming is legal and implemented correctly. A reset state always has a reset or initialization sequence associated with it which needs to be determined for the retimed circuit as well. A negative retiming may give rise to states in the retimed circuit which do not have equivalent states in the original circuit and which may not obey the original reset sequence.

It is worth noting that any state reachable from a reset state is also a reset state, and hence, k-realizable for all k. It is also worth noting that an initialization sequence appended or prefixed with any number of arbitrary inputs is also an initialization sequence.

Theorem 6 *An initialization sequence for a retimed circuit, after a correctly implemented legal retiming, can be derived from the initialization sequence for the original circuit by prefixing it with an arbitrary input sequence of length k, where k is the maximum negative value by which any node in the circuit was retimed.*

Proof: Let C and C_r represent the original circuit and the circuit obtained after a retiming r respectively. Assume the circuits have been verified using the techniques presented earlier. We consider two cases.

Case 1: All node labels in the retiming r are positive. In this case, C can be considered to be the retimed version of C_r with all node labels reversed in sign. Each state in C_r is then equivalent to some state in C. Consequently, the initialization sequence for C_r is the same as that for C.

Case 2: Some or all node labels in r may be negative. In this case, there may be states in the retimed circuit which do not have any equivalent states in

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits," Ph.D. Dissertation, The University of Texas at Austin, December 1995.

the original circuit. Again, consider that C may be seen as the retimed version of C_r with node labels reversed in sign. By Theorem 4.1, all k-realizable states in C_r have equivalent states in C. States which are not k-realizable may be either unrealizable, 1-realizable,..., up to $(k-1)$-realizable. All states which have incoming transitions from unrealizable states are at least 1-realizable, since that transition itself satisfies the condition of 1-realizability. Similarly, all states with transitions from 1-realizable states are at least 2-realizable and so on. All $(k-1)$-realizable states must have all transitions into states which are at least k-realizable. Thus, applying any arbitrary input sequence of length k to any state in C_r results in the final state always being at least k-realizable. All k-realizable states in C_r have equivalent states in C and obey the reset sequence. Hence, the reset sequence for C prepended with an arbitrary input sequence of length k is a reset sequence for C_r. $\square$

Thus, verification of the retimed circuit also ensures that the initialization sequence for the retimed circuit is a simple extension of the initialization sequence for the original circuit.

2.9 Debugging Techniques

Location of errors in the design is just as important as error detection. In fact, most of the time spent in circuit verification is devoted to finding the cause of errors and correcting them. This problem needs to be addressed in the area of implementation verification of sequential circuits as well, not just in design verification. Most of the work done in this direction deals with combinational circuits or sequential circuits in which the state encoding is known [57]–[59].

This is applicable when there exists a correct version of the net list, and the designer wishes to check for errors in a net list optimized manually but leaving the same state encoding. These techniques are also applicable when the designer has specified the state encoding to be used by a synthesis tool and wishes to verify the synthesized net list. In this problem domain, these techniques have been shown to be very efficient in detecting a large class of commonly occurring errors.

Several such types of low-level design errors are identified in [57,59] and targeted by the diagnosis procedure. These include a missing input to a gate, use of a wrong logical gate and a single missing gate, among others. It is assumed that only a single design error occurs at a time, though the same techniques would also be useful when multiple errors occur. Essentially, all these techniques compare the Boolean functions for all outputs and state elements in the correct circuit and the faulty circuit and then use efficient techniques to narrow the search for the error in the set of circuit elements affecting the incorrect function. Some error correction procedures, when the error can be corrected without changing the global structure of the circuit, are also described.

While these techniques will increase the utility of any verification tool, a multilevel framework requires diagnosis procedures targeting higher level design errors as well. Errors can be introduced at any stage in the design cycle before a gate level description is even synthesized. Such errors change the global structure of the state transition graph of the design. Cheng and Jou [60] have proposed just such a functional fault model which assumes a single transition in the circuit is incorrect, *i.e.*, either some output value on that transition is incorrect or the next state is wrong or both. Such errors can occur when a higher level description is being refined into an RTL description by partitioning and allocation or when input to a synthesis tool is being generated from a higher level description. Cheng and Jou generate a test suite to detect such faults using heuristics to narrow the possible incorrect destination states for each faulty transition [60]. However, they only handle very small machines with their technique and make the assumption that input sequences to travel between all pairs of states are pre-computed, which is not feasible for large machines. Their requirement of the same number of states in faulty and fault-free machines is also too restrictive for general applicability.

Implicit state enumeration techniques have been used to generate diagnostic sequences for larger machines by generating differentiating sequences for the faulty and fault-free machines [61]. The procedure implementing the basic algorithm is shown in Figure 2.14. The product machine of the faulty and fault-free machines is formed with output functional vector $\Lambda_p = \Lambda_{faulty} \odot \Lambda_{faultfree}$ and next-state functional vector $\Delta_p = (\Delta_{faulty}, \Delta_{faultfree})$. Basically, an output of the product machine is 0 whenever the value for that output does not match for the two component machines and 1 otherwise. The characteristic function of the set of transitions with matching outputs is generated (step 3). Starting from a given initial state of the product machine (s_0), the algorithm performs a forward traversal, saving the set of newly visited states (new_i) at each step, until a discrepancy in some the output for the component machines is encountered (step 5). A sequence of input vectors (inp_j) to this product machine state is then generated (steps 6–14) using the saved state sets (new_i) to do a backward traversal from the state showing incorrect outputs (s_n) back

$new_0 = s_0 = Reached =$ initial state(s);
$i = 0$;
$\tau = \bigwedge_i \lambda_{p_i}$;
while $(new_i \neq 0)$
    if $(new_i \wedge \bar{\tau} \neq 0)$
        $(inp_n,\ s_n) =$ PickOneCube$(new_i \wedge \bar{\tau})$;
        $j = n - 1$;
        while $(j \geq 0)$
            $pred = \mathbf{PreImage}(\Delta_p,\ s_{j+1})$;
            $pred = pred \wedge new_j$;
            $(inp_j,\ s_j) =$ PickOneCube$(pred)$;
            $j = j - 1$;
        endwhile;
        return;
    endif;
    $To = \mathbf{Image}(\Delta_p,\ new_i)$;
    $i = i + 1$;
    $new_i\ =\ To\ -\ Reached$;
    $Reached\ =\ Reached\ \cup\ new_i$;
endwhile

Figure 2.14: Procedure to generate distinguishing sequence

to the initial state (s_0). The PickOneCube routine just picks one cube from the ON-set of the function given as the argument. Forward traversal is accomplished using the Image routine, which, as mentioned earlier, gives the set of states in the product machine that can be reached in one clock cycle **from** some state in the set of states given as argument to the routine. Backward traversal is accomplished using the PreImage routine, which gives the states **from which** some state in the set of states given as argument can be reached. In this case, the PreImage routine also gives the inputs causing the transitions.

This algorithm is the basic engine used by us to provide debugging hints to the designer. However, it requires specification of initial states which would be compatible if both machines were fault-free. This information is not always available to us and forces us to rely on some heuristics to apply the algorithm.

The intent is to either determine the incorrect transition or generate a distinguishing sequence as a debugging aid. We prove that such a distinguishing sequence must necessarily include the incorrect transition. We conjecture that determining the exact incorrect transition is NP-hard [62], even with *a priori* knowledge of compatible starting states. The methodology followed in attempting to locate design errors is outlined below. A design error may be either an incorrect output value or incorrect destination state for some transition. It is assumed that there is a single design error, and multiple errors can be handled by repeated application of these techniques.

1. If the state encodings in the specification and implementation are the same, compare the Boolean functions for the outputs and flip-flops to determine the incorrect output or next-state function. Correspondences

between data registers, *i.e.*, same partial encodings, also go a long way in reducing the state space and simplifying the debugging problem.

2. If the state encodings are different, verify the circuits for subsets of the outputs at a time in the hope of identifying an incorrect output function. If the error is in an output value and the state transitions are correct, this will give the set of compatible states for the correct outputs as well as identify the incorrect output function. Comparing the set of compatible state pairs with the set of 1-compatible state pairs for the incorrect output gives the state pair containing the state in which that output has been assigned a wrong value because that state pair is missing from the set of 1-compatible state pairs. A distinguishing sequence can also be generated using any state pair from the set of compatible state pairs as starting states.

3. If the specification is not contained in the implementation for any of the outputs, the error is probably in a state transition. In this case, knowledge of some states in the specification and the implementation which are supposed to be compatible, for example a reset state, can help us in generating a distinguishing sequence for the two machines. In the absence of such information, we generate the set of 1-compatible state pairs for the specification and implementation machines. This set is correct because the outputs are not erroneous, only the destination state is incorrect for some transition. Then all compatible state pairs must belong to this set of 1-compatible state pairs. We pick a specification state with the least number of 1-compatible implementation states and generate differentiating sequences for each such pair.

Obviously, these techniques are only aids to the designer and must be seen as available debugging tools to exploit any knowledge of the design the designer may have, such as state compatibility in the specification and implementation.

Proposition *Assuming a single error in the form of either an incorrect output or incorrect destination state for a single transition, a distinguishing sequence generated by the algorithm in Figure 2.14 necessarily traverses that incorrect transition.*

Proof: A distinguishing sequence generated by the algorithm in Figure 2.14 starts from states which are supposed to be compatible, *i.e.*, any input sequence, of any length, applied to the two machines in those states is supposed to produce the same output sequence for both machines. This implies that corresponding destination states of every transition encountered in the two machines must be compatible as well. An input sequence can produce different output sequences only if some states encountered by the sequence are k-distinguishable for some k. This, in turn, can only occur if one of the transitions encountered has an incorrect output value or goes to an incompatible, *i.e.*, wrong, destination state. □

If the incorrect destination state and the correct destination state are k-distinguishable, the sequence will terminate k inputs after the incorrect transition. In case of the error being an incorrect output value, the distinguishing sequence will terminate at the incorrect transition.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Chapter 3

Design Verification

The second paradigm for formal verification involves proving either that a design is correct with respect to a set of properties specified by the designer, or that the design possesses the functionality intended by the designer. The former scenario deals with proving that all the behavior of the design is **consistent** with that of the specified properties and does not violate any of the specified requirements. In general, the set of properties need not be a complete specification of the behavior of the system. For example, if the design being verified is a system of concurrently operating state machines implementing some complex protocol, the properties may deal with general features of the system such as absence of deadlock or guarantee of mutual exclusion from a shared resource. On the other hand, the latter scenario involves specifying properties which embody the intended functionality of the design. The verification then checks that the design **implements** the desired behavior. The properties are thus very specific to the design being verified. Any methodology for performing formal design verification must incorporate both the above capabilities in some form or other. This chapter presents an environment for the specification and verification of such properties which is based on similar concepts of state compatibility in finite state machines as used in implementation verification.

Given the basic techniques for verifying properties, the most important tool to handle the verification of realistic circuits is to control the complexity by abstracting the design into a reduced model. Unlike implementation verification, there exists a large semantic gap between the design and the specified properties and very few correspondences. This makes the problem of finding appropriate abstractions difficult, to say the least. There has been extensive research on using abstraction techniques to reduce the size of the model being verified such that the proof of correctness holds on the actual design if it holds on the reduced model [15,31]. One common technique is to introduce varying degrees of non-determinism in the model by leaving unspecified those details that are not pertinent to the property being verified. Another technique, called compositional verification [29,30], attempts to reduce the state space of a system of interacting component machines by simplifying each component with respect to the property being verified. In spite of the considerable progress being made, finding the appropriate and correct abstractions still requires considerable expertise.

On the plus side, there are a few characteristics of this problem which can be exploited to simplify the verification task. The model to be verified is usually at a sufficiently high level of abstraction to be understandable by the user, as opposed to a net list which can be extremely difficult to decipher. This makes it easier to specify the intended behavior in terms of the observable signals of the design and to debug the design. Also, since the design is usually described in some hardware description language (HDL), it is fairly easy to modify it in order to add or remove some behavior to aid in the verification.

However, since property checking remains the most troublesome formal technique to integrate into the design process, design verification by simulation, in spite of its limitations, is still the main resort of designers. Improving the quality of the test vectors so as to improve the functional coverage then assumes great importance. This chapter also presents our view of functional coverage as the fraction of control flow covered by the test sequences and presents the application of the concepts of implicit state enumeration, developed for formal verification, to the problem of evaluating the coverage in a sequential circuit for the sequences of test vectors.

3.1 Property Checking

Properties are usually classified into two categories, **safety** (invariant) properties and **liveness** properties. Safety properties specify conditions of the form *nothing bad will ever happen*, whereas liveness properties express behavior that specifies that *something good will eventually happen*. For example,

- no more than one output of the decoder will be high at a given time

is a safety property, and

- a request will always be eventually acknowledged

is a liveness property. Bounded liveness properties place a bound on the time interval over which the property is specified, unlike unbounded liveness properties. The following property

- an ALU instruction will give the correct result in the accumulator exactly three clocks after instruction issue

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

is an example of a bounded liveness property.

A third category of properties deals with the functional correctness of a design – for example, the property that a circuit should add two numbers correctly. This kind of property usually involves dealing with high level functions like addition and abstract data types like integers and so cannot be handled by techniques operating at the Boolean level. Theorem provers are able to prove such properties more easily, and their utility in our verification framework will be discussed in Chapter 4. Other properties dealing with functional correctness may specify the desired behavior of the design at the Boolean level. These can be cast as liveness properties and are discussed here in that perspective.

In verifying safety and liveness properties, we choose one of two specification styles. Properties may be verified by checking for either

- absence of bad behavior

or for

- existence of good behavior

In the earlier examples, the safety property could be checked by looking for a *bad* state where two or more outputs of the decoder are high. Similarly, the liveness property is best checked by looking for a *bad* state from which there exists some path on which a request is never acknowledged. On the other hand, the bounded liveness property can best be verified by checking that all paths from the instruction fetch state satisfy the specified *good* behavior. Choosing the proper specification style is important in expressing properties succinctly. The specification style also affects the subsequent verification in a significant way. Note that the circuit descriptions being verified are incompletely specified finite state machines. Consequently, we cannot make any assumptions about the unspecified behavior and must view it pessimistically. Casting the verification as a check for absence of *bad* behavior requires that we consider the unspecified behavior to mimic **any** *bad* behavior. On the other hand, casting the verification as a check for existence of *good* behavior requires that we **not** consider the unspecified behavior to fulfill any requirements of the specification. The next section illustrates several examples of properties specified as state machines.

3.1.1 Specification Framework

Different approaches to design verification employ different specification frameworks on the basis of three main criteria,

- application domain
- efficiency of the decision procedure to check satisfaction
- ease of expression.

Most automated, state-based approaches express the properties to be checked either as formulas in temporal logic or as automata.

Symbolic trajectory evaluation [22] uses a restricted form of linear temporal logic to express properties as bounded length sequences of circuit states for the verification of state-transformation systems like processors and memories. The temporal logic used has only the next-time operator, and verification is done efficiently using an extension of symbolic simulation. In this approach, the specifications explicitly involve the circuit state variables and hence require intimate knowledge and understanding of how the circuit state is implemented. Limited expressiveness of the temporal logic is also a hindrance in verifying certain types of liveness properties.

The DDL verifier of Fujita *et al.* [63] also takes formulas in linear temporal logic as specifications. It automatically translates the formulas to the form of state transitions, extended to express eventuality conditions, by recursively decomposing the formulas. Satisfiability of the formulas is checked by looking for infinite paths in the transition graph of the product machine of the specified property and the design. Such paths must necessarily include any transitions with eventuality conditions. The new version of their verification program [64] (part of a hierarchical specification and verification framework) uses an extended form of BDDs called Time Extended BDDs to manipulate expressions containing eventuality operators and also takes specifications in the form of timing diagrams. In order to handle practical designs, a technique of **filtering** the design description to drop irrelevant detail is employed.

CTL model checking [13] verifies that a finite state system satisfies properties expressed as formulas in branching-time temporal logic CTL. This is particularly well suited to the verification of systems with several interacting state machines implementing some complex protocol. While the decision procedure to check satisfaction of CTL formulas is very efficient, one drawback is that the temporal logic formulas can become exceedingly difficult to understand, and there is a danger of the designer misunderstanding what property has actually been verified. In general, it may be easier to express behavior dealing with sequences of states using state machines than it is to write the correct CTL formula.

The language containment approach [31] describes a property as an ω-automaton and verifies correctness by checking that the language of the system is contained in the language of the property. This verifies that all behaviors of the implementation are consistent with the specified behavior. Like CTL model checking, the language containment approach is also well suited to the verification of systems of interacting state machines and has been applied to the verification of large systems using a succession of verification and refinement steps. Unfortunately, certain types of properties involving existentiality are not expressible by ω-automata. For example, the property that there exists a path to a reset state from any design state cannot be so expressed. A unified approach to design verification, incorporating both language containment and CTL model checking to complement each other, is being implemented in the HSIS tool from UC Berkeley [65].

Labeled State Machines

The property or behavior to be verified is viewed as a sequence of inputs and consequent outputs and is encoded as a type of non-deterministic finite state machine, with a designated start state, called a **labeled** finite state machine. This state machine differs from a conventional FSM in that there are two types of states: **labeled states**, which may specify non-deterministic behavior, and non-labeled states, which always specify deterministic behavior. It is acceptable for an input to a labeled state to cause transitions to more than one next state. The significance of the labeled states lies in the fact that the input-output behavior of the labeled machine in its start state correctly models the property to be verified if **at least one** outgoing transition from labeled states which are reachable from the start state satisfies the behavior desired by the property. This is in contrast to non-labeled states, where **all** outgoing transitions must satisfy the desired behavior for the labeled machine to correctly model the property. In other words, at least one path through a labeled state and all paths through non-labeled states must satisfy the desired behavior for the labeled machine to correctly model the property.

For ease of understanding the specification of certain properties, labeled states are equipped with labeled outgoing transitions, where the inputs causing the labeled transition are unspecified or partially specified. This implies that the machine undertakes that transition for **some** (unspecified) value of the inputs. This is used when checking for the existence of transitions satisfying certain conditions without prior knowledge of the inputs that would induce them.

A labeled state machine may be partially specified. Unspecified outputs and unspecified next states are used to specify unconstrained behavior. A labeled FSM has a sink state to absorb all transitions with unspecified next states. On entering the sink state, the machine loops in that state to signify unconstrained behavior. Dummy output signals implementing the logic of some complex condition to be checked are also frequently added to the design and used to simplify the expression of the desired property.

The following examples illustrate the manner in which different properties may be specified as labeled state machines with some transitions having unspecified inputs. Both safety and liveness properties are equally important in proving the correctness of a design, and properties of both types can be specified in this framework. The verification algorithm attempts to find all states in the design satisfying the behavior of the specification in its start state. In the following examples, and usually for the systems being considered in this thesis, such as processors, micro-controllers and data paths, **p** signifies some event on the input signals, and **q** is a consequent event on the output signals. In general, **p** and **q** could be any events on the input or output signals. Labeled states are distinguished by concentric circles and labeled transitions have the **u** (for unknown) variable as input. All transitions with unspecified destination states end up in the sink state.

