Formal verification of sequential circuits using implicit state enumeration

Mets, A.A.

Award date:
1994

Formal Verification of Sequential Circuits using Implicit State Enumeration

Master Thesis
by Arjen A. Mets

IBM Thomas J. Watson Research Center
February 1994 - October 1994
by order of Prof.Dr.ing. J.A.G. Jess,
supervised by ir. G.L.J.M. Janssen
and Dr. A. Kuehlmann

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section

This thesis is typeset by LaTeX in Computer Modern.

Formal Verification of Sequential Circuits using Implicit State Enumeration

Arjen A. Mets
Master Thesis

For Raffie

Abstract

Logic errors in sequential circuit designs are an important problem for circuit designers. They can delay getting a new product on the market or cause the failure of an electronic system that is already in use. The most widely used method for sequential verification is based on extensive simulation. This approach becomes infeasible when the number of reachable states of the circuit is very large. In the past, there has been considerable research on the use of theorem proving techniques for the verification of sequential circuits. The application of such methods requires an excessive amount of user guidance which is not acceptable in a practical design environment.

This thesis discusses another approach to address the sequential verification problem. In this approach, the behaviour of a sequential circuit is modeled as a finite state machine. State enumeration techniques are used to compute all reachable states of the product machine of two finite state machines. The equivalence of the two machines is verified by checking that they produce the same sequence of output values for any valid sequence of input values. In contrast to theorem proving, state enumeration techniques can be highly automated which makes them easier to incorporate in practical design methodologies. The proposed technique is based on the use of Binary Decision Diagrams to represent next state relations and sets of states. This avoids the explicit construction of the state transition graph of the finite state machine. Experimental results show that state enumeration techniques can handle circuits of practical size.

Acknowledgements

I would like to thank Prof. J. Jess and Dr. David LaPotin for providing me with the opportunity to do my graduation project at the IBM Thomas J. Watson Research Center. Furthermore, I owe special thanks to my supervisors Geert Janssen and Andreas Kuehlmann who contributed substantially to the ideas in this thesis. Both of them helped me to solidify my ideas in countless discussions when undoubtly they had more important things to do. Thanks also to Leon Stok, Koen van Eijk and Arvind Srinivasan who provided valuable suggestions while proofreading this thesis.

Yorktown Heights, September 1994.

Contents

Chapter 1

Introduction

Design correctness is a major concern throughout the design process of an integrated circuit. This involves checking several design qualities including functional behaviour, timing and testability. In this thesis, the discussion is limited to verifying the functional behaviour of a sequential circuit. There are several practical reasons for verification. The most obvious one is the high cost of correcting errors in digital designs. This cost increases with the rising level of integration in digital circuit technology. Another, perhaps less obvious reason is that despite of the use of automated design software, parts of a design are still implemented manually. This is especially true for commercial microprocessor designs where custom design techniques are used extensively in an effort to fully exploit CMOS performance.

Simulation has traditionally been used to check certain functional design properties. However, the simulation approach has proven to be inadequate due to the computational demands of the task involved. It is not practically feasible to simulate all possible input patterns to verify a hardware design. A relatively recent alternative to simulation is formal verification. Formal verification can be seen as a (set of) technique(s) which exhaustively proves certain functional design properties. Formal verification is, in some sense, like a mathematical proof. Just as correctness of a mathematically proven theorem holds regardless of the particular values it is applied to, correctness of a formally verified design holds regardless of its input values. Thus, consideration of all cases is implicit in a formal verification methodology.

Verity is a formal verification tool developed at the IBM T.J. Watson Research Center. It can be used to verify the logical equivalence between a high-level description of a design written in Verilog and a transistor-level net-list. Flexibility is allowed in the transistor-level implementation as a wide variety of implementation styles including static, dynamic and self-timed logic are supported. Verity is in daily use at several IBM design centers. It has been used successfully for several mainstream IBM processors including PowerPC[1] processor implementations.