- *Property 1* Event **p** always causes event **q** to occur exactly 3 clocks later. Find *good* states in which event **p** occurs and such that event **q** occurs on the third clock on all paths from those states.

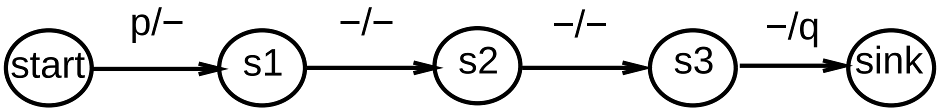


Figure 3.1: Example property 1.

- *Property 2* Event **p** always causes event **q** either 2 or 3 clocks later. Check for *bad* states in which **p** occurs and from which there exists a path such that **q** does not occur on either the second or third clock after **p**.

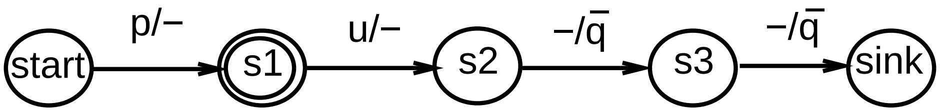


Figure 3.2: Example property 2.

- *Property 3* Event **p** causes event **q** to always eventually occur. Check for *bad* states in which **p** occurs and from which there exists some path along which **q** never occurs.

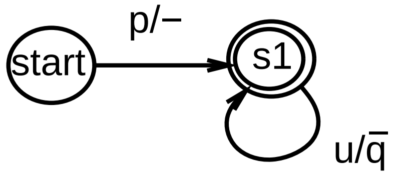


Figure 3.3: Example property 3.

- *Property 4* Event **q** never occurs.

  Check for *bad* states where **q** occurs on any transition from those states.

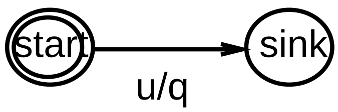


Figure 3.4: Example property 4.

- *Property 5* Is **q** ever true for 3 clocks after **p** occurs?

  Find *good* states in which **p** is true and there exists a path such that **q** is true for the next 3 clocks.

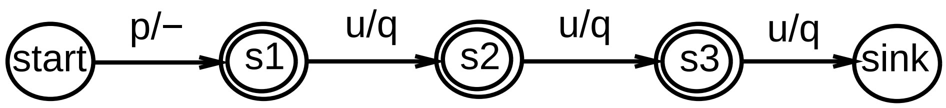


Figure 3.5: Example property 5.

The intent is to provide designers with an easy and intuitive specification format, along with the power to implicitly search the entire state space of the design. Frequently, the designer has already identified instances of *bad* behavior in the form of a sequence of inputs and consequent outputs. He or she can then check for the existence of such behavior in the design by modeling the sequence as a state machine. Of course, not all liveness or safety properties can be handled in the manner shown above. However, the framework is

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

expressive enough to handle all properties which may occur in practice during the verification of systems such as processors and data paths. The full expressiveness of this specification framework can be gauged only by using it for the verification of realistic examples. The specification and verification of a simple microprocessor using labeled state machines is discussed in Chapter 5.

3.1.2 Verification Algorithm

The verification algorithm must compute all states in the design such that the input-output behavior of the design from those states is compatible with that of the specification from its starting state. In other words, it must find all design states compatible with the specification starting state. However, the notion of compatibility as defined earlier applies to deterministic specification and implementation machines. That notion must be modified to account for specifications expressed as labeled state machines. To better understand the compatibility of states in a labeled state machine with states in an implementation machine, we introduce the artifact of **sequence sets** and the concept of **weak compatibility**.

Definition 21 *The set of all applicable input sequences SEQ to a labeled state machine C in state s is split into mutually disjoint sets of input sequences, called* **sequence sets** $\{seq_i^{(s)}\}$ $(\bigcap_i \{seq_i^{(s)}\} = \Phi)$, *such that each sequence set contains one and only one applicable input vector for each reachable labeled state but all applicable input vectors for each reachable non-labeled state, and* $SEQ = \bigcup_i \{seq_i^{(s)}\}$.

Definition 22 *State s_a in labeled state machine C_a is* **weakly k-compatible** *with state s_b in state machine C_b if s_a is compatible with s_b for all input sequences of length lesser than or equal to k in at least one sequence set $\{seq_i^{(s_a)}\}$.*

Consequently, two states are weakly k-compatible implies at least some or all successors are weakly $k-1$-compatible. Two states are weakly compatible if they are weakly k-compatible for all k.

Definition 23 *State s_a in labeled state machine C_a is* **weakly compatible** *with state s_b in state machine C_b if s_a is compatible with s_b for all input sequences in at least one sequence set $\{seq_i^{(s_a)}\}$.*

The verification algorithm seeks to find all design states weakly compatible with the start state of the labeled specification machine. While checking for weak compatibility with labeled states, the algorithm must look for design states with at least one outgoing transition with the same input-output behavior as some transition from the labeled state such that the ensuing behavior of the design satisfies that of the specification, *i.e.*, corresponding successive states are also weakly compatible. Such an algorithm to compute all pairs of weakly compatible states in the two circuits by finding successive sets of pairs of weakly k-compatible states is shown in Figure 3.6. We refer to state pairs containing labeled states as labeled state pairs and others as non-labeled pairs.

The algorithm first computes all pairs of compatible transitions, *i.e.*, transitions with the same outputs for the same inputs, in the two machines and initializes the set of pairs of weakly compatible states in the two machines to contain all state pairs (steps 1-2). The set of weakly 1-compatible pairs of

1. Find all pairs of compatible transitions τ;
2. $WC_0 = true$;
3. $k = 0$;
4. repeat
5. $\quad WC_{k+1} =$ {labeled state pairs in WC_k with at least one successor via compatible transitions in WC_k} $\cup$ {non-labeled state pairs in WC_k with all successors in WC_k}
6. $\quad k = k + 1$;
7. until $(WC_k = WC_{k-1})$;

Figure 3.6: Algorithm for computing pairs of weakly compatible states.

states is computed as the union of the set of those labeled state pairs with at least one pair of compatible outgoing transitions and the set of those non-labeled state pairs with all outgoing transitions compatible. The set of weakly 2-compatible state pairs is then computed as the union of the set of weakly 1-compatible labeled state pairs with at least one pair of outgoing compatible transitions ending up in states that are weakly 1-compatible and the set of non-labeled weakly 1-compatible state pairs with all transitions ending up in states that are weakly 1-compatible. In the same manner, the algorithm computes successive sets WC_k of weakly k-compatible state pairs (step 5) till a fixed-point is reached $WC_k = WC_{k-1}$. This is the set of all weakly compatible state pairs (step 7). The algorithm always terminates because the sequence $WC_0, WC_1, WC_2, ..$ is monotonically decreasing on a finite domain.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Computing pairs of compatible transitions is an important step in the verification process. If the transition in the implementation machine being considered has all outputs specified, it is compatible with all specification transitions that produce the same outputs, wherever defined for the specification, for the same inputs. If the implementation transition has undefined outputs, compatibility depends on the property in question. If the specification machine is expressing some desirable behavior, the implementation transition with undefined outputs is not compatible with any specification transitions. On the other hand, if the specification machine is expressing some undesirable behavior, the implementation transition is compatible with all specification transitions. The reasoning behind this is that the implementation must have *good* behavior explicitly implemented to satisfy a specification, whereas any behavior left unspecified in the implementation could possibly be implemented as *bad* behavior in later refinements.

Theorem 7 *The algorithm in Figure 3.6 correctly computes the set of all pairs of weakly compatible states in the labeled specification machine and the implementation machine.*

The proof of this theorem is similar to that presented for Theorem 1 in Section 2.3. Only the presence of labeled states requires modification of the accompanying lemma as follows.

Lemma 3 *If all weakly k-compatible states $\vec{q}_a$ and $\vec{q}_b$ of a labeled specification machine C_a and implementation machine C_b, respectively, are also weakly $k+1$-compatible, then they are weakly compatible.*

Proof: We will first prove that $\vec{q}_a$ and $\vec{q}_b$ are weakly $k+2$-compatible by contradiction. Assume they are not weakly $k+2$-compatible. However, we know $\vec{q}_a$ and $\vec{q}_b$ are at least weakly 1-compatible. We consider two cases.

Case 1: Specification state $\vec{q}_a$ is a non-labeled state. States $\vec{q}_a$ and $\vec{q}_b$ are not weakly $k+2$-compatible then implies that some successors are not weakly $k+1$-compatible. But $\vec{q}_a$ and $\vec{q}_b$ are weakly $k+1$-compatible by assumption and consequently all their successors must be weakly k-compatible. This again implies, by the statement of the lemma, that all successors are weakly $k+1$-compatible, which contradicts the previous statement.

Case 2: Specification state $\vec{q}_a$ is a labeled state. States $\vec{q}_a$ and $\vec{q}_b$ are not weakly $k+2$-compatible implies that there are no successor states via compatible transitions that are weakly $k+1$-compatible. But $\vec{q}_a$ and $\vec{q}_b$ are weakly $k+1$-compatible implies there is at least one pair of successor states via compatible transitions that is weakly k-compatible. By the statement of the lemma, that particular pair is also weakly $k+1$-compatible, which is again a contradiction.

Hence, our assumption is wrong and $\vec{q}_a$ and $\vec{q}_b$ are weakly $k+2$-compatible. Applying the same argument for $k+3$ and so on proves that $\vec{q}_a$ and $\vec{q}_b$ are weakly k-compatible for all k and hence weakly compatible. $\square$

As always, the verification algorithm is implemented using BDDs to manipulate sets of states and transitions. Again, consider a Mealy machine defined by the 5-tuple $(I, O, S, \Delta, \Lambda)$, where $I = \{0,1\}^m$ is the input space, $O = \{0,1\}^l$ is the output space, $S = \{0,1\}^n$ is the state space, $\Delta : I \times S \rightarrow S$ is the next-state functional vector, and $\Lambda : I \times S \rightarrow O$ is the output functional vector. Let C_a be the labeled partially specified machine representing the property and C_b be the partially specified machine representing the design. Let $U(\vec{q}_a)$ be the characteristic function representing the set U of labeled states in the specification machine, and η_{a_i} and η_{b_i} represent the transitions with defined outputs for the ith output in the specification and implementation respectively. We introduce dummy inputs in the specification to translate non-deterministic behavior to deterministic behavior.

We first compute the set of pairs of compatible transitions in the two machines. As discussed earlier, this set may either include (Equation 3.1) or exclude (Equation 3.2) transitions in the design with undefined outputs, given by the negation of the predicate η_{b_i} for the ith output. On the other hand, transitions in the specification with undefined outputs, $\neg\eta_{a_i}$ for output i, match all corresponding transitions in the design. The predicate τ given below encodes this set of transition pairs,

$$\tau(\vec{x}, \vec{q}_a, \vec{q}_b) \;=\; \bigwedge_{i=1}^{l} \left[\; (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b)) \;\vee\; \neg\eta_{b_i}(\vec{x}, \vec{q}_b) \right. \\ \left. \vee\; \neg\eta_{a_i}(\vec{x}, \vec{q}_a) \;\right] \qquad (3.1)$$

or

$$\tau(\vec{x}, \vec{q}_a, \vec{q}_b) \;=\; \bigwedge_{i=1}^{l} \left[\; (\; (\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b)) \;\wedge\; \eta_{b_i}(\vec{x}, \vec{q}_b)\;) \right. \\ \left. \vee\; \neg\eta_{a_i}(\vec{x}, \vec{q}_a) \;\right] \qquad (3.2)$$

We now wish to compute pairs of weakly compatible states in the specification and the design using the predicate τ. To find the set of weakly $k+1$-compatible state pairs WC_{k+1} from the set of weakly k-compatible pairs WC_k, we find the **inverse image** and **pre-image** (Equation 2.2) of the set WC_k, *i.e.*, the pairs of states which must end up in weakly k-compatible states in one transition , and the pairs of states which could end up in weakly k-compatible states in one transition.

The inverse image of WC_k can be given by $\forall \vec{x} : WC_k(\vec{Q}_a, \vec{Q}_b)$, and the pre-image can be given by $\exists \vec{x} : WC_k(\vec{Q}_a, \vec{Q}_b)$. We must further ensure that the transitions to WC_k from its pre-image are compatible, *i.e.*, satisfy the predicate τ. The set of $k+1$-compatible state pairs is then given by the union of these two image sets, restricted to the set of weakly k-compatible pairs. The characteristic functions of these sets of state pairs are as follows,

$$WC_0 \;=\; 1$$

$$\begin{aligned} WC_{k+1}(\vec{q}_a, \vec{q}_b) \;=\; WC_k(\vec{q}_a, \vec{q}_b) \bigwedge [\; &(\neg U(\vec{q}_a) \;\wedge\; \forall \vec{x} : WC_k(\vec{Q}_a, \vec{Q}_b)) \\ \bigvee\; &(U(\vec{q}_a) \;\wedge\; \exists \vec{x} : (\tau \wedge WC_k(\vec{Q}_a, \vec{Q}_b)))\;] \end{aligned} \quad (3.3)$$

The set of weakly compatible state pairs $\mathcal{WC}$ is obtained when the fixed-point is reached.

$$\mathcal{WC}(\vec{q}_a, \vec{q}_b) \;=\; WC_k(\vec{q}_a, \vec{q}_b) \qquad if\ WC_{k+1} = WC_k \quad (3.4)$$

The set of design states compatible with the start state of the specification machine $\vec{q}_{a_{start}}$ has characteristic function given by

$$\mathcal{WC}(\vec{q}_a, \vec{q}_b)_{\vec{q}_a = \vec{q}_{a_{start}}}$$

Design states not covering any specification state are given by $\neg\exists \vec{q}_a : \mathcal{WC}(\vec{q}_a, \vec{q}_b)$.

As an example, consider the finite state machine shown in Figure 3.7(a) with one input and one output. For the sake of simplicity, the machine is fully

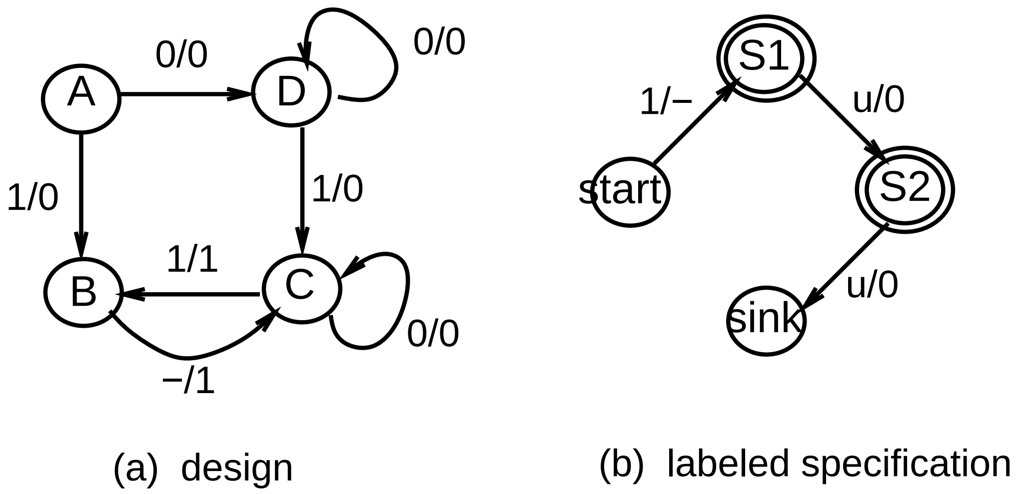


Figure 3.7: A verification example.

specified. We wish to check that whenever the input goes high, the output will go high within two clock cycles. We perform this verification by checking for occurrence of *bad* behavior which conflicts with the specified behavior: *is there any sequence of states such that the output stays low for two clock cycles after the input has gone high*? The labeled specification machine is shown in Figure 3.7(b). $S1$ and $S2$ are the labeled states in the specification machine and the outgoing transitions are labeled with unspecified inputs.

We now consider a specification state to be weakly compatible with a design state if the sequence of outputs produced by the design is identical to that produced by the specification, starting in their respective states, for all input sequences in at least one of the sequence sets for that specification state. As an example, the four sequence sets for the start state of the specification in Figure 3.7(b) for applicable input sequences of length three are $\{100\}$, $\{101\}$, $\{110\}$, $\{111\}$. The verification program indicates that design state D is compatible with the specification start state and points to the existence of a *bad* sequence in the design starting at state D.

In the case when we are required to check for existence of *good* behavior, the verification program computes all states that satisfy the specified behavior, *i.e.* are weakly compatible with the specification start state. It is then a simple matter to check whether a given set of design states, usually the most general starting states, belong to this set.

3.1.3 Error Diagnosis

Providing the designer with information pointing to the cause of errors in the design is just as important as the verification itself. If the verification program provides useful debugging information, the design can be corrected easily and subjected to verification again in a short time. Generating an error trace is one way of aiding debugging. Error traces fall into two categories, **counterexamples** and **witnesses**. Counterexamples are execution traces that point to the cause of a mismatch between the behaviors of the implementation and the specification. On the other hand, witnesses are execution traces that excite paths in the implementation along which the specified property holds, *i.e.*, the two behaviors match.

Generation of Counterexamples

In the absence of labeled states in the specification machine, a distinguishing sequence for the specification start state and any given design state can be found by building the transition relation for the product machine and traversing the product state space [61]. The procedure to implement this algorithm is given in Figure 2.14 in Section 2.9. However, if the specification has labeled states, the procedure in Figure 2.14 needs to be modified slightly. This modified procedure is shown in Figure 3.8. In case of labeled state pairs, a discrepancy in the outputs of the specification and implementation (step 6) is noted only if there exists not even one compatible outgoing transition pair, unlike non-labeled state pairs where a discrepancy is noted if all outgoing transitions are not compatible.

$$discrp \;=\; new_i \wedge \neg\tau \wedge \neg U \;\bigvee\; \forall \vec{x} : new_i \wedge \neg\tau \wedge U$$

The Image routine to find the image of a set of states new_i in the product machine (step 17) is also modified for labeled states (U) to include just those new states (To) which are reached in one step only through compatible transitions (τ). This modified routine **LImage** has the following steps. Here $\vec{q}$ stands for a state in the product machine, $\vec{q} = \langle \vec{q}_a, \vec{q}_b \rangle$.

- $T_1(\vec{x}, \vec{q}, \vec{Q}) = \; T(\vec{x}, \vec{q}, \vec{Q}) \wedge new_i(\vec{q})$
- $T_2(\vec{x}, \vec{q}, \vec{Q}) = \; T_1(\vec{x}, \vec{q}, \vec{Q}) \wedge \neg U(\vec{q})$
- $T_3(\vec{x}, \vec{q}, \vec{Q}) = \; T_1(\vec{x}, \vec{q}, \vec{Q}) \wedge U(\vec{q}) \wedge \tau(\vec{x}, \vec{q})$
- $To(\vec{Q}) = \exists \vec{x}, \exists \vec{q} : \; (\; T_2(\vec{x}, \vec{q}, \vec{Q}) \vee T_3(\vec{x}, \vec{q}, \vec{Q}) \;)$
- $To(\vec{q}) = \mathrm{Compose}(To(\vec{Q}), \vec{q})$

The sequence generated by this procedure is the shortest distinguishing sequence because the algorithm performs an implicit breadth-first traversal of the state transition graph.

```
new_0 = s_0 = Reached = initial state(s);
Build Transition Relation(T);
Build Compatible Transitions(τ);
i = 0;
while (new_i ≠ 0)
    if (discrp ≠ 0)
        (inp_n, s_n) = PickOneCube(new_i ∧ ¬τ);
        j = n − 1;
        while (j ≥ 0)
            pred = PreImage(T, s_{j+1});
            pred = pred ∧ new_j;
            (inp_j, s_j) = PickOneCube(pred);
            j = j − 1;
        endwhile;
        return;
    endif;
    To = LImage(T, new_i);
    i = i + 1;
    new_i = To − Reached;
    Reached = Reached ∪ new_i;
endwhile
```

Figure 3.8: Procedure to generate counterexamples.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Generation of Witnesses

In cases where the implementation satisfies the specified property, we may wish to find an input sequence that contributes to compatibility rather than distinguishes between states, in other words, a witness. This is particularly true for specification machines involving labeled states. Finding a witness in the case of a specification machine without labeled states is trivial because all weakly compatible states are then truly compatible and all input sequences produce the same outputs for the two machines. On the other hand, in case of specifications with labeled states, an implementation satisfies the specification only for certain input sequences (at least one sequence set). The procedure for generating witnesses in such cases is described in Figure 3.9.

In generating a witness, the pre-computed set of weakly compatible state pairs $\mathcal{WC}$ plays an important role. The algorithm generates a witness by traversing the state space of the product machine from the initial states. In contrast to generation of counterexamples, the sequence generation is done during the forward traversal itself and there is no need to perform backward traversal. The key idea is that the traversal is done by always picking new states, via compatible transitions, that are weakly compatible (steps 9-13). This guarantees that the sequence generated produces identical outputs for the specification and the design. The PickOne routine ensures that the *sink* state is not picked as the next state in the specification machine as long as there are other alternatives (step 14). The predicate *Picked* keeps track of the transitions that are added to the witness sequence so that the algorithm realizes when a transition that has already been picked is chosen again (step 15-18). Finally the inputs and the destination states are extracted from the picked transitions

s_0 = Initial State of Product Machine;
$Picked = 0$;
Build Transition Relation(T);
Build Compatible Transitions(τ);
Build Weakly Compatible State Pairs($\mathcal{WC}$);
$\mathcal{WC}(\vec{Q})$=Compose($\mathcal{WC}(\vec{q}),\ \vec{Q}$);
$i = 0$;
repeat
    $pred = s_i(\vec{q}) \wedge T(\vec{x},\ \vec{q},\ \vec{Q})$;
    if (s_i contains a **labeled** state)
        $pred = pred \wedge \tau(\vec{x},\ \vec{q})$;
        $pred = pred \wedge \mathcal{WC}(\vec{Q})$;
    endif;
    $pick$ = PickOne($pred$);
    if ($pick \subset Picked$)
        break;
    endif;
    $Picked = Picked\ \cup\ pick$;
    $(inp_i,\ s_{i+1}) = \exists \vec{q} : pick(\vec{x},\ \vec{q},\ \vec{Q})$;
    $i = i + 1$;
    $s_i(\vec{q})$ = Compose($s_i(\vec{Q}),\ \vec{q}$);
until (s_i contains $sink$ state);

Figure 3.9: Procedure for witness generation.

(step 19). The algorithm terminates either when the sink state is picked as the next state in the specification machine, since all subsequent behavior of the implementation is now allowed, or when a previously picked transition is chosen again, which implies that the execution path or witness has fallen into a loop.

Consider the example of Figure 3.7 again. The verification program indicates that state D in the implementation is weakly compatible with the specification start state. Providing these as the initial states to the witness generation routine of Figure 3.9 generates the following output,

Inputs		Next state specification	Next state implementation
1	$\longrightarrow$	S1	C
0	$\longrightarrow$	S2	C
0	$\longrightarrow$	sink	C

3.1.4 Specification Input Format

The difficulty of expressing specifications succinctly and correctly is one aspect of verification which is often overlooked in academic tools. Most verification tools accept specifications in some form of temporal logic, automata, timing diagrams or some input language specific to that tool. A drawback of expressing specifications in some temporal logic is the complexity of writing them as formulas in that logic. It can sometimes be very difficult to understand the formulas and thus easy to misunderstand what properties have actually been verified. Timing diagrams are an option which is very familiar to designers. So are state diagrams or programs in a commercial hardware description language.

In keeping with our stated intention of providing a framework more intuitive to designers, we accept descriptions of the specification machines in a small subset of VHDL. This also enables designers to simulate the description for sample input vectors and gain some measure of confidence in the correctness of their specification itself. The specification format is a single VHDL process with a single wait statement depending on the rising edge of the clock and a single case statement depending on the state. The STATE signal and the LABEL signal are the only signals of enumerated type. STATE assumes values of all symbolic states in the machine, whereas LABEL gives only the labeled states. The specification can contain signals which are bits or bit vectors, with associated arithmetic operations. The only values signals can assume are '0','1' and 'X' (don't care).

As an example, Figure 3.14 shows the state machine for the labeled specification machine in Figure 3.7(b). INP is the input and OUTP is the single output. LABEL gives the labeled states $S1$ and $S2$. All transitions with undefined next states are directed to the sink state. On entering the sink state, the machine loops in that state forever to imply unconstrained behavior. The specification state machine can be easily extracted from such VHDL programs and input to the verifier to build the BDDs and perform the verification.

3.2 Improving Simulation Coverage

While much progress has been made in automating the verification process and using formal verification tools at the lower levels of the design hierarchy, simulation is still the primary means of checking the correctness of a high level or behavioral description. The design is simulated for all the vectors in a func-

```
architecture BEHAVIOR of SPEC is
 type STATE is (START, S1, S2, SINK);
 type LABEL is (S1, S2);
begin
process
begin
 wait until CLK = '1' and not(CLK'stable);
 case STATE is
   when START =>
     if (INP = '1') then
       STATE <= S1;
       OUTP <= 'X';
     else
       STATE <= SINK;
       OUTP <= 'X';
     end if;
   when S1 =>
     STATE <= S2;
     OUTP <= '0';
   when S2 =>
     STATE <= SINK;
     OUTP <= '0';
   when SINK =>
     STATE <= SINK;
     OUTP <= 'X';
 end case;
end process;
end BEHAVIOR;
```

Figure 3.10: Example specification machine in VHDL.

tional test suite and the simulation output is checked to determine whether the design is behaving as specified. The designer depends on information provided by the functional specification to generate the required tests and attempts to check that the design implements all specified behavior correctly.

It is common practice to retain the generated functional vectors for testing of the low level implementation. These tests may be used to validate the correct behavior of the implementation in the absence of a formal verification methodology. They are also sometimes used as manufacturing tests. Given the complexity of VLSI designs, it is a demanding and expensive task to perform automatic test pattern generation on large net lists. Using the existing functional tests and performing fault simulation and fault dropping results in a smaller fault set which can be targeted for test pattern generation [66]. A comprehensive functional test suite can give a stuck-at fault coverage as high as 70%. Thus, the importance of a good functional test suite cannot be overemphasized in today's design environment.

The term "good" functional test suite, meaning a test suite providing high functional coverage, immediately brings up the question of how we define and evaluate the coverage of a set of input stimuli. For the sake of clarity and to leave no room for misunderstanding, we now differentiate functional coverage from fault coverage. Fault coverage deals with the physical faults which may occur in the system and which are modeled using well-understood models such as the stuck-at and bridging fault models. Fault coverage of a test sequence is then defined as the fraction of all modeled faults which are detected by that test sequence. In contrast, no such standard fault model exists for functional faults. Ideally, functional coverage metrics should quantify the fraction of specified behavior of the design that has been exercised by the input stimuli. However, since the functional specification is in an abstract form, it is easier to determine the fraction of behavior of the circuit description covered by the input sequence. Thus, a high coverage will ensure correctness of the modeled behavior, but not that **all** specified behavior is included in the model. This latter problem is analogous to determining completeness of a set of properties in formal design verification and remains an unsolved problem to date. There is one more significant difference between design verification and physical testing. Unlike testing for physical faults, which is done on the actual chip, design verification is undertaken on a software model usually written in some hardware description language (HDL). Consequently, all signals and registers in the design are observable during simulation, unlike testing.

The ideal metric for estimating the functional coverage of a test sequence would be the number of possible execution paths exercised by that input sequence. However, finding all possible paths has exponential complexity and attempting to exercise all of them is out of the question. The next best measure is the fraction of reachable states or possible transitions that have been exercised. This is the approach adopted in this thesis. However, most designs have a data path of substantial size which contributes to a huge state space. Exercising all possible transitions in such a design is neither feasible nor desirable due to the enormous amount of CPU cycles it would consume. Attempting to cover the control part of the state space would exercise sufficient behavior to uncover most design errors of significance. The difficulty lies in differentiating control from data and extracting the control part from the design. Most approaches to this problem are closely tied to the input description language and depend on the syntactic style. This is a serious drawback. In this thesis, the extracted control flow machine (ECFM) model, which embodies the flow of control in a sequential circuit, is presented as a solution. This model can be extracted automatically from the circuit description. Significantly, the model extraction is independent of the syntactic style of the HDL program and depends only on the identification of data registers in the circuit. This extracted control flow machine is utilized to determine the amount of control behavior exercised by a set of input stimuli. In cases where the control machine is defined as a separate unit, computing the fraction of covered transitions using the techniques presented here is a simple task.

3.2.1 Related Work

Attempts at estimating the quality of functional test suites have followed several approaches. One approach defines a functional fault model, in the form of **design errors** [57,67], and attempts to gauge test quality by looking at faulty circuit behavior, *i.e.*, behavior of the circuit with a fault inserted, and checking how many of the design errors are detected by the functional vectors. Thus, it is closely modeled after the stuck-at fault coverage metric. The coverage is then determined by computing the fraction of all possible such errors in the net list detected by the tests. While this technique is very effective at the gate level or at lower levels to uncover common designer mistakes, it is primarily a localized structural approach and gives no indication of the extent to which the behavior of the design has been exercised.

Another approach to estimating functional test vector coverage looks at the fault-free behavior of the circuit and does not deal with a fault model.

For example, the number of statements in the HDL program that are executed during simulation are taken as a measure of the coverage. In [68], an extended finite state machine (EFSM) model, which is a state machine with a collapsed data path and far fewer states than the original circuit, is extracted from the behavioral description EFSM. Exhaustive traversal of this machine ensures that all statements in the original behavioral program are executed. However, EFSM extraction depends heavily on a certain description style of the behavioral HDL program. Furthermore, checking execution of every statement in the HDL program is not a sufficient measure of the functional coverage. The coverage given by this metric is drastically affected by the description style. For example, coverage of a (for instance, VHDL) program written in a pure behavioral manner using a series of nested **ifs** will differ drastically from that of the same program written in a dataflow style using concurrent assignment statements.

Some ad-hoc approaches to evaluating coverage just check that the state elements in the circuit assume all possible values during simulation to ensure all states are visited. But state coverage does not reflect transition coverage, which is a more accurate measure since it covers the different possible behaviors from each state. A state transition monitor was used in the functional verification of processors in a parallel supercomputer in [69]. The details of its efficacy are not provided, but the state machines tested have at most 110 states and 140 transitions.

3.2.2 The Extracted Control Flow Machine Model

The state space of a practical circuit usually includes a huge data space which results in a state explosion when manipulating a model of the circuit. A succinct description of the control flow in a sequential circuit is often required in several applications including test generation and design verification. Extracting this information from a circuit description is not an easy task at any level of detail, including the behavioral level. For example, consider a simple **if** conditional statement in VHDL,

if (sel) then outp <= a ; else outp <= b ; end if ; and equivalent concurrent assignment statements,

outp(i) <= (sel AND a(i)) OR (NOT(sel) AND b(i)) ;

Whereas the control machine extracted from the first program will have two outputs, one to a multiplexer and the other a load signal, the machine extracted from the second description will have only one output load signal. Thus, an extraction method that depends on the syntactic style of the HDL program is undesirable. Note that synthesis of the complete state machine from the VHDL program is a trivial task given the state-of-the-art in automatic synthesis tools.

Rather than depend on the HDL semantics, we seek to extract a model of the control flow in a sequential circuit by looking at the mathematical model of a finite state machine representing the complete circuit. The difficulty in identifying the control circuitry often lies in defining the interface of the control unit with the rest of the circuit, not in differentiating the control registers from registers holding pure data. In the approach presented here, the designer

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

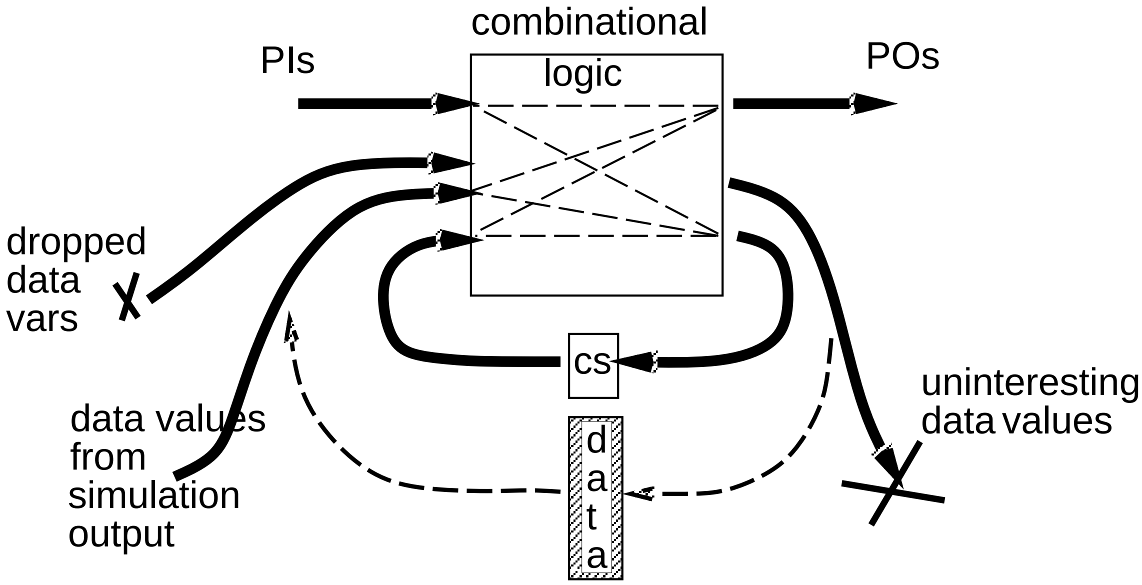


Figure 3.11: Extracting the ECFM from the FSM representing a sequential circuit.

chooses which registers he or she wishes to consider as contributing to the control state space and which make up the data. The key is that we are not interested in the data part of the state space, *i.e.*, in the different values that the data registers hold during the simulation. Consequently, we drop the data registers from the circuit and determine the **support set** of the primary outputs and control state variables to find the new extended set of circuit inputs. The support set of an output or state variables is just the set of inputs and state variables on which that output or state variable depends. This is more easily explained with the help of a figure.

Consider the finite state machine in Figure 3.11. The shaded registers correspond to data registers. They are dropped from the circuit since we are not interested in the data part of the state space. We are also not interested in the actual data values, so we drop the inputs to these data registers from

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

consideration (marked by the big cross in Figure 3.11). We now determine the support set (shown by dotted lines) of the primary outputs and control state in terms of the primary inputs, control state variables and data variables. This forms the new set of primary inputs and may include some data variables also. The rest of the data variables are dropped (marked by a small cross in Figure 3.11). This modified state machine is the extracted control flow machine (ECFM). Its state space represents the **control space** of the actual circuit and it retains the primary outputs of the actual circuit. Every state transition in the ECFM involves only assignment to primary outputs, and no update of data. However, note that the transitions now depend on an extended input set which includes data variables. The state space of the ECFM is significantly smaller than that of the actual circuit while still exhibiting the **same control flow**.

Example: Consider the simple state transition graph shown in Figure 3.12. The circuit has one input (**start**), one output (**done**), four 16-bit registers (**r**, **m**, **regA** and **regB**) and a flag flip-flop (**flag-bit**). The behavior described basically involves swapping the values in registers **regA** and **regB** if the value in **regA** is less than the value in **regB**. This sort of behavior is common where registers **r** and **m** can be the ALU registers and **regA** and **regB** are part of a general-purpose register file. The complete state space of the finite state machine for this example is of the order of 10^{20} states. Manipulating a state space of this magnitude is no easy task even for implicit state enumeration methods. However, most of the state space is due to the data registers (**r**, **m**, **regA** and **regB**). It would suffice to cover the control flow to obtain high functional coverage rather than look at the whole state space.

The ECFM for the circuit under consideration is shown in Figure 3.13.

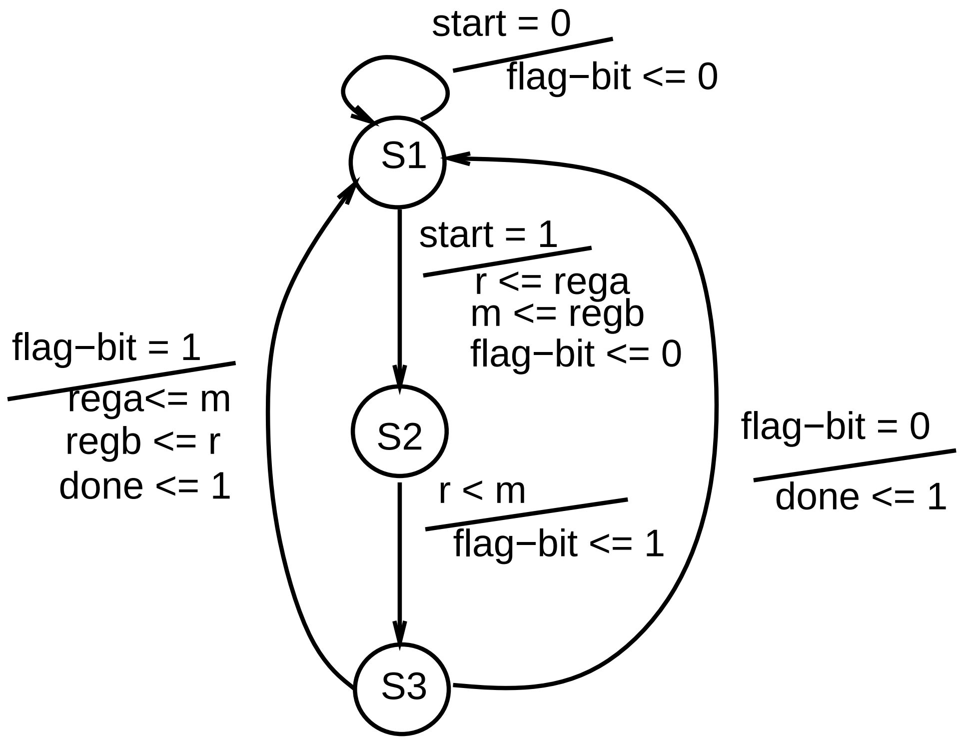


Figure 3.12: State transition graph for example circuit.