The current version of Verity addresses the verification of sequential circuits in a restricted

[1] PowerPC is a trademark of International Business Machines, Incorporated.

manner. Registers in both designs must be identified and matched a priori between the high-level description and transistor-level implementation. This reduces the sequential verification problem to a combinatorial verification problem. This thesis is directed towards extending Verity to perform sequential verification. It involves developing new algorithms enabling sequential verification for practical designs in a way that requires the least amount of input from the designer.

1.1 Scope of the Thesis

There are a variety of approaches to verify sequential circuits. A comprehensive overview can be found in [6, 7]. The different techniques can be characterized into two categories:

1. Techniques adapted from theorem proving based on higher-order logic models take a top-down view of the hardware verification problem. They iteratively modify the hypothetical theorem by applying axioms or other previously proven theorems until it becomes a tautology. Because of their generality, these methods can model almost any behavioural system property. However, due to the universality, a great deal of user knowledge and guidance is required to successfully apply such techniques to practical designs.

2. State exploration techniques follow a bottom-up approach by explicitly or implicitly visiting all reachable states of the product machine $\mathcal{M}_A \times \mathcal{M}_B$ of two Finite State Machines (FSMs) $\mathcal{M}_A$ and $\mathcal{M}_B$ [4, 5, 19]. For all possible transitions from the initial states they check that both machines produces the same output value, thus proving the functional equivalence of $\mathcal{M}_A$ and $\mathcal{M}_B$ with respect to the pair of initial states. The use of Binary Decision Diagrams (BDDs) to represent sets of states together with a symbolic depth-first or breadth-first traversal of the state transition graph made this approach applicable for designs with a large number of states [4, 16, 19]. In contrast to theorem proving, state exploration techniques can be highly automated which makes them easier to incorporate into practical design methodologies. On the other hand, the size of BDDs for representing sets of states for practical circuits often grows exponentially, which limits the general application of such methods. This is a major problem since, beside dynamic variable ordering techniques, no effective variable pre-ordering technique for this application is known.

As noted above, formal verification techniques based upon state exploration tend to be more automatic than methods based on theorem provers. In addition, state exploration methods can easily produce a counter example trace that helps the user to debug a faulty circuit. Both, degree of automation and the option to provide the user with a counter example, are major issues in a practical design environment. Therefore, the approach in this thesis is directed towards state exploration techniques.

1.2 Organization

The remainder of the thesis is organized as follows. Chapter 2 presents the background definitions and terminology on Boolean functions, sequential circuits, FSMs and BDDs.

State exploration techniques for the verification of synchronous sequential circuits are the subject of chapter 3. The algorithms presented here use a symbolic breadth first search of the state-transition graph of the product machine $\mathcal{M}_x = \mathcal{M}_A \times \mathcal{M}_B$ of the FSMs $\mathcal{M}_A$ and $\mathcal{M}_B$ [5]. BDDs are used to represent sets of states and next-state relations.

Chapter 4 addresses the problem of finding the optimal ordering of the next-state relations in the case when conjunctive partitioned next-state relations are used. The size of the BDD representing the next-state relation rapidly grows too large for complex circuits. In order to overcome this limit, the full next-state relation is represented as a list of conjuncted next-state relations [4], one for each state bit. The ordering of the next-state relations has a significant impact on how early in the computation variables can be quantified out. The problem of finding the optimal ordering of the next-state relations which minimizes the size of the intermediate BDD is **NP**-hard [8].

In chapter 5, we describe the verification approach taken by Verity and show how state exploration techniques can be used to verify sequential circuits. In addition, it is shown how these techniques can be used to present a counter example trace in case of a verification failure. Conclusions and topics for future research are discussed in chapter 6.

Chapter 2

Definitions and Terminology

This chapter presents the definitions and terminology necessary for the remainder of the thesis.