The flag-bit is considered part of control state (as shown in the circles in the figure), whereas the registers **r**, **m**, **regA** and **regB** are pure data. As seen from the figure, control flow depends on **r** and **m** and they are considered as primary inputs. Registers **regA** and **regB** are dropped from the circuit. This ECFM has just five states but actually embodies the complete flow of control in the actual circuit. Recall that the actual circuit has 67 flip-flops and 10^{20} states. The number of primary inputs to the ECFM is 33 as compared to 1 for the original circuit. However, this increase is overshadowed by the decrease in state space. The net effect is a dramatic reduction in the size of the state space. Furthermore, scaling the data path in the actual circuit to 32 bits does not result in any increase in the number of ECFM states which demonstrates

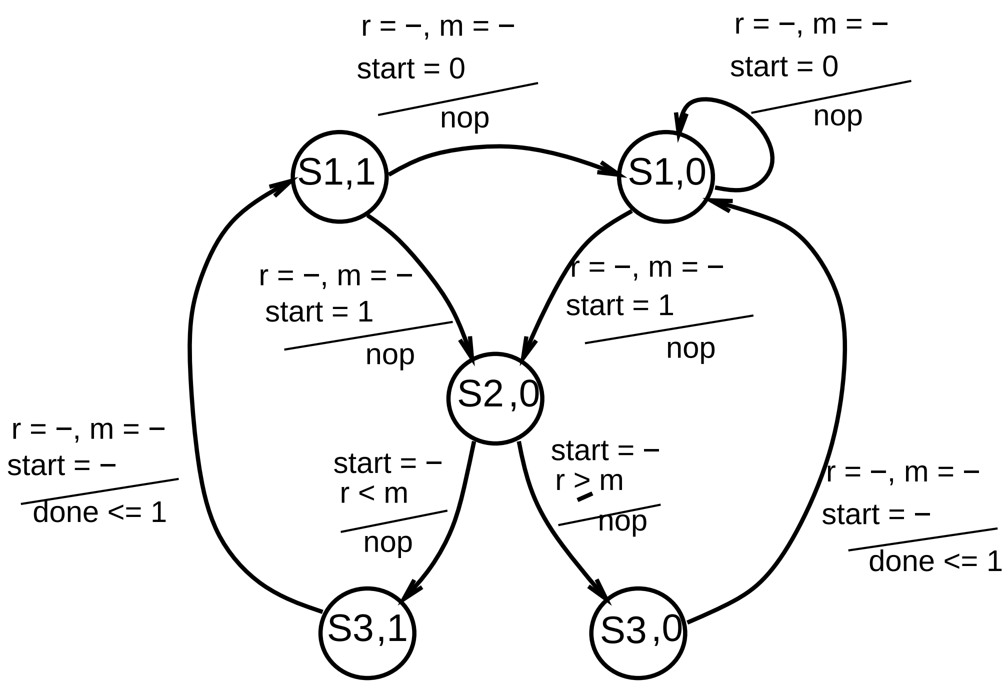


Figure 3.13: Extracted control flow machine for example circuit.

the scalability of this model with circuit size.

3.2.3 Computing Functional Coverage

We now describe how the ECFM can be used to derive a pragmatic estimate of the functional coverage provided by a set of input sequences. This capability is critical, since it is obvious that design errors cannot be detected if the simulation never enters the state or exercises the transition on which the erroneous behavior is displayed. Note that we assume the tests have already been generated, *i.e.*, we are not doing functional test generation. In line with the arguments stated earlier, we define functional coverage as the **coverage of the control flow** in the circuit.

Coverage Metrics

In order to compute a quantitative measure that reflects the quality of a test suite, we use two metrics, a state coverage metric (SCM) and a more accurate transition coverage metric (TCM).

Definition 24

$$SCM = \frac{Number\ of\ states\ visited}{Total\ number\ of\ reachable\ states} \tag{3.5}$$

Definition 25

$$TCM = \frac{Number\ of\ transitions\ traversed}{Total\ number\ of\ reachable\ transitions} \tag{3.6}$$

Obviously, TCM is the more comprehensive metric of the two. It is possible to have 100% state coverage without having 100% transition coverage. However, there is virtually no additional penalty for computing SCM. 100% SCM can be the first target. After covering all the control states, covering the remaining transitions will be an easier task. SCM can give the designer hints that are easier to understand than TCM as to how he or she can improve the quality of the tests.

Computing the set of reachable states for large circuits is not feasible even when using implicit enumeration methods. More importantly, it is not the designer's goal to exercise all transitions of the state machine as this will be prohibitively expensive for circuits with a data path of even moderate size. Rather, they are interested in whether all the control behavior of the circuit has been fully tested. To achieve this, the coverage metrics are now computed on the ECFM model and reflect the amount of control behavior exercised during simulation. It should be noted that computing the reachable states in the

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

ECFM does not directly correspond to the control states that may be reachable in the actual machine. Rather, it is an over-estimation of the reachable control state space because some data variables of the original circuit are primary inputs in the ECFM and may assume values not possible in the actual circuit. However, it is intuitively a close approximation because in general data assumes any value.

Using the ECFM Model

Most current approaches to functional coverage of verification tests depend on metrics that reflect the fraction of statements in the behavioral HDL program exercised during the simulation. However, though attempting to target control flow, this coverage figure varies greatly with the description style and can be misleading. As a simple example, again consider the circuit in Figure 3.12. A VHDL description of this state machine, written in a popular style, is shown in Figure 3.14. For design verification, it would suffice to cover the control flow to test the functional behavior rather than look at the whole state space. So we would wish to take some random values for the data in registers **regA** and **regB** and check that they are indeed swapped if **regA** is less than **regB** and left untouched otherwise. Setting **regA=10** and **regB=11** and feeding the input vector 01XX will exercise all the statements in the VHDL program and should give 100% functional coverage. However, as is obvious, we have not tested the case where **regA** is not lesser than **regB** and no swapping takes place. Thus, even in such a simple machine, HDL coverage is seen to be an incorrect measure.

On the other hand, the ECFM extracted from the underlying state machine is independent of the description style of the HDL program and encodes

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.


```
architecture BEHAVIOR of EXAMPLE1 is
 type STATE is (S1, S2, S3);
 signal CS : STATE;
 signal r, m, regA, regB : bit-vector(15 downto 0);
begin
process
begin
 wait until CLK = '1' and not(CLK'stable);
 case CS is
   when S1 =>
     flag-bit <= '0';
     if (START = '1') then
       CS <= S2;
       r <= regA;
       m <= regB;
     else
       CS <= S1;
     end if;
   when S2 =>
     CS <= S3;
     if (r < m) then
       flag-bit <= '1';
     end if;
   when S3 =>
     CS <= S1;
     if (flag-bit = '1') then
       regA <= m;
       regB <= r;
     end if;
     DONE <= '1';
 end case;
end process;
end BEHAVIOR;
```

Figure 3.14: VHDL description of example.

the complete control flow in the original circuit. The values for data variables which are primary inputs to the ECFM are determined from the simulation runs of the actual circuit during design verification and fed to the ECFM along with the input vectors to compute the control coverage. Pointers to the part of the state space that is not covered by the given tests are provided in the form of uncovered ECFM states or transitions. This is of paramount importance because it can greatly facilitate additions to the test suite to achieve better coverage.

Consider the ECFM for the same circuit shown in Figure 3.13. The values of the data variables **r** and **m**, which are primary inputs to the ECFM, can easily be determined from the simulation runs of the actual circuit performed during design verification, and fed to the ECFM along with the input vectors while computing the coverage. The test sequence above does not enter the ECFM state **(S3,0)** signifying incomplete control coverage. Ensuring that every transition in the ECFM is exercised during verification ensures that both cases of swapping registers **regA** and **regB**, and leaving them untouched are covered.

Though the number of states in the ECFM is only five, each state has 2^{33} outgoing transitions for the 33 inputs. Though it takes only a few input vectors to exercise the full control flow, the transition coverage metric for such a test set, as computed above, will be a microscopic number and will not reflect the actual functional coverage. To avoid this, transitions in the ECFM are grouped into sets of **equivalent transitions**.

Grouping Equivalent Transitions

Transitions that have the same effect on the control flow should be grouped

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

together, so that if one of them is exercised by the given input sequence, all of the transitions in that group are considered to be exercised.

Definition 26 *Two transitions of the ECFM have the same effect on the control flow if they have the same current state, the same next state and they modify the output behavior of the ECFM in the same manner. Such transitions are called* **equivalent transitions**.

Note that this definition of control flow takes primary outputs into account. Two transitions with different assignments to primary outputs, though with same current and next states, signify different functional behavior and must both be traversed to achieve complete functional coverage. One drawback of the above view is that two equivalent transitions may have different assignments to those data values which have been dropped from the ECFM and this difference is overlooked. However, the user has the ability to force transitions to be inequivalent simply by assigning different values to a dummy primary output on those transitions.

Example: For the circuit in Figure 3.13, several transitions can be grouped for each state pair. This grouping is shown in Figure 3.15. For instance, one group is all transitions between state (**S1,1**) and state (**S2,0**) with **start=1** since that transition is independent of the other inputs. Thus, traversing one transition effectively covers 2^{32} transitions. Building groups for every state pair gives us eight groups as shown in the figure. Traversing just one transition from each group can give 100% functional coverage, which will also be reflected in the transition coverage metric.

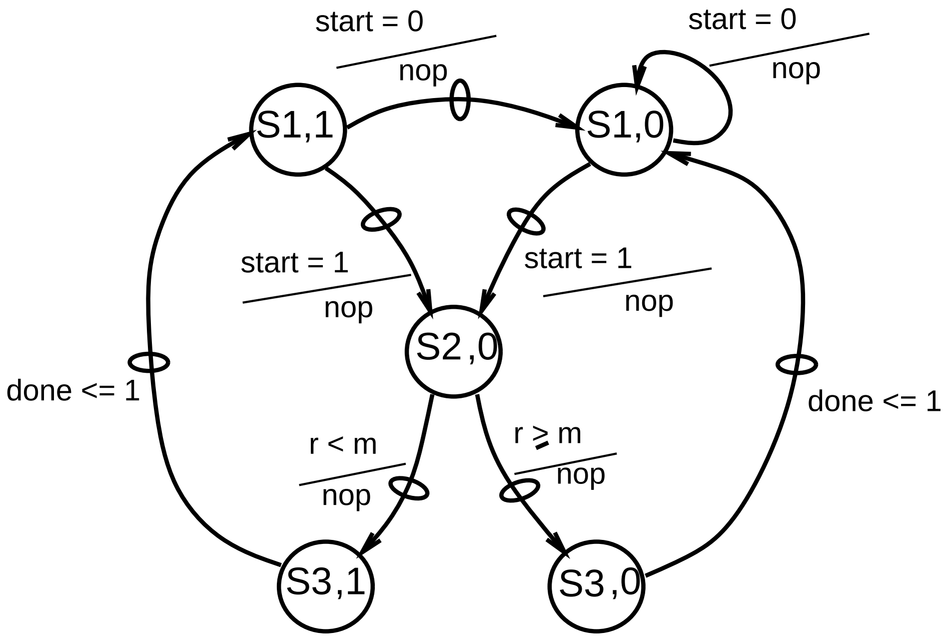


Figure 3.15: ECFM for example showing grouping of transitions.

Thus, the TCM now gives a more pragmatic coverage measure for the amount of control flow exercised by a set of input stimuli.

3.2.4 Algorithms

Again, consider the sequential circuit as a Mealy–type FSM defined by the tuple (I, O, S, s^0, Δ, Λ), where I is the input alphabet, O is the output alphabet, S is the finite set of states of the machine, s^0 is the set of start states, Δ is the next state functional vector ($\Delta : S \times I \rightarrow S$) and Λ is the output functional vector ($\Lambda : S \times I \rightarrow O$). Note that now we have a set of initial states specified, because simulation is performed with some flip-flops, usually the control flip-flops, initialized (except when checking reset behavior).

Algorithm For Finding Reachable States

As mentioned earlier, given an FSM with next state function $\Delta(\vec{x}, \vec{q})$, the transition relation of the machine can be written as:

$$T(\vec{x}, \vec{q}, \vec{Q}) = \prod_{i=1}^{l} Q_i \odot \delta_i(\vec{x}, \vec{q}) \tag{3.7}$$

Reachability analysis is performed as a breadth–first traversal of the given FSM. The basic operation for this type of traversal is computation of the **Image** of a set of states, which basically gives the set of states that can be reached in one transition from some state in that set (Equation 2.2). The Image calculation of a set of states $C(\vec{q})$ is performed in three steps:

- $T_1(\vec{x}, \vec{q}, \vec{Q}) = T(\vec{x}, \vec{q}, \vec{Q}) \wedge C(\vec{q})$
- $T_2(\vec{Q}) = \exists\, \vec{x}, \vec{q}\;\; T_1(\vec{x}, \vec{q}, \vec{Q})$
- Finally, $T_3(\vec{q}) = \text{Compose}(T_2(\vec{Q}), \vec{q})$

The first two steps can be performed efficiently using the elegant method developed by Clarke *at al.* [70]. The well known algorithm [34,61] for computing reachable states is described in Figure 3.16. Given a set of present states, the set of states reachable in one step is computed as the Image of the set of present states by the next state functional vector Δ (step 3). The set of the newly reached states is then taken as the set of present states (step 4), and the process is repeated until no new states are reached. After each iteration the set of states reached thus far (*Reached*) is updated by adding the set of newly reached states (step 5).

Algorithm For Computing Covered Transitions

1. $Reached = From =$ initial state(s);
2. while (From $\neq 0$)
3. $\quad To = \textbf{Image}(\Delta,\ From)$;
4. $\quad From\ =\ To\ -\ Reached$;
5. $\quad Reached\ =\ Reached\ \cup\ From$;
6. endwhile

Figure 3.16: Algorithm for reachability analysis.

A transition in an FSM can either be represented as a pair of the form $\langle \vec{q}, \vec{x} \rangle$, where $\vec{q}$ is the current state and $\vec{x}$ is the input vector applied to it, or as a triple of the form $\langle \vec{q}, \vec{Q}, \vec{x} \rangle$, where $\vec{Q}$ is the next state. The latter better facilitates the representation and manipulation of sets of equivalent transitions.

The Image procedure outlined earlier is modified to give the set of state pairs $\langle$current state, next state$\rangle$ for a particular input vector $\vec{x}_i$, rather than for **some** input vector. More specifically, in the second step, the values in vector $\vec{x}_i$ are substituted in T_1, instead of existentially quantifying out the input variables. The third step involving the composition operation is eliminated. This modified image routine, **MImage**, is as follows:

- $T_1(\vec{x}, \vec{q}, \vec{Q}) =\ T(\vec{x}, \vec{q}, \vec{Q}) \wedge\ C(\vec{q})$
- $T_2(\vec{q}, \vec{Q}) =$ Substitute ($T_1(\vec{x}, \vec{q}, \vec{Q}),\ \vec{x}_i)$)

Definition 27 *The* **Output Relation** *is defined as*

$$O(\vec{y}, \vec{x}, \vec{q}) = \prod_{i=1}^{k} y_i \odot \lambda_i(\vec{x}, \vec{q}) \tag{3.8}$$

where each variable y_i *,* $i = 1, 2, \ldots, k$ *represents one of the* k *outputs of the FSM.*

The Augmented Transition Relation encodes the relationship between outputs, current state, next state and inputs.

Definition 28 *The* **Augmented Transition Relation** *is defined as*

$$ATR(\vec{y}, \vec{x}, \vec{q}, \vec{Q}) = T(\vec{x}, \vec{q}, \vec{Q}) \wedge O(\vec{y}, \vec{x}, \vec{q}) \tag{3.9}$$

$ATR(\vec{y}, \vec{x}, \vec{q}, \vec{Q}) = 1$ *if the next state is* $\vec{Q}$ *and the output is* $\vec{y}$ *for input* $\vec{x}$ *in current state* $\vec{q}$*.*

The procedure for computing the transitions that are exercised by a given input sequence, $\{\vec{x}_0, \vec{x}_1, \ldots, \vec{x}_n\}$, is described in Figure 3.17. The algorithm groups equivalent transitions together (steps 6–10), so that if one is exercised by the given input sequence, all of the transitions in that group are considered to be exercised. The ATR enables identification of equivalent transitions more easily, as described below.

From and **M** are characteristic functions representing the set of current states in terms of present state variables and the set of ⟨current state, next state⟩ pairs in terms of present state variables and next state variables respectively, and are represented as OBDDs. $\mathbf{T}_{prev}$ and $\mathbf{T}_{curr}$ are the characteristic functions which store the sets of traversed transitions. The modified image routine, MImage, is used to compute the next states for a given set of states and input vector (step 6). In fact, this predicate **M** encodes **all** transitions between those pairs of current states and next states. To find those transitions

Build Output Relation(O);

Build Augmented Transition Relation(ATR);

From($\vec{q}$) = InitialState(s);

$T_{prev} = 0$;

for (i=1;i $\leq$ number_of_vectors ;i++)

  $M(\vec{q},\vec{Q})$=MIMAGE(From($\vec{q}$), $\vec{x}_i$);

  $Z_1(\vec{x},\vec{y},\vec{q},\vec{Q})$=ATR($\vec{x},\vec{y},\vec{q},\vec{Q}$) $\wedge M(\vec{q},\vec{Q})$;

  $Z_2(\vec{y},\vec{q},\vec{Q})$ = Substitute ($Z_1(\vec{x},\vec{y},\vec{q},\vec{Q})$, $\vec{x}_i$);

  $Z_3(\vec{x},\vec{y},\vec{q},\vec{Q}) = Z_1(\vec{x},\vec{y},\vec{q},\vec{Q}) \wedge Z_2(\vec{y},\vec{q},\vec{Q})$;

  $Z_4(\vec{x},\vec{q},\vec{Q}) = \exists\ \vec{y}\ \ Z_3(\vec{x},\vec{y},\vec{q},\vec{Q})$;

  $T_{curr}(\vec{x},\vec{q},\vec{Q}) = T_{prev} \vee Z_4(\vec{x},\vec{q},\vec{Q})$;

  $T_{prev} = T_{curr}$;

  $To(\vec{Q}) = \exists\ \vec{q}\ M(\vec{q},\vec{Q})$;

  $From(\vec{q})$=Compose($To(\vec{Q})$, $\vec{q}$);

endfor;

Figure 3.17: Procedure for calculating transition coverage.

from this set which are equivalent to the transition induced by $\vec{x}_i$, we compute a predicate $\mathbf{Z}_1$ relating all output values and corresponding inputs for the set of ⟨current state, next state⟩ pairs given by $\mathbf{M}$ (step 7). We then determine the values on the outputs for the induced transition (step 8) and encode them in $\mathbf{Z}_2$. We need to find the subset of transitions in $\mathbf{Z}_1$ with the same output values as the transition being exercised. We intersect $\mathbf{Z}_2$, which encodes the required output values, with $\mathbf{Z}_1$, which encodes all transitions between that set of ⟨current state, next state⟩ pairs to obtain the group of equivalent transitions (step 9). Quantifying the output variables out of $\mathbf{Z}_3$ (Step 10) gives the equivalent transitions encoded as an OBDD $\mathbf{Z}_4$. Finally, the characteristic function $\mathbf{T}_{curr}$ stores all the exercised transitions (Step 11).

After completion of the loop, the TCM can easily be computed as the fraction of true valuations of $\mathbf{T}_{curr}$ (after existentially quantifying out the next state variables) over the fraction of true valuations of the OBDD storing the reachable transitions. The SCM can also be computed easily by existentially quantifying the next state and the input variables out of T_{curr}. Then SCM is the fraction of true valuations of this resulting OBDD over the fraction of true valuations of the OBDD encoding the reachable states.

3.2.5 Experimental Results

In this section, we present the ECFM automatically extracted from a microprocessor circuit description. The Viper is a 32-bit microprocessor designed at RSRE, England for safety-critical applications (a more complete description can be found in Chapter 5). It has four general purpose registers and two ALU registers, along with a memory address register and instruction register giving a total of 251 flip-flops. The number of primary inputs is 33. The approximately 40 instructions include arithmetic, comparison and Boolean operations besides instructions for reading and writing to memory. The Viper has a unique feature in that it halts operation if an exception is raised, which occurs if an illegal instruction is fetched or overflow occurs.

The total number of states is more than 10^{75}. However, most of it is due to the presence of the register file. We can cover an appreciable part of the control flow by exercising only a fraction of the 10^{85} transitions in the state diagram. We identified 5 flip-flops, the control, status and overflow flip-flops, as encoding the flow of control. We did not include the instruction register as part of the control state. At this point, it is instructive to discuss the ramifications of identifying a register as control or data. Recall that if a register is considered as data, it is deleted from the state space. However, it may appear in the ECFM primary input list because it is included in some support set. On the other hand, the dropped data register may not appear in the ECFM at all if it is not included in any support set. This may be undesirable in some cases. Let us take the destination index register as an example. If this register is considered as data, it is deleted from the state space. Furthermore, we are not interested in the register file, so no output or state depends on the destination index register and it does not appear in the ECFM at all! This may be undesirable if we are interested in making sure that all the registers are accessed during verification. In such case, the destination index register should be retained as part of control state. It is thus possible for the designer to look at the generated ECFM and go back and extend the control space if necessary, generating a new ECFM which includes any missing behavior.

The ECFM generated by us had 5 flip-flops and 75 inputs, with 17 reachable states out of 32 possible states. Thus, even with a conservative reachability analysis, we were able to significantly prune the state space under consideration. Building the BDDs for the ECFM and building the support set to determine the ECFM input list took 0.51 seconds on a SUN Sparc10. For lack of a functional test suite for the Viper, we decided to evaluate the functional coverage of a test set used for stuck-at fault detection. This test set consists of 4126 vectors, giving a stuck-at fault coverage of 87%. Evaluating the transition coverage of the ECFM for these vectors gave us TCM = 78.9% and SCM = 88.2%. Thus, a significant part of the control behavior is not exercised by the test vectors, though they provide high stuck-at fault coverage! The time taken to build the transition relation, the output relation and the ATR was 0.8 seconds. The time to compute the coverage metrics was 5.5 minutes.

The coverage evaluation program also gives us the transitions of the ECFM that are not covered by the input stimuli. As an indication of the behavior not covered by the given test set, we looked at one of the reachable states which were not covered. The circuit does not enter the halt state with the flag bit set. The flag bit is supposed to be set as a result of comparison operations or as carry/borrow for arithmetic operations, and is pivotal in nullifying certain instructions. The circuit is supposed to enter the halt state whenever overflow occurs or an illegal instruction is executed. This implies that an instruction causing overflow was not tested with the flag bit already set.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Chapter 4

Accepting High Level HDL Input

The techniques for verification of sequential circuits presented here are geared toward the verification of circuits in a practical design environment. However, these techniques will not gain acceptance in the design community unless they can take as input circuit descriptions which are used for other design tasks – for example, descriptions in a hardware description language like VHDL [1].

A major problem with hardware description languages is the semantic gap that exists between circuit descriptions and proof systems. Hardware description languages are formal languages, not formal systems which can be used to reason about the circuits they describe. However, modifying or extending the particular language that a verification system accepts to support the description of hardware designs requires the user to learn a new language and translate existing descriptions into that language. Consequently, it is not a popular approach with designers. Some attempts at formal verification using commercial HDLs transform the HDL input into a semantically equivalent form acceptable as input by theorem provers [71,72] which is again not agreeable to circuit designers. One solution to this problem is to specify a transformation of HDL descriptions into an equivalent intermediate mathematical model, such as finite state machines, that can be manipulated by a circuit verifier. This

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

solution has the added advantage that it can potentially make the circuit verifier independent of any single hardware description language. However, in order to maintain a high degree of confidence in the correctness of the transformation process, it is often necessary to restrict the use of the HDL to a well-understood subset. A second difficulty encountered when attempting to verify circuits specified by hardware description languages is related to **abstraction mechanisms**. Data abstraction is related to the use of abstract data types in the circuit specification. For example, the circuit may be specified using integers, although at the gate level the circuit is actually operating on bits, and we have seen that most automated verifiers use only a Boolean model of representation such as Binary Decision Diagrams. There have been a few approaches to translate HDL descriptions into input to automated verifiers [73]–[75]. These approaches either require the two descriptions to be verified to have the same number of registers, or require specification of an initial state for both machines. A hybrid approach, using a theorem prover as the front-end and a symbolic simulator as the back-end, so as to draw on the strengths of both approaches, was presented in [76]. However, the theorem proving expertise required by the front-end proved to be a hurdle and the system was used chiefly as a symbolic simulator.

The framework presented here has sought to address the requirement of accepting high level programs in commercial HDLs as input by accepting specifications in the form of VHDL descriptions (using only a well-understood subset of VHDL) and translating them into the tuples representing finite state machines accepted by the verifier. The subset of VHDL is modeled after the subset accepted by most commercial synthesis tools, and the rules for the translation

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

process are closely linked to the simulation semantics of VHDL. Obviously, we can only verify VHDL descriptions of designs which can be represented as finite state machines. Although VHDL has been chosen as the input language, the methodology presented here is general enough to be applied to any HDL.

VHDL descriptions of circuits at different levels of abstraction, from the **scheduled behavioral** level down to the gate level, are acceptable as input to this framework. The scheduled behavioral level has operations implicitly scheduled into states, *i.e.*, clock periods or phases are clearly delineated, but is a pure behavioral description in every other sense. It may be devoid of any structural information and could be described as a series of procedure and function calls. Such a description may contain high-level operators like $+$, $\leq$, and abstract data types like integers. This corresponds to a behavioral description after the scheduling stage [77] in the design process and is one level above the microarchitectural level. For example, the specification could be a scheduled control/data-flow graph. Though this is not the highest level possible, it is an essential step in the design process and is often the highest level of specification written in a hardware description language.

4.1 Compilation of VHDL Descriptions

The translation, or compilation, of VHDL programs into the form of finite state machines is done on the basis of an operational view of the program. Briefly, the operational semantics of a programming language is concerned with identifying the states of the computation represented by the program and specifying the meaning of constructs in the language as their effect on the state, *i.e.*, by a state transition function [78]. We identify the states of the computation of a VHDL program in the form of signals or variables which may be considered to be state-holding devices. The following are some examples of the compilation rules. Execution of a `WAIT` statement inside a process implies a change of state, and since we only deal with synchronous circuits, every `WAIT` statement condition must be on a clock edge. Other types of `WAIT` statements, for example, those with a timeout clause, are not allowed. A process with `WAIT` statements is then called a clocked process. A `SIGNAL`, other than a primary output, which is assigned in a clocked process implies a state-holding device. A `VARIABLE` which is read before it is assigned in a clock cycle also implies a state-holding device. The VHDL subset currently accepted for verification excludes constructs which do not have a mapping to hardware, for example, record types. Other restrictions on the VHDL subset include exclusion of resolution functions, bidirectional ports and multi-valued logic. Timing constructs are also excluded; the design must not depend on signal propagation delay for correct functionality. We assume that clock speeds are chosen so that the correct values from combinational blocks are latched on clock edges.

The compilation of VHDL programs into finite state machines is achieved by the extraction of a symbolic control table from the VHDL program and a data path formed by insertion of combinational subcircuits to perform the operations specified in the program. The general nature of the compilation operation for behavioral VHDL descriptions is shown in Figure 4.1. In the preprocessing step, the VHDL description is parsed, and state variables are identified. For VHDL programs which are described as a series of procedure and function calls and which may not have explicit control state variables, state variables and state transitions are introduced strictly on the basis of the oc-

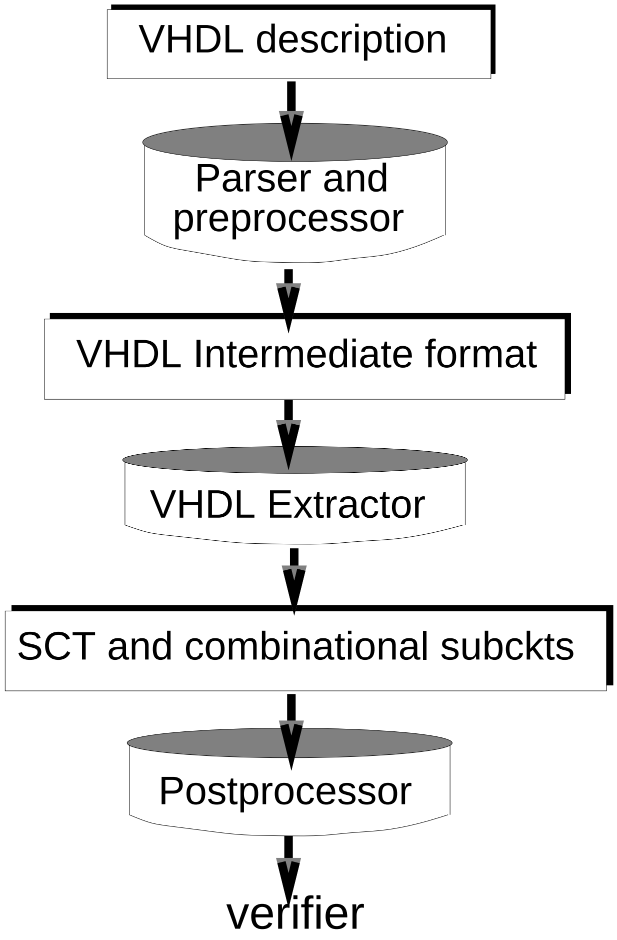


Figure 4.1: Compilation methodology.

currence of `WAIT` statements. Abstract data types are converted to bit vectors. Each assignment statement involving a high-level operation is replaced with a signal uniquely associated with that statement. These signals form the control signals to the data path formed by insertion of Boolean templates to perform the specified operations. Details of the preprocessor can be found in [79]. The preprocessor generates an intermediate form which is used by the extractor to generate the symbolic control table (SCT).

4.1.1 Symbolic Control Table

The SCT represents the behavior of the finite state machine that implements the control section of the VHDL program. It describes states, transitions between states, control signals that regulate data path actions, and output signals. Formally, the symbolic control table can be defined as a set of quadruples $< C \times S \times S \times A >$ where:

- C is a set of conditions, which represent the values of the input signals that will regulate the behavior prescribed by the particular quadruple
- S is a set of states, which may be symbolic or binary
- A is a set of actions, which are assignments of values that will be driven either on the output signals, or on the signals that the control section employs to regulate the operation of the data path.

The various conditions in the quadruples of the symbolic control table are stored as sets of cubes and are manipulated with set operators that are defined as C++ routines. Details of the extraction of the SCT from a VHDL program may be found in [80].

4.1.2 Generation of a Verified Data Path

The VHDL descriptions may contain high level operators like +, <, etc. on abstract data types like integers. Generation of FSMs with data paths from behavioral descriptions containing such operators is handled by using a library of Boolean templates. These templates are "golden" implementations of high level operations on abstract data types by Boolean operations on Boolean data types. For example, a template describing a ripple-carry adder on bit vectors (of specified length) is used to represent the addition operation on two integers, given the mapping between integers and bit vectors. The correctness of each template, with respect to first principles, is proved *a priori* with the use of the Boyer-Moore theorem prover [81]. A template is first proved to correctly implement a specification in Boyer-Moore logic written in terms of bit vectors by building the BDDs for each and comparing them. This Boyer-Moore structural specification is then shown to be equivalent to a higher level specification, also in Boyer-Moore logic, on abstract data types, say integers. A mapping function to map integers to bit vectors and vice versa is defined for this purpose. For example, the adder template operating on bit vectors is proved to correctly realize the addition operation on two integers, from a finite range, before inclusion in the library. Thus, the user does not have to be concerned with theorem proving.

Note that the Boolean templates are combinational sub-circuits and not sequential units. This is consistent with the level of abstraction of the VHDL descriptions, which is the scheduled behavioral level. A high-level operation implemented sequentially must have the steps in the sequential algorithm enumerated in the VHDL description with each step possibly involving some combinational operations. Boolean templates can then be utilized for the operations in each step.

A Boolean template is inserted in the behavioral description for every instance of a high level operator, resulting in a **maximally parallel** data path for the high level description. Most importantly, we have complete confidence in the correctness of this data path. Theorem proving techniques are thus

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

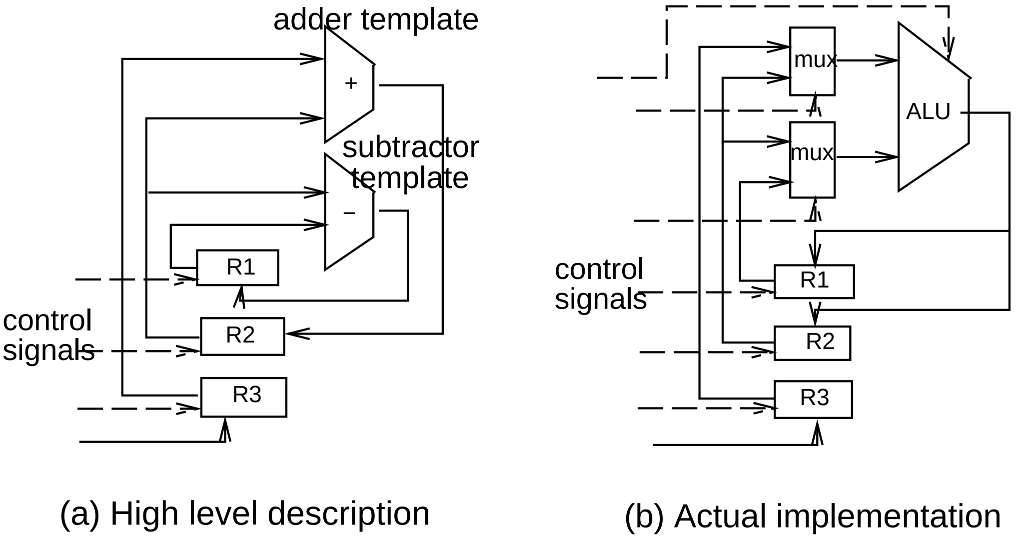


Figure 4.2: Generation of a verified data path

used, in an off-line manner, to handle data abstraction by correctly translating abstract, easy-to-describe specifications into Boolean specifications. After the insertion of the templates, the functional units in the data path of the resulting finite state machine have no correspondence with the functional units of the implementation. Correspondingly, the control units in the two descriptions do not correspond. Consequently, verification of large circuits cannot be performed by verifying the control unit of the implementation against that of the specification, or the implementation data units against units in the specification. For example, the data path for a high level description with the operations `R1 = R1 - R2` and `R2 = R2 + R3` executed sequentially is shown in Figure 4.2(a), whereas a possible implementation is shown in Figure 4.2(b). Nevertheless, the insertion of the templates accurately preserves the design intent of the original VHDL program.

The Boyer-Moore prover has also been used to verify the correctness of parameterized hardware module generators for the CATHEDRAL-II silicon compiler [82]. The modules are the basic building blocks of the synthesis system, and their correctness is crucial to the correctness of the synthesized designs. In that system, a parameterized structure description of a module is proven to correspond to a parameterized specification, both expressed in Boyer-Moore logic. In [82], such a correctness proof for an ALU module generator has been discussed. Further, for each generated instance of a module, net list comparison is used to verify the module layout against the Boyer-Moore structural description with instantiated parameters. Our framework could easily be enhanced to utilize parameterized template generators in a similar manner.

The symbolic control table and the Boolean templates now constitute a finite state machine representing the VHDL program. The postprocessor then maps the symbolic control table into a set of Boolean equations, which are input to the verifier along with the Boolean templates for the data path. The postprocessor may also have to do state assignment for symbolic states appearing in the symbolic control table. Currently, this is done in a random manner, because the verifier does not depend on the knowledge of the particular state encoding. However, the state assignments could be manipulated to reduce the sizes of the BDDs and improve efficiency.

The primary concern in this simplistic, syntax-directed translation of VHDL programs is minimizing the possibility of errors occurring in the translation process itself. In general, mapping HDL descriptions to hardware is not a difficult problem for synthesis systems. Errors are most likely to occur while optimizing the designs for area, performance, etc. Keeping this in mind, no optimizing transformations are attempted during the translation process in our

Table 4.1: Compilation of VHDL descriptions into FSMs with data path on a Sparc1.

example ckt	*Extraction of control table*		*Boolean templates*			
			Behavioral		*RTL*	
	Behavioral	*RTL*	*type*	*number*	*type*	*number*
Sequencer	-	12.1s	-	-	-	-
CPU	26.2s	28.1s	add	13	add	1
			comp	1		
AM2901	12.9m	4.4m	add	3	add	1
			misc.	11	misc	11
AM2910	18.8m	3.0m	sub	9	sub	2
			comp	13	comp	2
Viper	4.2m	-	add	4	-	-
			sub	2		
			misc	5		
Chip1	-	1.1s	-	-	add	6
					sub	4

verification methodology.

4.2 Compilation results

The C++ program for compilation of VHDL descriptions into finite state machines initially used the MCC VHDL analyzer [83]. A stand-alone parser and pre-processor is now being developed at the University of Texas [79]. Information about compilation of VHDL programs of several circuits into FSMs is given in Table 4.1. Behavioral and RTL descriptions of circuits have been compiled into finite state machines for the purpose of verification.