2.1 General Definitions

An n input, m output Boolean function F is a mapping $\mathbb{B}^n \rightarrow \mathbb{B}^m$ where $\mathbb{B} = \{0, 1\}$. $\mathbb{B}^n$ is called the *domain* of F, and $\mathbb{B}^m$ is called the *co-domain* of F. If $m > 1$, then F is a *multiple output* function. Let $\underline{x} = \{x_1, \cdots, x_n\}$ be the variables spanning the domain and $\underline{y} = \{y_1, \cdots, y_m\}$ be the variables spanning the co-domain. Then, F can be represented as a vector of y_i with $i = 1, \cdots, m$, where each y_i corresponds to a single output Boolean function f_i, which may depend on x_i with $i = 1, \cdots, n$:

$$F = [y_1, y_2, \cdots, y_m] = [f_1(x_1, \cdots, x_n), \cdots, f_m(x_1, \cdots, x_n)]$$

The support of a Boolean function is the set of variables the function depends on.

Definition 2.1 The *image* of a function $F : \mathbb{B}^n \rightarrow \mathbb{B}^m$, over a specified sub-domain $C \subseteq \mathbb{B}^n$, is defined as

$$F(C) = \{\underline{y} \in \mathbb{B}^m \mid \underline{y} = F(\underline{x}),\ \underline{x} \in C\}.$$

If $C = \mathbb{B}^n$, the image of C by F is called the *range* of F.
Similarly, the *inverse image* or *pre-image* over a specified subset $A \subseteq \mathbb{B}^m$ is the set

$$F^{-1}(A) = \{\underline{x} \in \mathbb{B}^n \mid \underline{y} = F(\underline{x}), \underline{y} \in A\}$$

□

Image and pre-image computations play an important role in the verification algorithms which are discussed in Chapter 3.

2.2 Sequential Circuits and Finite State Machines

In this thesis we consider single clock synchronous sequential circuits composed of combinational gates and storage elements. All storage elements are controlled by the same clock. A Mealy-type finite state machine (see Figure 2.1) is used to model the behaviour of the circuit.

Definition 2.2 A *Mealy finite state machine* (FSM) $\mathcal{M}$ is characterized by the 6-tuple

$$\mathcal{M} = (X, Y, S, S^0, \Delta, \Lambda)$$

with

X : the input alphabet, $X \subseteq \mathbb{B}^m$
Y : the output alphabet, $Y \subseteq \mathbb{B}^p$
S : a finite set of states, $S \subseteq \mathbb{B}^n$
S^0 : the set of initial or reset states
Δ : the next-state function $\Delta : S \times X \rightarrow S$
Λ : the output function $\Lambda : S \times X \rightarrow Y$

The sets X, Y, S and S^0 are non-empty. It is assumed that all primary inputs and outputs as well as all the states are Boolean valued as is generally the case in digital circuits. The next-state function Δ and the output function Λ implicitly define the *state transition graph* (STG) of the given FSM. States that can be reached, under some input sequences, from one of the initial states are called *reachable* states. □

The next-state function and the output function generally are multiple output functions. For a given circuit with m inputs, p outputs and n state-registers, the next-state function is $\Delta : \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}^n$. Each component function of Δ is associated with a state register in the circuit. The domain of Δ is the product of two Boolean sub-spaces, $\mathbb{B}^m$ corresponding to the *input space* and $\mathbb{B}^n$ corresponding to the *state space*. The variables spanning the input space are associated with the primary inputs of the circuit and are called *primary input variables*. The variables spanning the state space are associated with the outputs of the state registers and are called the *present state variables*. The variables spanning the co-domain are associated with the inputs of the state registers and are called the *next-state variables*. The next-state function models the combinational logic which determines the next-state of a state variable. Similarly, the output function is $\Lambda : \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}^p$, where the variables spanning the co-domain are associated with the *primary outputs* of the circuit.

The next-state relation which can be derived from the next-state function plays an important role in the verification algorithms discussed in chapter 3. Intuitively, a pair of states (s', s) is contained in the next-state relation if state s' is reachable in one step from state s under some valid input vector $\underline{x}$. More formally:

Definition 2.3 Let f be the Boolean function vector of next-state functions $f : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^n$. Let $X = \{x_1, \cdots, x_m\}$ be the set of input variables, $V = \{s_1, \cdots, s_n\}$ be the