For each circuit, the behavioral description is a single VHDL process with operations scheduled into clock periods or states, abstract data types like integers and no structural information. The middle level is a behavioral RTL description with the design partitioned into functional blocks. The various columns show the time taken to extract the symbolic control table from the VHDL description and the number of Boolean templates inserted for high level operators for both behavioral and RTL descriptions. As can be seen, the number of Boolean templates is different for the FSMs representing the behavioral and RTL specifications, leading to differently structured data paths. The times shown are for extraction of the symbolic control table from each description. Insertion of the Boolean templates was done manually. The new parser and pre-processor [79] fully automates this process.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Chapter 5

Hierarchical Verification

The process of designing and implementing integrated circuits follows a natural hierarchy from the abstract high level specifications which express the desired functionality to the final low level net list implementing the circuit. This top-down, hierarchical style is indispensable in handling the complexity involved in this process. For the same reasons, a hierarchical verification style which fits in well with the design flow is a desirable feature in any verification approach. The techniques presented in this thesis can be incorporated into such a hierarchical framework.

5.1 A Hierarchical Verification Framework

The top level design in this framework is a high level behavioral description written in some hardware description language. This description is incompletely specified, but does have micro-architectural information like the number of clock cycles required to execute some instruction, whether the design is pipelined, etc. This level of description contains abstract data types like integers and high level operators for operations like addition and subtraction. Any algorithm for implementing complicated operations, for example a division algorithm, is scheduled into a specific number of clock cycles in such a scheduled behavioral description. This top level design is verified by the user against properties specified on the basis of the desired functionality and with full knowledge of the micro-architectural choices made in the design. This behavioral description is then successively refined into more detailed descriptions through a series of well-defined synthesis steps, whether manual or automated, such as partitioning, hardware allocation, unit binding, architectural optimization and logic synthesis. The design at each stage in this synthesis process can be verified to correctly implement the design at a higher level by proving that it contains that specification machine. Pre-verified Boolean templates are used to verify the correct implementation, with respect to mathematical specifications, of high level operators at lower levels.

The skeleton of this framework is shown in Figure 5.1. Very often the design at some stage in the synthesis process does not meet performance requirements or some other interface requirements and has to be redesigned. This could entail modifying the top level behavioral description and verifying it again before repeating the synthesis process (this loop is not shown in Figure 5.1). It is then very important that verification not be the bottleneck in this cycle and fast, automated verification techniques such as the ones proposed here can prove to be indispensable.

As we have seen, design verification techniques cannot exploit simple correspondences available at lower levels in the design hierarchy because of the nature of the specifications and the large abstraction gap from the implementation. They must then utilize other abstraction mechanisms to reduce the size of the model and the complexity of the verification process for large and complex circuits. One natural abstraction mechanism is to leave unspecified

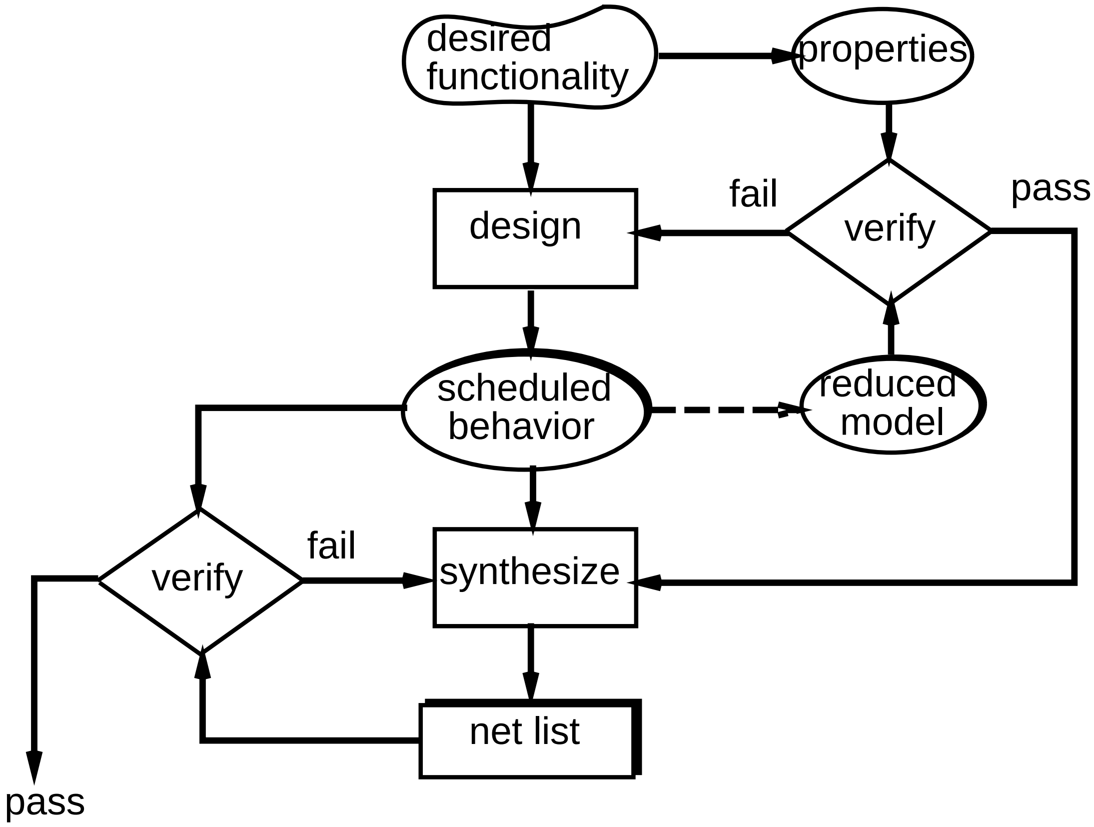


Figure 5.1: Hierarchical verification.

that part of the design behavior which is not pertinent to the property being verified. This is effective in cases where it reduces the size of the state space of the model. An example is a circuit being fed inputs from an external controller. The behavior of the controller could be left unspecified so that the circuit is verified for all possible inputs from the controller. Such modular verification techniques have been utilized in the context of CTL model checking [29] and language containment [31].

While the above example results in a conservative form of verification, it is not true that all abstraction mechanisms ensure that a property holds at lower levels of the design hierarchy. The use of non-determinism is a case in point. Incompletely specified descriptions could be viewed as non-deterministic state machines. Using CTL model checking [13] to verify existential properties of such machines offers no guarantees for the property to hold in less abstract, refined implementations. Existential properties cannot even be specified in the framework for hierarchical reasoning built around the language containment paradigm [31].

The framework presented here also requires care in its application to the hierarchical verification of circuits. As explained earlier, our design verification paradigm views properties as either specifying *good* behavior or looking for the existence of *bad* behavior. A property specifying *good* behavior holds for a design only if that behavior is explicitly defined in the design description. Verifying containment of that design in a lower-level implementation then proves that the implementation possesses all specified behavior of the design and by implication, also satisfies the property. Essentially, design states that satisfy the property have compatible states in subsequent implementations which contain the design. Therefore, a set of general starting states will satisfy the property in both the design and a lower-level implementation. However, the same cannot be said of properties requiring *good* behavior of **all** states, not just of a set of general starting states. For example, proving that there exists a path to a good state from all states in the design does not ensure that there exists such a path for all states in a lower-level implementation, though it may contain the design. However, the containment verification points out any incompatible states in the implementation which may not satisfy the good behavior. If such states do exist, it is the user's responsibility to verify the behavior in those states (as was done to check reset behavior of the implementation in Figure

2.2).

Verification of a property which checks for absence of *bad* behavior requires all unspecified behavior be considered to be undesirable. Hence, if no design state exhibits the *bad* behavior, all implementation states compatible with any design state will definitely not exhibit the *bad* behavior. Given containment of the design in the implementation, this set of compatible states will necessarily include any specified set of initial states. However, such a property may not hold of incompatible states in refined implementations. Again, however, the containment check points out any such incompatible states in the implementation.

In summary, the hierarchical reasoning employed here ensures that any properties, whether existential or universal, which are satisfied by the states in a design will necessarily hold true of all implementation states implementing the specified behavior of the design, *i.e.*, states compatible with some design state.

Apart from using incomplete specifications, the verification methodology followed in this framework gives us a convenient mechanism to reduce the state space of the circuit model. As has been stated earlier, this framework is targeted at state-transformation systems like processors. Such systems have large state spaces chiefly due to their data paths. Correct implementation of these data path operations is verified during implementation verification using pre-verified Boolean templates as the specifications. Consequently, functional correctness properties dealing with data path operations of the design need not be verified during design verification. The properties to be verified then deal mainly with control and synchronization aspects of the design. In many cases, this enables us to scale down the size of the data path without affecting the validity of the property being verified. In some instances, it is possible to specify the properties to be verified on the control part of the design only, eliminating the data path altogether [44]. However, nothing can alter the fact that making the design verification problem tractable is always ultimately dependent on the user's experience and knowledge of the design.

5.2 A Complete Example

This section presents the verification of a simple microprocessor, the Viper, with its instruction set as the top level specification and a gate level design as the final implementation. The verification requires the use of all the techniques presented in this thesis, and thus is a good example to illustrate their efficacy. The Viper was originally designed with formal verification in mind, and as such, its key feature is its simplicity. It has no pipelining, no interrupts, no cache or other such features of current microprocessors. However, we feel it is a significant first step in the application of these techniques to current design practice.

5.2.1 The Viper Microprocessor

We provide a brief description of the Viper and its architecture. A complete description can be found in [49]. The Viper is a 32-bit microprocessor designed at RSRE, England for safety-critical applications. It has an accumulator, two index registers and a program counter in its register file, as well as a memory address register, an instruction register and two ALU registers. The instruction set provides several arithmetic, logical and comparison operations, as well as

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

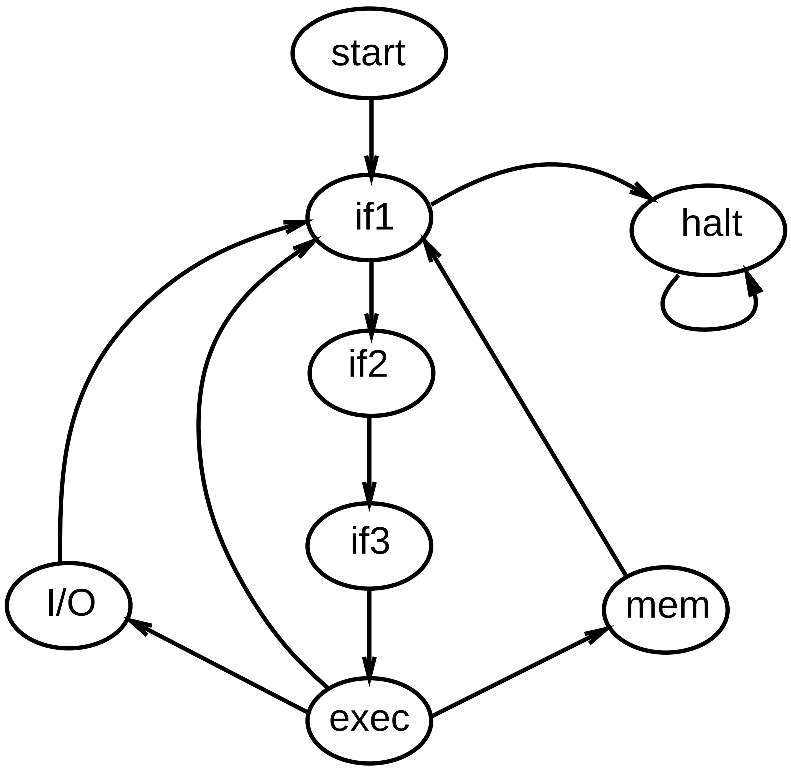


Figure 5.2: High level state diagram of Viper.

memory read and write instructions and a no-op instruction, for a total of thirty-five instructions. There is a single bit register which holds the result of comparisons. In the Viper, any illegal operation, overflow or computation of an illegal address halts execution and raises the **stop** signal from the ALU.

The top level circuit description is a behavioral description in VHDL with operations scheduled into states. The state diagram of the major state machine at this level is shown in Figure 5.2. Certain key registers like the program counter and status registers are initialized in the **start** state. The instruction is fetched and decoded, and the program counter is incremented in the **if1**, **if2** and **if3** states. The instruction is executed in the **exec** state. An exception causes the **stop** flag to be set, and the processor enters the **halt** state on the next clock. The net list synthesized from this description has 251 flip-flops and approximately 5000 gates. We verified several properties of this design, including part of the instruction set, some of which are described below. The net list was then verified against the behavioral description as mentioned earlier and elaborated here.

5.2.2 Design Verification

We ultimately wish to verify that the behavioral description possesses certain desired functionality such as initialization, correct execution of instructions or correct response to illegal instructions. In verifying instructions, we will check that the processor executes each instruction correctly from a most general set of starting states and returns to some state in that set after each instruction. This implies that any sequence of these instructions will also execute correctly. Thus, the properties we wish to check involve only one traversal through the instruction fetch-execute-writeback loop. In the Viper, any instruction compares at most two registers at a time and suitably scaling down data registers can still correctly exercise boundary conditions such as overflow. Furthermore, at this stage we only wish to check that an instruction is decoded correctly and that data transfer occurs properly, not that the implementation of the operation specified by the instruction is functionally correct. On the basis of all the above considerations, we can justifiably scale down the data path of the processor without affecting the validity of the verification. In fact, since boundary conditions are still exercised correctly, any property involving eventuality can also be verified on the reduced design. Properties involving some specific sequence of instructions are examples of properties that may not hold

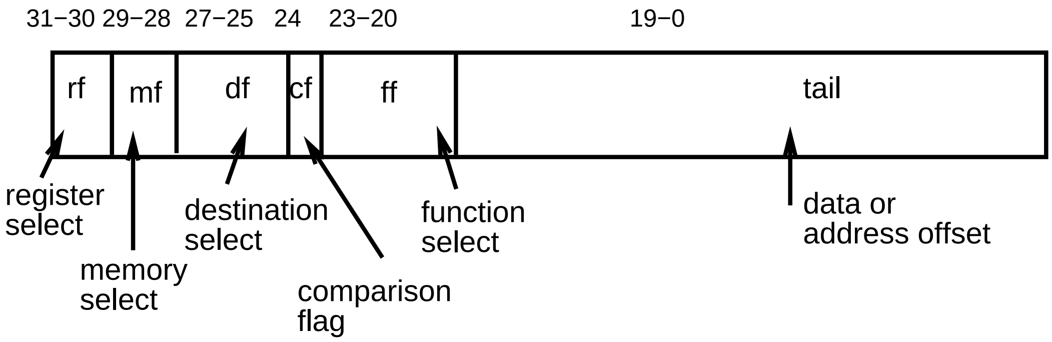


Figure 5.3: Instruction format of Viper.

on the actual design.

Since the verification compares the BDDs for the Boolean functions for each operation being executed, we require at least two bits to distinguish between the different instructions in the instruction set of the Viper and verify that the instruction has been decoded correctly. Consequently, we reduced each register in the register file from 32 bits to 2 bits. The data offset in the instruction register was correspondingly reduced from 20 bits to 1 bit. The instruction format is shown in Figure 5.3. The scaled down design has 34 flip-flops (approximately 10^9 reachable states) and is of manageable size. In specifying properties to be verified, we attempted to exercise every path in the flow chart for instruction decoding for the Viper shown in Figure 5.4. We also verified the correct operation of the stop mechanism for certain instructions. Some of the properties verified for the scaled down design are described below. Runtimes are on a Sparc10 with 160Mb RAM.

Property (1): Verifying that the start state of the Viper is not reachable from all states took 2.1 seconds. This was done by adding a dummy output to the design, setting it high only in the start state and modeling the specification

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

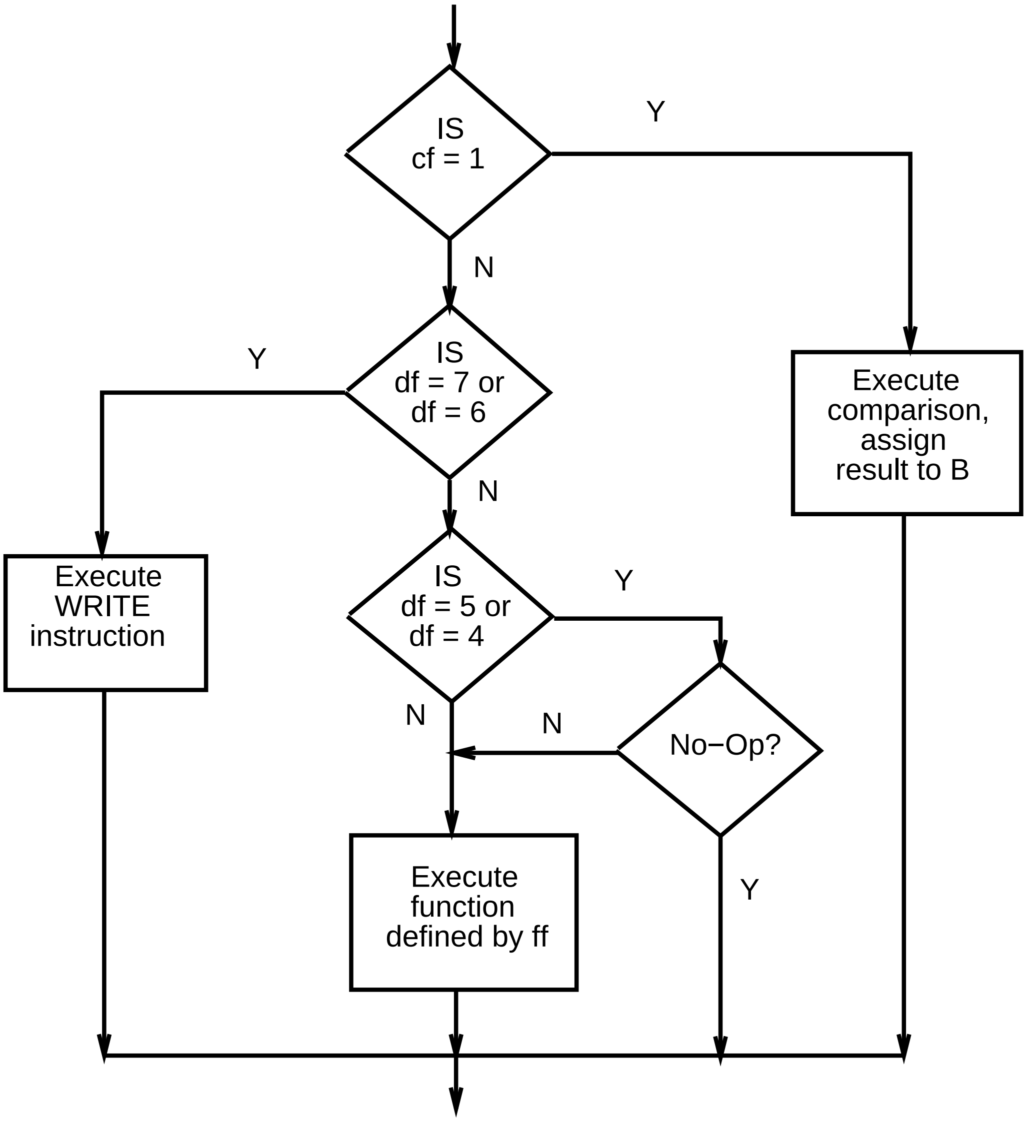


Figure 5.4: Logic of instruction decoding in Viper.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

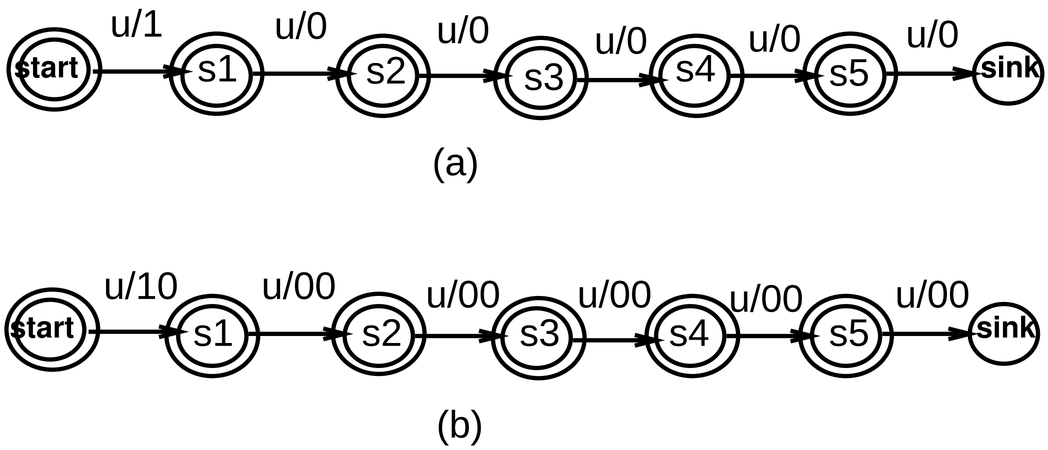


Figure 5.5: Specification for correct instruction sequencing in Viper.

machine as a single state with a self loop and output always low. The start state is then unreachable from states compatible with the specification state (which is all the states as is obvious from the state diagram in Figure 5.2). This exposed a serious flaw in the design: it cannot be initialized on power-up!

Property (2): An instruction requires a maximum of five clock cycles. We checked whether execution always returns to the fetch instruction state **if1** within six clocks by adding an output to the design which was high only in the fetch state and using the specification machine in Figure 5.5(a). The result told us that execution will not return to the **if1** fetch state in six clocks if the **stop** flag is set in the fetch state. This took 3.5 seconds. The witness produced was a sequence of don't care vectors with the machine starting in the **if1** state with the **stop** flag set and then looping in the **halt** state. Time taken to build the transition relation was 19.5 seconds and witness generation took 0.6 seconds. We then checked only those execution sequences where the **stop** signal stays low. Adding a second output to the specification which was always low Figure 5.5(b), corresponding to the **stop** output of the design, did not show any design

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

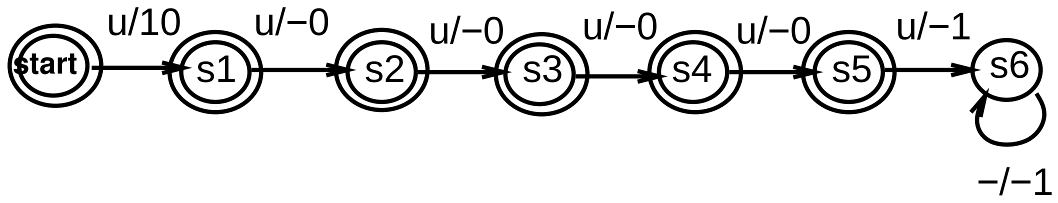


Figure 5.6: Specification for correct halt sequence for Viper.

state weakly compatible with the specification start state. This proved that the desired behavior was satisfied by the design in cases where the exception is not raised. Checking this property took 7.2 seconds.

Property (3): On the lines of the above property, we attempted to verify that there do indeed exist instructions where the machine goes into the **halt** state and stays there. The labeled specification machine for this property is shown in Figure 5.6. The first output is a dummy output, also added to the Viper, and set high only in the fetch state **if1**. It is unspecified in the other states of the property machine because execution returns to the fetch state at some point before going to the **halt** state, as can be seen from Figure 5.2, and we did not want to place undue constraints on the execution path. The **stop** signal is the second output. The fetch state **if1** was shown to be weakly compatible with the specification start state in 9.6 seconds. The witness sequence generated gave an illegal instruction as the cause of the exception being raised. Witness generation took 5.2 seconds after the transition relation was built.

We also verified a few instructions from the instruction set [49]. Since the data path of the model is 2 bits wide, the intent was to verify the decoding logic and proper data transfer rather than correct function implementation. For these properties, the data signals in the specification machines, for example **reg** in Figure 5.7, are mapped to the corresponding registers in the register

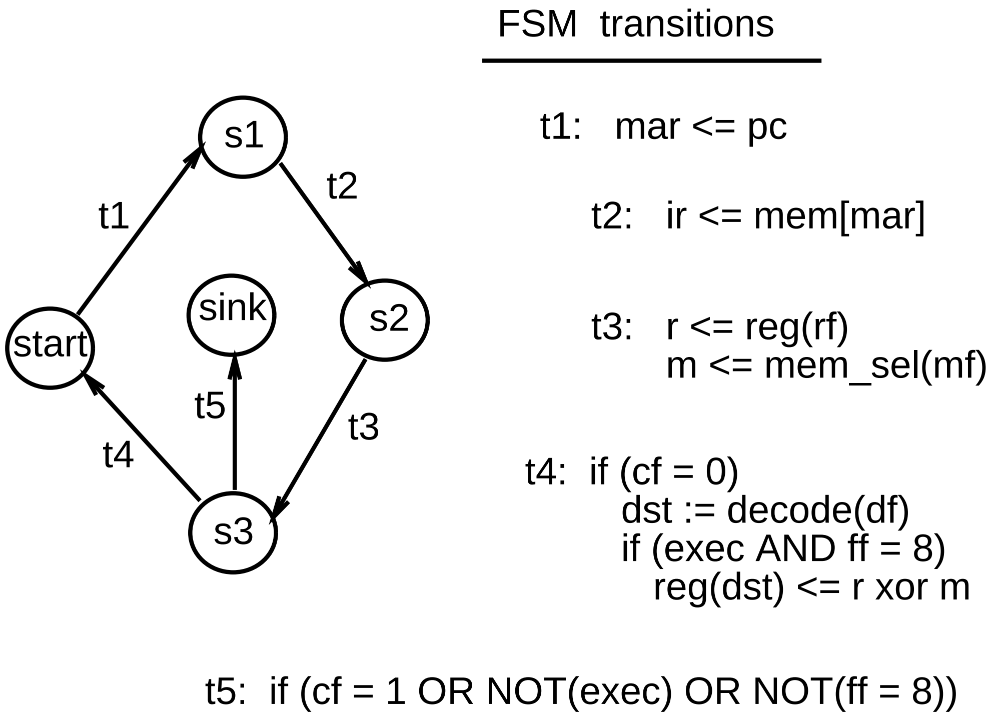
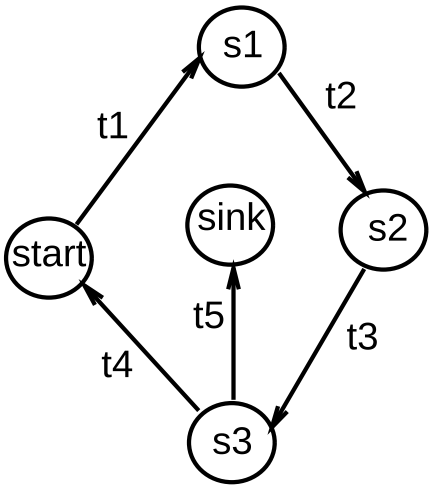


Figure 5.7: Specification for XOR instruction.

file of the design. This mapping implies that a specification data signal and its corresponding design register must have the same values in every clock cycle. To facilitate this, the flip-flops are dropped and the flip-flop inputs are taken as primary outputs and the flip-flop outputs are made primary inputs. This ensures that the corresponding signals feed the same values into the specification and the design on each clock. The intent is to show that the design implements the behavior of the property machine for the same data values in the user-visible registers.

Property (4): The specification machine for the simple XOR instruction is shown in Figure 5.7. The path in the decoding logic of Figure 5.4 followed by this specification machine is demarcated by cf=0, df=3 or 2 or 1 or 0, and ff = 0. The No-op loop from Figure 5.4 was left out of this simple machine.



FSM transitions

t1: mar <= pc

t2: ir <= mem[mar]

t3: r <= reg(rf)
m <= mem_sel(mf)
dst <= decode(df,b)

t4: if (cf = 0 AND exec AND ff = 8)
if (NOT(no−op))
reg(dst) <= r XOR m

t5: if (cf = 1 OR NOT(exec) OR NOT(ff = 8))

Figure 5.8: Specification for XOR and No-op instructions.

There are no labeled states in this specification machine. The fetch state **if1** in the major state machine, with the **stop** flag reset and for any values of other registers, was shown to be compatible with the specification start state. Verification took 1.88 minutes.

The No-op instruction depends on the value of the comparison flag. Including the No-op loop in Figure 5.4 as part of the behavior of the property machine requires mapping the comparison flag flip-flop of the Viper to a corresponding flip-flop in the property machine. Due to this mapping, any flow of control in the Viper involving the comparison flag must be mimicked in the specification machine exactly. For this design, selection of the destination register for the result of ALU operations depends on the comparison flag. This

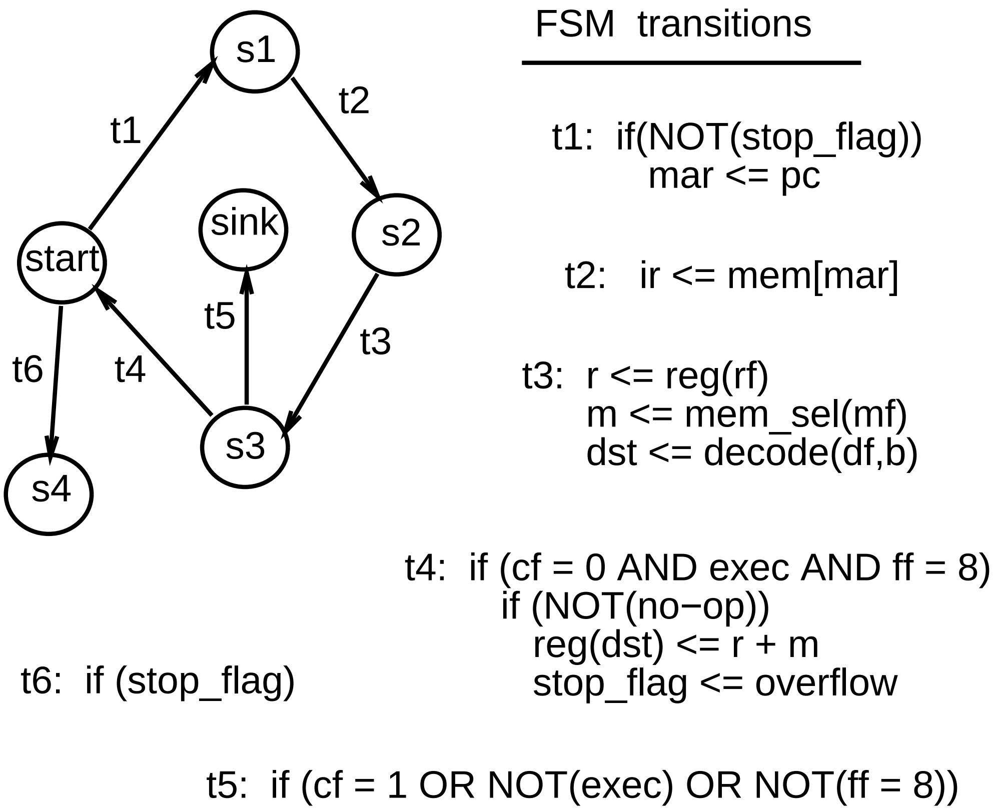


Figure 5.9: Specification for ADD instruction.

selection is made in the **if2** state in the major state machine of the Viper and must be mimicked in the specification machine. This modified specification machine is shown in Figure 5.8. The difference lies in the operations done on each transition. The specification covers the paths cf=0, df=5 or 4 or 3 or 2 or 1 or 0, and ff=0 in Figure 5.4. Verification took 1.78 minutes. Again, the **if1** state with **stop** flag reset was shown to be compatible with the specification start state.

Property (5): The specification machine for the ADD instruction is shown in Figure 5.9. The distinguishing feature of this instruction is that the Viper raises a flag when an overflow occurs and halts instead of fetching

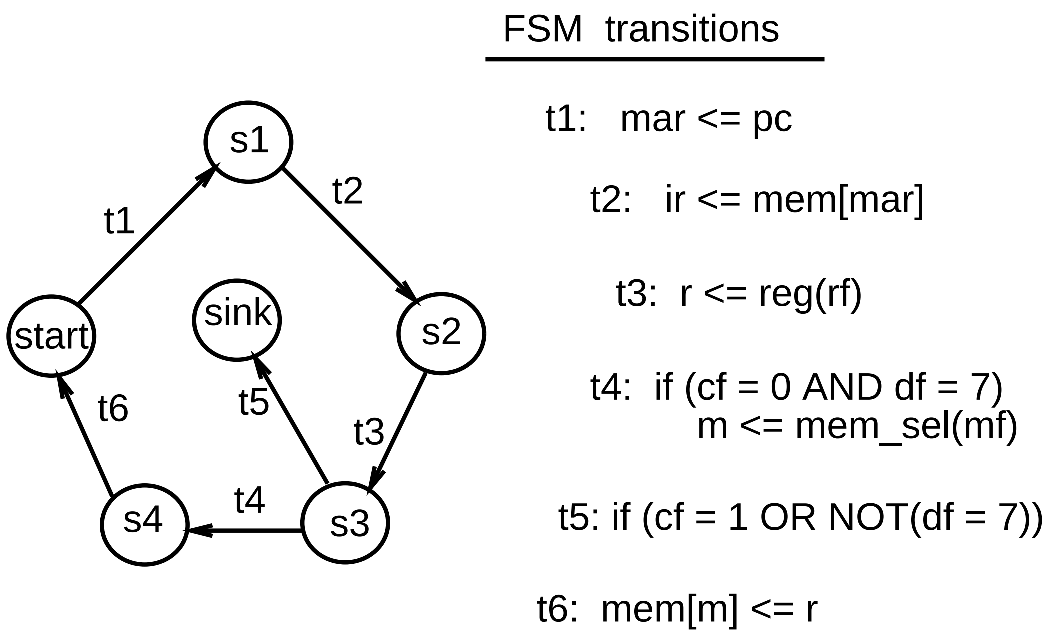


Figure 5.10: Specification for WRITE instruction.

the next instruction. The path covered by this specification is the same as the previous path with ff=5 instead of ff=0. It took 6.39 minutes to show that the fetch state **if1** is compatible with the specification start state.

Property (6): The specification machine for a simple comparison instruction ($r < m$) is the same as the machine shown in Figure 5.7 with the appropriate changes in the instruction encoding. There are no labeled states in this specification machine as well. The path in Figure 5.4 taken by this machine is the one demarcated by cf=1 and ff=0. The fetch state **if1** in the major state machine, with the **stop** flag reset and for all values of other registers, was shown to be compatible with the specification start state. Verification took only 15.6 seconds.

Property (7): The specification machine for the WRITE instruction which covers the path cf=0, df=7 or 6 in Figure 5.4 is shown in Figure 5.10. It has six states but is otherwise similar to the machine for the XOR instruction. Verification took 13.1 seconds and the fetch state **if1** with **stop** flag reset was shown to be compatible with the specification start state.

Note that verification is always over all possible data values for the model, all possible initial states and for all possible input sequences. This can prove indispensable when the execution flow is more complex, as in pipelined processors and processors with interrupts, to check for a sequence of inputs causing some bad sequence of states.

5.2.3 Implementation Verification

The synthesized net list was verified against the VHDL description by proving that it strongly contained the FSM extracted from the VHDL program. These results are shown in Table 2.5 and are not repeated here. As stated earlier, the compilation of the VHDL description into a state machine required four adder templates, two subtractor templates and five miscellaneous templates implementing Boolean operations like XOR on bit-vectors. In contrast to this data path for the specification consisting of several pre-verified functional units, the net list has only a single ALU implementing all the necessary operations. The extraction of the symbolic control table only took 4.2 minutes on a Sparc1.

All the registers in the register file, as well as the instruction register, the memory address register and the ALU registers correspond exactly in the VHDL description and the net list. Such correspondences enabled us to abstract away 248 of the 251 flip-flops. The encoding of the control states, however, was different in the two descriptions and combinational verification techniques would not suffice. After data correspondences, the modified descriptions had 283 primary inputs and 304 primary outputs. Due to this large input-output set and the diverse arithmetic operations in the instruction set, the BDDs could not be built for all the outputs together using a single variable ordering. Thus verification was unsuccessful in spite of the state space being reduced to only 8 states by data correspondence. We then partitioned the outputs into three sets (Table 2.5) and constructed the BDDs for each set separately to compute the set of strongly compatible states. The subsequent verification found no errors in the synthesized net list.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Chapter 6

Conclusion

This thesis has shown that automated, state-based verification techniques can be used to verify large circuits efficiently and without requiring user expertise in formal logic systems. The techniques presented here exploit particular characteristics of the problem domain, like correspondences between data registers of the specification and implementation, to make the verification problem tractable. Unfortunately, such correspondences do not exist in cases such as a pipelined implementation and its non-pipelined specification because the two descriptions do not operate in lock-step. However, given current design practice, data register correspondence is usually a justifiable assumption. These techniques were extended to handle retiming, the only optimization technique applied late in the design cycle which destroys the otherwise simple one-to-one register correspondences. Though proving strong containment is a conservative verification approach, it did not result in any false negatives for the verification of the circuits considered by us. It remains to be seen whether the aggressive optimization done in a practical design scenario would produce so many false negatives as to make this approach too cumbersome.

We have utilized the power of theorem provers in a small way through the use of a library of Boolean templates which addresses one of the most apparent shortcomings of state-based approaches. An added benefit is that the Boolean templates can also be used in a stand-alone manner as specifications to verify individual combinational modules. While these templates have been used to generate a provably correct data path, the compilation of HDL programs into state machines remains the "informal" link in this formal methodology. The subset of VHDL accepted by us is small so as to be able to keep the translation process simple and error-free. Designers should also follow the same principles. Any constructs that are open to misinterpretation, whether by a tool or another designer, are potential causes of errors. Designers must also be aware of the assumptions made by a verification tool. For example, the verification techniques presented here only handle single-phase, edge-triggered flip-flops. Direct application of these techniques to circuits with latches would require the implicit assumption that signal delays between latches are longer than the clock period.