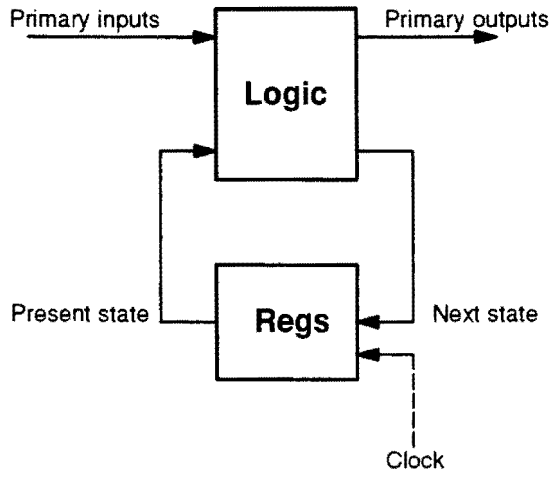


Figure 2.1: Mealy machine model

set of present state variables and $V' = \{s_1', \cdots, s_n'\}$ be the set of next-state variables. The *next-state relation* or *characteristic function* of f, denoted by $N : \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}$, is defined as:

$$N(X, V, V') = \prod_{1 \leq i \leq n} (s_i' \equiv f_i(\underline{x}, \underline{s}))$$

where f_i represents the next-state function for each state bit. The next-state relation is a functional representation of the following set:

$$\{(\underline{x}, \underline{s}, \underline{s'}) \in \mathbb{B}^m \times \mathbb{B}^n \times \mathbb{B}^n \mid \underline{s'} = f(\underline{x}, \underline{s})\}$$

□

The next-state relation can be abstracted from the primary inputs if we are only interested in the existence of an input value rather than the value itself. In that case, the next-state relation becomes a binary relation $N(V, V')$. The next-state relation of a synchronous sequential circuit can easily be derived from its structure. As a practical example, consider the modulo 8 counter in figure 2.2.

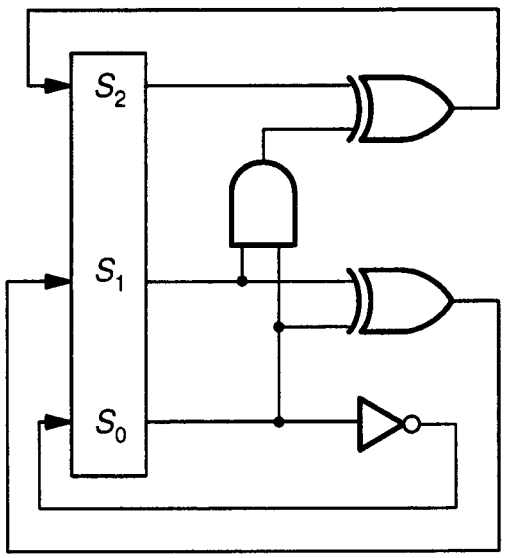


Figure 2.2: Synchronous circuit example: modulo 8 counter

The next-state functions for each state bit of the counter are given by:

$$s_0' = \neg s_0$$

$$s_1' = (s_0 \oplus s_1)$$

$$s_2' = (s_0 \wedge s_1) \oplus s_2$$

The equations above are used to define the relations for the individual state bits:

$$N_0(V, V') = (s_0' \equiv \neg s_0) \tag{2.1}$$

$$N_1(V, V') = (s_1' \equiv s_0 \oplus s_1) \tag{2.2}$$

$$N_2(V, V') = (s_2' \equiv (s_0 \wedge s_1) \oplus s_2) \tag{2.3}$$

which describe the constraints that each s_i' must satisfy in a legal transition. The next-state relations for the individual state bits are combined by taking their conjunction to form the next-state relation for the complete circuit:

$$N(V, V') = N_0(V, V') \wedge N_1(V, V') \wedge N_2(V, V') \tag{2.4}$$

In section 2.3, we show how Binary Decisions Diagrams can be used to represent next-state relations.