Understanding the assumptions made and abstractions employed in building the circuit model and knowing precisely what has been verified is even more difficult in design verification. As we have seen, the size of the model has to be reduced using a host of available abstraction mechanisms to facilitate verification. The hierarchical framework presented here ensures that properties proved on an incompletely specified, top level specification hold on all implementation states implementing the specified behavior. However, incompleteness of specifications is not always sufficient and must be frequently used in conjunction with other abstractions. It is very easy to overlook all the implications of the abstraction used and assume that a property has been verified on the actual circuit when it has only been verified on the abstract model. The capability for generation of counterexamples and witnesses is thus important not just for debugging but also to check the validity of the model by checking that the paths satisfying the property are valid in the actual circuit.

There needs to be much more work done in exploring the expressiveness and facility of the proposed specification framework for the verification of real systems. At the same time, the techniques forming the basis of the verification algorithms have wide application in other areas of computer-aided design. The ECFM is a prime example of applying these techniques to a seemingly unrelated area: improving coverage of simulation vectors. Automatic extraction of the control flow from a design has several other potential applications, for example in areas like formal design verification and test generation. One technique for dealing with the state explosion problem in the formal verification of large systems is to reduce the state space by eliminating parts of it that are not relevant to the issue at hand. The ECFM should provide a convenient means of extracting the essence of a circuit for the purpose of verifying properties dealing with the flow of control in the circuit.

Though state-based techniques are attractive, theorem provers have a number of significant strengths such as handling data abstraction and exploiting the hierarchical nature of complex designs. While the library of Boolean templates significantly extends the applicability of our state-based verifier, it would be remiss of us to relegate theorem proving to this minor role. With design complexity far out of the reach of current verification practice, theorem proving has a major role to play in a hierarchical, modular verification methodology. In the simplest scenario, component modules of a design could be verified using state-based approaches, and provers used to prove the correctness of the composition of the modules. The challenge lies in integrating this approach into a user-friendly environment.

Throughout this work, the goal has been the successful application of automated techniques to the verification of large circuits. But there has been no concerted effort on our part to improve the basic low level techniques used in the verification program. There is no dearth of research in this direction as is evident by the work in generating good variable orders for OBDDs, improving the efficiency of the Image and Inverse Image operations and proposing new, more efficient variations of BDDs. Any verification tool, including ours, will benefit from these advances. Unfortunately, there does not seem to be any imminent breakthrough in improving the capabilities of current techniques significantly. However, it is our opinion that there are huge gains to be made in applying existing techniques to real designs and simply exploiting the opportunities provided by the unique nature of the design environment, as is borne out by the results presented here.

Finally, formal verification is not meant to be a push-button technique or to provide a guarantee of correctness. Formal techniques should be viewed as tools that are available to designers to probe, explore, debug, and on the whole, better understand their designs. It is then the designer's prerogative to put these powerful tools to good use.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

BIBLIOGRAPHY

[1] *IEEE Standard VHDL Language Reference Manual - Std 1076-1987.* 1988.

[2] *Formal Design Verification of Hardware Design*, M. Yoeli, Ed. IEEE Computer Society, 1991.

[3] M. McFarland, "Formal Verification of Sequential Hardware: A Tutorial," *IEEE Trans. on Computer-Aided Design*, vol. 12, No. 5, pp. 633–654, May, 1993.

[4] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, vol. 1, No. 2, pp. 151–238, Oct., 1992.

[5] W. Hunt, "Microprocessor Design Verification," *Journal of Automated Reasoning*, vol. 5(4), pp. 429–460, Dec., 1989.

[6] J. Joyce, "Formal Verification and Implementation of a Microprocessor," *VLSI Specification, Verification, and Synthesis*, G. Birtwistle and P. Subrahmanyam, Eds. Kluwer Ac., pp. 129–157, 1988.

[7] M. Srivas and M. Bickford, "Formal Verification of a Pipelined Microprocessor," *IEEE Software*, Sept., 1990, pp. 52–64.

[8] S. Akers, "Binary Decision Diagrams," *IEEE Trans. on Computers*, vol. C-27, No. 6, pp. 509–516, June, 1978.

[9] L. Fortune, J. Hopcroft and E. Schmidt, *The Complexity of Equivalence and Containment for Free Single Variable Program Schemes*, vol. 62 of LNCS. Springer-Verlag, 1978.

[10] R. Bryant, "Graph–based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, vol. C-35, No. 8, pp. 677–691, Aug., 1986.

[11] O. Coudert, C. Berthet and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Proc. of Intl. Workshop on Automatic Verification Methods for Finite State Systems*, vol. 407 of LNCS, pp. 365–373, 1989.

[12] O. Coudert, C. Berthet and J. Madre, "Verification of Sequential Machines Using Boolean Functional Vectors," *Formal VLSI Correctness Verification*. Elsevier Sc., pp. 179–196, 1990.

[13] J. Burch, E. Clarke, K. McMillan and D. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *Proc. 27th DAC*, 1990, pp. 46–51.

[14] 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. 1, No. 2, pp. 244–263, April, 1986.

[15] E. Clarke, O. Grumberg and D. Long, "Model Checking and Abstraction," *Proc. ACM Symp. on Principles of Prog. Languages*, Jan., 1992.

[16] J. Burch, E. Clarke and D. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," *Proc. 28th DAC*, 1991, pp. 403–407.

[17] J. Burch, E. Clarke, K. McMillan, D. Dill and L. Hwang, "Symbolic Model Checking: 10^20 States and Beyond," *Proc. 5th LICS*, 1990.

[18] K. McMillan and J. Schwalbe, "Formal Verification of the Encore Gigamax Cache Consistency Protocol," *Proc. of Intnl. Symp. on Shared Mem. Multiproc.*, Japan, 1991, pp. 242–251.

[19] D. Dill, A. Drexler, A. Hu and C. Yang, "Protocol Verification as a Hardware Design Aid," *Proc. ICCD*, 1992, pp. 522–525.

[20] I. Gertner and R. Kurshan, "Logical Analysis of Digital Circuits," *CHDLs and their Applications*, 1987, pp. 47–68.

[21] Z. Har'El and R. Kurshan, "Software for Analytical Development of Communication Protocols," *AT&T Tech. Jrnl.*, vol. 69, No. 1, pp. 45–59, Jan., 1990.

[22] R. Bryant, D. Beatty and C. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," *Proc. 28th DAC*, 1991, pp. 397–402.

[23] R. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Trans. on Computer-Aided Design*, vol. 6, pp. 634–649, July 1987.

[24] D. Beatty and R. Bryant, "Formally Verifying a Microprocessor Using a Simulation Methodology," *Proc. 31st DAC*, 1994, pp. 596–602.

[25] O. Coudert and J. Madre, "A Unified Framework for Formal Verification of Sequential Circuits," *Proc. ICCAD*, 1990, pp. 126–129.

[26] C. Pixley, "A Computation Theory and Implementation of Sequential Hardware Equivalence," *Proc. DIMACS Workshop on CAV '90*, Providence, 1991, pp. 293–320.

[27] F. Aelten, J. Allen and S. Devadas, "Event-based Verification of Synchronous, Globally Controlled, Logic Designs Against Signal Flow Graphs," *IEEE Trans. on Computer-Aided Design*, vol. 13, No. 1, pp. 122–134, Jan., 1994.

[28] M. Genoe, L. Claesen, E. Verlind, F. Proesmans and H. Man, "Automatic Formal Verification of Cathedral-II Circuits from Transistor Switch Level Implementation upto High Level Behavioral Specifications by the SFG-Tracing Methodology," *Proc. EDAC*, 1992, pp. 54–58.

[29] E. Clarke, D. Long and K. McMillan, "Compositional Model Checking," *Proc. 4th Annual Symp. on Logic in Comp. Sc.*, Asilomar, CA, June, 1989, pp. 353–362.

[30] M. Chiodo, T. Shiple, A. Sangiovanni-Vincentelli and R. Brayton, "Automatic Compositional Minimization in CTL Model Checking," *Proc. ICCAD*, 1992, pp. 172–178.

[31] R. Kurshan, "Analysis of Discrete Event Coordination," *Stepwise Refinement of Digital Systems*, 1989, pp. 414–453, LNCS 430, Springer-Verlag.

[32] A. Hu and D. Dill, "Reducing BDD Size by Exploiting Functional Dependencies," *Proc. 30th DAC*, 1993, pp. 266–271.

[33] G. Cabodi, P. Camurati and S. Quer, "Auxiliary Variables for Extending Symbolic Traversal Techniques to Data Paths," *Proc. 31st DAC*, 1994, pp. 289–293.

[34] H. Touati, H. Savoj, B. Lin, R. Brayton and A. Sangiovanni–Vincentelli, "Implicit State Enumeration of Finite State Machines Using BDD's," *Proc. ICCAD*, 1990, pp. 130–133.

[35] F. C. Hennie, *Finite–State Models for Logical Machines.* John Wiley, 1968.

[36] A. Friedman and P. Menon, *Theory and Design of Switching Circuits.* Comp. Sc. Press, 1975.

[37] S. Hwang and A. R. Newton, "An Efficient Verifier for Finite State Machines," *IEEE Trans. on Computer-Aided Design*, vol. 10, No. 3, pp. 326–334, March, 1991.

[38] B. Lin, H. Touati and R. Newton, "Don't Care Minimization of Multi-Level Sequential Logic Networks," *Proc. ICCAD*, 1990, pp. 414–417.

[39] C. Pixley, G. Beihl and E. Pacas-Skewes, "Automatic Derivation of FSM Specification to Implementation Encoding," *Proc. ICCD*, 1991, pp. 245–249.

[40] T. Filkorn, "A Method for Symbolic Verification of Synchronous Circuits," *Computer Hardware Description Languages and Their Applications*, D. Borrione and R. Waxman, Eds. Elsevier Sc., pp. 249–259, 1991.

[41] O. Coudert, C. Berthet and J. Madre, "Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams," *Proc. Workshop on Automatic Verification of Finite State Systems*, Rutgers, 1990, pp. 23–32.

[42] G. Cabodi, P. Camurati and S. Quer, "Efficient State Space Pruning in Symbolic Backward Traversal," *Proc. ICCD*, 1994, pp. 230–235.

[43] T. Tamisier, "Computing the Observable Equivalence Relation of a Finite State Machine," *Proc. ICCAD*, 1993, pp. 184–187.

[44] M. Fujita, "RTL Design Verification By Making Use of Datapath Information," *Proc. ICCD*, 1992, pp. 592–497.

[45] K. Brace, R. Bryant and R. Rudell, "Efficient Implementation of a BDD Package," *Proc. 27th DAC*, 1989, pp. 40–45.

[46] M. Fujita, H. Fujisawa and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," *Proc. ICCAD*, Nov. 1988, pp. 2–5.

[47] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proc. ICCAD*, 1993, pp. 42–47.

[48] J. Jain, J. Bitner, D. Moundanos, J. Abraham and D. Fussell, "A New Scheme to Compute Variable Orders for Binary Decision Diagrams," *Proc. 4th Great Lakes Symp. on VLSI*, Mar., 1994, pp. 105–108.

[49] W. Cullyer, "Implementing Safety-Critical Systems: The Viper Microprocessor," *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam, Eds. Kluwer Ac., pp. 1–26, 1988.

[50] J. Bitner, J. Jain, M. Abadir, J. Abraham and D. Fussell, "Efficient Verification of Multiplier and Other Difficult Functions Using IBDDs," *Proc. CICC*, May, 1992, pp. 5.5.1–5.5.5.

[51] C. Leiserson and J. Saxe, "Optimizing Synchronous Systems," *Proc. 22nd Ann. Symp. on Foundations of Computer Science*, 1981, pp. 23–26.

[52] C. Leiserson, F. Rose and J. Saxe, "Optimizing Synchronous Circuitry Through Retiming," *Proc. 3rd Caltech Conf. on VLSI*, 1983, pp. 87–116.

[53] N. Shenoy, K. Singh, R. Brayton and A. Sangiovanni-Vincentelli, "On the Temporal Equivalence of Sequential Circuits," *Proc. 29th DAC*, 1992, pp. 405–409.

[54] H. Touati and R. Brayton, "Computing Initial States of Retimed Circuits," *IEEE Trans. on Computer-Aided Design*, vol. 12, No. 1, pp. 157–162, Jan. 1993.

[55] S. Malik, K. Singh, R. Brayton and A. Sangiovanni-Vincentelli, "Performance Optimization of Pipelined Logic Circuits Using Peripheral Retiming and Resynthesis," *IEEE Trans. on Computer-Aided Design*, vol. 12, No. 5, pp. 568–578, May, 1993.

[56] B. Lockyear and C. Ebeling, "The Practical Application of Retiming to the Design of High-Performance Systems," *Proc. ICCAD*, 1993, pp. 288–295.

[57] S. Kang and S. Szygenda, "Design Validation: Comparing Theoretical and Empirical Results of Design Error Modeling," *IEEE Design and Test*, March, 1994, pp. 18–26.

[58] J. Madre, O. Coudert and J. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM," *Proc. ICCAD*, 1989, pp. 30–33.

[59] P. Chung, Y. Wang and I. Hajj, "Diagnosis and Correction of Logic Design Errors in Digital Circuits," *Proc. 30th DAC*, 1993, pp. 503–508.

[60] K. Cheng and J. Jou, "A Functional Fault Model for Sequential Machines," *IEEE Trans. on Computer-Aided Design*, vol. 11, No. 9, pp. 1065–1073, Sept., 1992.

[61] H. Cho, G. Hachtel and F. Somenzi, "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *IEEE Trans. on Computer-Aided Design*, vol. 12, No.7, pp. 935–945, July, 1993.

[62] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, W. H. Freeman, 1978.

[63] F. Maruyama and M. Fujita, "Hardware Verification," *IEEE Computer.* pp. 22–32, Feb., 1985.

[64] M. Fujita and H. Fujisawa, "Specification, Verification and Synthesis of Control Circuits with Propositional Temporal Logic," *Computer Hardware Description Languages and their Applications.* Elsevier Sc., pp. 265–279, IFIP, 1990.

[65] R. Hojati, T. Shiple, R. Brayton and R. Kurshan, "A Unified Approach to Language Containment and Fair CTL Model Checking," *Proc. 30th DAC*, 1993, pp. 475–481.

[66] M. Abramovici, M. Breuer and A. Friedman, *Digital Testing and Testable Design*. New York: W. H. Freeman, 1990.

[67] M. Abadir, J. Ferguson and T. Kirkland, "Logic Design Verification via Test Generation," *IEEE Trans. on Computer-Aided Design*, vol. 7, No. 1, pp. 138–148, Jan., 1988.

[68] K. Cheng and A. Krishnakumar, "Automatic Functional Test Generation Using the Extended Finite State Machine Model," *Proc. 30th DAC*, 1993, pp. 86–91.

[69] M. S. Pierre, S. Yang and D. Cassiday, "Functional VLSI Design Verification Methodology for the CM-5 Massively Parallel Supercomputer," *Proc. ICCD*, 1992, pp. 430–435.

[70] J. Burch, E. Clarke, D. Long, K. McMillan and D. Dill, "Symbolic Model Checking for Sequential Circuit Verification ," *IEEE Trans. on Computer-Aided Design*, vol. 13, No. 4, pp. 401–424, April, 1994.

[71] S. Rajgopal, K. Hedlund and D. Reeves, "Integrating Hardware Verification with CHDLs," *CHDLs and their Applications*. IFIP, pp. 153–163, 1991.

[72] D. Borrione, L. Pierre and A. Salem, "Formal Verification of VHDL Descriptions in the PREVAIL environment," *IEEE Design & Test of Computers*, June, 1992, pp. 42–55.

[73] J. Madre and J. Billon, "Proving Circuit Correctness Using Formal Comparison Between Expected and Extracted Behavior ," *Proc. 25th DAC*, 1988, pp. 205–210.

[74] T. Kam and P. Subrahmanyam, "Comparing Layout with HDL Models: A Formal Verification Technique," *Proc. ICCD*, 1992, pp. 588–591.

[75] A. Bartsch, H. Eveking, H-J. Faerber, M. Kelelatchew, J. Pinder and U. Schellin, "LOVERT - A Logic Verifer of Register-Transfer Level Descriptions," *Formal VLSI Correctness Verification, VLSI Design Methods II*, L. Claesen, Ed. IFIP, pp. 247–256, 1990.

[76] C. Seger and J. Joyce, "Linking BDD-Based Symbolic Evaluation to Interactive Theorem-Proving," *Proc. 30th DAC*, 1993, pp. 469–474.

[77] D. Gajski, A. Wu, N. Dutt and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Ac., 1992.

[78] J. E. Donahue, *Complementary Definitions of Programming Language Semantics*, vol. LNCS No. 42. Springer-Verlag, 1976.

[79] N. Nayampally, "Compilation of High-level VHDL Descriptions for Finite-state Machine Verification," Dept. of ECE, Univ. of Texas at Austin, M.S. Thesis, Aug., 1995.

[80] J. Moondanos, "Formal VLSI Verification Techniques Based on State-Machine Comparison," Dept. of ECE, Univ. of Texas at Austin, PhD. Dissertation, Aug., 1993.

[81] R. Boyer and J. Moore, *A Computational Logic Handbook.* Ac. Press, 1988.

[82] D. Verkest, L. Claesen and H. Man, "On the Use of the Boyer-Moore Theorem Prover for the Correctness Proofs of Parameterized Hardware Modules," *Formal VLSI Specification and Synthesis, VLSI Design Methods I*, L. Claesen, Ed. IFIP, pp. 99–116, 1990.

[83] "MCC VHDL System : Intermediate Representation Description," Tech. Report MCC CAD-086-91 Q, 1991.

Y. V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits,"
Ph.D. Dissertation, The University of Texas at Austin, December 1995.

Vita

Yatin Hoskote was born in Bombay, India, on 27th May, 1968, the son of Lata and Vasant Hoskote. He entered the Electrical Engineering program at the Indian Institute of Technology, Bombay in 1985 and graduated with a B.Tech. in May, 1989. He received a Masters degree in the field of Fault-Tolerant Computing from the University of Texas at Austin in August, 1991. After a short internship with the CAD group at Advanced Micro Devices, Austin, he was admitted to the Ph.D program in Electrical and Computer Engineering at the University of Texas at Austin in Spring 1992. He has been a graduate research assistant at the Computer Engineering Research Center throughout his stay at the University. He worked as an intern with the IMD Architecture group at Intel Corpn., Folsom in the summer of 1992. In December 1994, he wed Ashwini Rao.

Permanent address: Laxminarayan Nivas
7th Road, Khar, Bombay 400052
India

This dissertation was typeset with LaTeX‡ by the author.

‡LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.