The *product machine* of two FSMs plays an important role in the verification algorithms which are discussed in the next chapter. It can be formally defined as:

Definition 2.4 The *product machine* $\mathcal{M}_x$ of a pair of machines $\mathcal{M}_A = (X, Y, S_A, S_A^0, \Delta_A, \Lambda_A)$ and $\mathcal{M}_B = (X, Y, S_B, S_B^0, \Delta_B, \Lambda_B)$ is characterized by $\mathcal{M}_x = (X, Y, S, S^0, \Delta, \Lambda)$ with

$$
\begin{aligned}
S &= S_A \times S_B \\
S^0 &= S_A^0 \times S_B^0 \\
\Delta\ ((s_A, s_B), x) &= (\Delta_A(s_A, x), \Delta_B(s_B, x)) \\
\Lambda\ ((s_A, s_B), x) &= (\Lambda_A(s_A, x) = \Lambda_B(s_B, x))
\end{aligned}
$$

The product machine $\mathcal{M}_x$ is made up of the two machines running in parallel. The states of $\mathcal{M}_x$ are pairs of states, one from $\mathcal{M}_A$ and one from $\mathcal{M}_B$. The next-state function Δ of $\mathcal{M}_x$ is defined to map pairs of states to pairs of states by applying Δ_A to the first state in the pair and Δ_B to the second one. □

Assume machines $\mathcal{M}_A$ and $\mathcal{M}_B$ are two FSMs to be compared. Intuitively, $\mathcal{M}_A$ and $\mathcal{M}_B$ are functionally equivalent if both machines have an identical interface and if, from a given pair of initial states, both machines produce the same sequence of output values for any valid sequence of input values. Figure 2.3 illustrates the equivalence check for two synchronous FSMs. The subscripts A and B are used to distinguish between both machines.

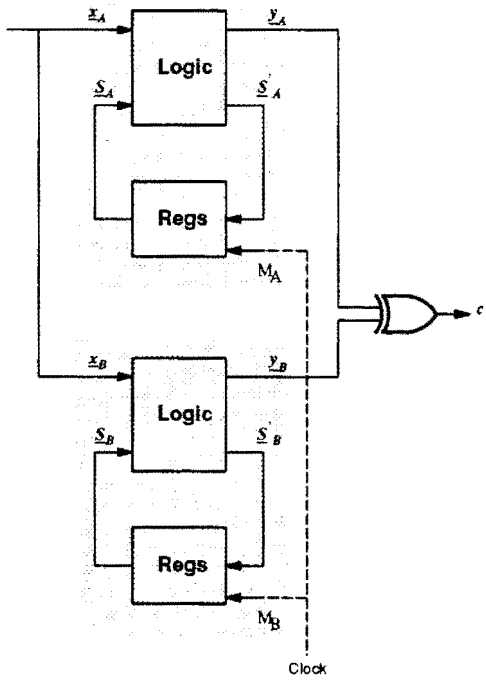


Figure 2.3: The product machine built of two FSMs

For the sake of functional comparison, the product machine $\mathcal{M}_x = \mathcal{M}_A \times \mathcal{M}_B$ is built. The primary input signals $\underline{x}_A$ and $\underline{x}_B$ are interconnected and driven by a common set of independent variables $\underline{x}$. All corresponding outputs are compared pairwise by *xor* functions whose results are logically *ored* to form signal c. The two machines are said to be functionally equivalent if and only if after initializing both machines $\mathcal{M}_A$ and $\mathcal{M}_B$ to their initial states, any input sequence produces a constant zero value on output c. Notice that no assumptions are made about the state encoding of the two FSMs to be compared.

If the FSMs $\mathcal{M}_A$ and $\mathcal{M}_B$ do not have state registers, both circuits implement a combinatorial function where the output values do not depend on past input values. In this case, successful comparison of the two circuits for a single clock cycle proves functional equivalence for any input sequence. This case is classified as *combinatorial logic verification.* The more general case where A and B contain arbitrary sets of state registers is referred to as *sequential logic verification.* In this thesis, no distinction is made between a FSM and a synchronous digital circuit.

2.3 Binary Decision Diagrams

This section briefly reviews the concept of Binary Decision Diagrams (BDDs). A more elaborate description can be found in [2]. Binary decision diagrams provide a compact, canonical form for the representation and manipulation of Boolean functions. They are formally defined as:

Definition 2.5 A *binary decision diagram* (BDD) is a directed acyclic graph (DAG) representation of a logic function. Each node in the DAG represents a Boolean function F and has an associated variable x_i and edges to exactly two other nodes (functions) in the DAG. The node F is written as the tuple (x_i, T, E) where x_i is called the top variable of the function F, T is the positive cofactor of F with respect to x_i and E is the negative cofactor with respect to x_i. The node F thus represents the function: $F = x_i T + \overline{x_i} E$. The sink nodes with nil then and else pointers, represent the Boolean constant functions 0 and 1. □

When using BDDs it is necessary to define an ordering on the Boolean variables. The variable at the root of the BDD is earlier in the ordering than all other variables. Each variable has a rank number which represents its position in the ordering. Ordered Binary Decision Diagrams are formally defined as:

Definition 2.6 An *ordered binary decision diagram* (OBDD) is a BDD with the constraint that the input variables are ordered. Every root-to-leaf path in the OBDD visits the input variables in strictly *ascending* order, i.e.,

$$\forall_{v \in V}\ rank(v) < min(rank(then(v)), rank(else(v))$$

where $rank(v)$ denotes the ordering position of variable v. □

Definition 2.7 A *reduced ordered binary decision diagram* (ROBDD) is an OBDD which does not contain a node v with THEN(v) = ELSE(v) nor does it contain distinct nodes v and v' such that the subgraphs rooted by v and v' are isomorphic. □

An important property of the ROBDD data structure is that it is a canonical form, i.e., two Boolean functions are equivalent if and only if they have the same ROBDD, assuming the same variable ordering. In this report the term BDD is used to refer to reduced ordered binary decision diagrams. The advantage of BDDs over other canonical representations of Boolean functions (such as a truth table) is that they are usually more compact and efficient algorithms exist to manipulate them. A particular application of BDDs is the representation of sets as *characteristic functions* defined as follows:

Definition 2.8 Let C be a set and A $\subseteq$ C. The *characteristic function* of A is the function $\mathcal{X}_A : \mathrm{C} \rightarrow \mathbb{B}$ defined by:

$$\mathcal{X}_A\,(a) = \begin{cases} 1 & \text{if } a \in A \\ 0 & \text{otherwise} \end{cases}$$

□

A characteristic function is nothing but another representation of a subset of a set. A characteristic function itself can be represented efficiently by a BDD, which is often much more compact than an explicit list of all elements. In addition, set operations can be represented by Boolean operations on the BDDs. For instance, the union of two sets is found by *oring* their BDDs. Relations can be represented as BDDs in a way similar to the representation of sets.

Definition 2.9 Let R be a binary relation over the sets A and B. The *characteristic function* of $R \subseteq A \times B$ is a function $\mathcal{X}_R : A \times B \rightarrow \mathbb{B}$ defined by:

$$\mathcal{X}_R(a,b) = \begin{cases} 1 & \text{if } (a,b) \in R \\ 0 & \text{otherwise} \end{cases}$$

□

The relation R can be represented by the BDD for its characteristic function. As a practical example, consider the most significant state bit of the modulo 8 counter described in section 2.2. Its next-state relation can be written as:

$$N_2(V,V') = (s_2' \equiv (s_0 \wedge s_1) \oplus s_2) \tag{2.5}$$

The corresponding characteristic function can be written as:

$$\mathcal{X}_R(\underline{s}', \underline{s}) = \overline{s_0} s_2' s_2 + \overline{s_1} s_2' s_2 + s_0 s_1 s_2' \overline{s_2} + s_0 s_1 \overline{s_2'} s_2 + \overline{s_2'}\,\overline{s_2}$$

Figure 2.4 shows the corresponding BDD. For the sake of simplicity, the BDD does not contain complement edges. Note that the structure of the graph depends on the chosen variable ordering. In this case, the variable order is s_0, s_1, s_2', s_2.

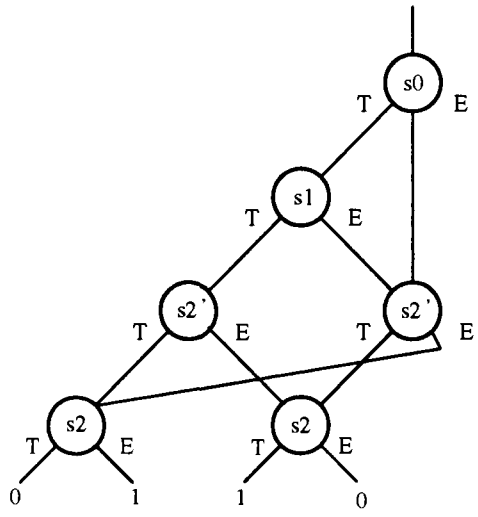


Figure 2.4: BDD representation for characteristic function $\mathcal{X}_R(V, V')$.

Boolean quantifiers such as the existential operator and the universal operator can also be implemented with BDDs as follows. The existential quantification (also called *smoothing*) of a Boolean variable v with respect to the Boolean formula f can be computed as

$$\exists v\, f = f_v + f_{\overline{v}}$$

where f_v is the usual cofactor operation. Similarly, the universal operator (also called the *consensus* operator) can be computed as

$$\forall v\, f = f_v . f_{\overline{v}}$$

The *generalized cofactor* is a new operator which can be used to reduce an image computation to a range computation. This operator was initially proposed by Coudert et al. in [5] and called the *constrain* operator. It can be formally defined as:

Definition 2.10 Let $F : \mathbb{B}^n \rightarrow \mathbb{B}$. Furthermore, let $c : \mathbb{B}^n \rightarrow \mathbb{B}$ with $c \neq 0$ and let $x_1 \prec x_2 \prec \cdots \prec x_n$ be an ordering of its input variables. The *generalized cofactor* of F with respect to c, denoted by F_c is defined as:

$$F_c = F \circ \pi_c$$

where the $\circ$ symbol denotes functional composition. The mapping $\pi_c : \mathbb{B}^n \rightarrow \mathbb{B}^n$ is defined as follows:

$$\pi_c(\underline{x}) = \begin{cases} \underline{x} & \text{if } c(\underline{x}) = 1 \\ \min\limits_{y, c(y)=1} d(\underline{x}, \underline{y}) & \text{if } c(\underline{x}) = 0 \end{cases}$$

$$\text{with } \; d(\underline{x}, \underline{y}) = \sum_{1 \leq i \leq n} |x_i - y_i|\, 2^{n-i}$$

π_c is a projection that maps a minterm $\underline{x}$ to the minterm $\underline{y}$ in the onset of c which has the closest distance to $\underline{x}$ according to the distance d. The particular form of the distance guarantees the uniqueness of $\underline{y}$ in this definition for any given variable order.

□

Intuitively, the representation of F_c is simplified by adding or removing minterms from the don't care set of c. Usually (but not always), the size of the BDD representing F_c is smaller than the BDD representation of F. A key property of the operator is that the image of a function F under a sub-domain c is equal to the range of the function F_c. The generalized cofactor can be computed efficiently in a single post-order traversal of the BDD representations of F and c [19].

2.3.1 Implementation of the BDD Package

Verity uses the BDD package available from the Design Automation Section at Eindhoven University of Technology. This package is based upon the BDD implementation as described by Brace, Rudell and Bryant [2]. The ITE operator forms the core of the package. ITE is a higher-order Boolean function defined for three arguments F,G,H which computes *if* F *then* G *else* H. This is equivalent to

$$\text{ITE (F,G,H)} = F.G + \overline{F}.H$$

The ITE operation can be used to implement all two argument Boolean operations. As ITE is the logical function performed at each node of the BDD, it is an efficient building block for many operations on the BDD. The programming language function for the ITE operator will be written as *ite*.

A global hash table, called the *unique table*, is used to allow a BDD node $< v, T, E >$ to be found in constant time. The unique table uses a hash function on the tuple $< v, T, E >$ to map the unique table entry to the BDD node $F = (v, T, E)$. All the nodes with the same hash value are stored in a linked list. Before a new node is added to the BDD, a lookup in the unique table determines if the node for that function already exists. If so, the existing node is used. Otherwise, a new node is created and stored in the unique table. With this technique, the BDD can be kept reduced without ever calling a reduce function.

The performance of the *ite* function is improved by the use of a memory function, the *computed table*. Unlike the unique table, the computed table has finite memory. Only the results of most recent computations are held. While the unique table is instrumental in maintaining the reduced form, the computed table is just for speeding-up *ite* computations. The table maps three nodes F,G and H to the result node *ite*(F,G,H) once this result has been computed. The computed table is implemented as a hash-based cache, i.e., a hash table without a collision chain. In case of collision, the new entry replaces the old one. Each entry in the computed table occupies 4 words: 3 words which form the key for the operation (e.g., ITE(F,G,H) uses F,G and H as the key) and a single word which is the operation result. A ratio of one cache entry is maintained for every four unique table entries so that the total memory usage of the package, including the overhead of the hash tables, is 24 bytes per BDD node on a 32 bit machine.

In order to recycle memory, the BDD package uses garbage collection based on reference counting. Each node in the BDD has a reference count, representing the number of other nodes that refer to it. A node with a reference count 0 is called dead; all nodes with positive reference counters are called alive. If a new node is created causing the unique table to become too full, then either garbage collection is performed (if the number of dead nodes in the DAG exceeds a given percentage, for instance 10% of the total number of nodes in the DAG) or the unique table is increased in size by a factor two. If the reference count overflows, the node will become *frozen*. A frozen node can never be freed.

Attributed edges have been proposed by several authors to improve BDD performance [10, 14, 17]. The edges in the BDD are tagged to indicate a modification of the function it points to. This reduces the size of the DAG by allowing a single node to represent several different functions. One trick that can be used is by explicitly indicating that a node's variable is to be interpreted negated, i.e., the roles of the *then* and the *else* edges are to be exchanged. This so called *inverted input edge* can be represented by a bit in the pointer value. This implementation trick is based on the fact that some bits of a pointer have a constant value because of memory alignment requirements. A sufficient rule to maintain a canonical BDD is to force the memory address of the *then* pointer to be less than the memory address of the *else* pointer. Another attribute edge is the *complemented edge*. It indicates that the connected formula is to be interpreted as the complement of the ordinary formula. Thus with complement edges, f and $\overline{f}$ share the same subgraph. f is an ordinary pointer to the subgraph and $\overline{f}$ is a complement pointer. Again, a bit in the pointer value can be used to represent the edge. A dot on an edge indicates it is a completed edge. To maintain a canonical form, the place where complement edges are used must be constrained:

1. The *then* link of every node must be a non-complemented edge and
2. the **1**-function is represented by a non-complemented edge to the only terminal node; the **0**-function is a complemented edge to this node.

As a result, always the left member of each equivalent pair shown in figure 2.5 is chosen.

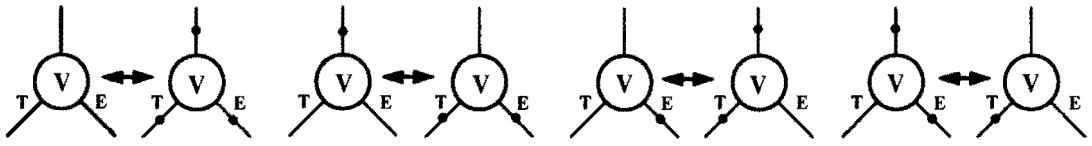


Figure 2.5: Four equivalent pairs of BDDs with complement edges.

Dynamic variable ordering [18] is used to address the problem that the size of the BDD representation for a given Boolean function depends on the chosen variable order. Dynamic variable ordering changes the current variable order as BDD operations are performed. This is done periodically by applying a minimization algorithm which reorders the variables

of the BDD to reduce its size. Because the variable order is no longer static, this technique is referred to as dynamic variable ordering. Dynamic variable ordering differs from the typical use of BDDs where variables are ordered once when the BDD is created and the order is maintained throughout all subsequent processing. Details of Rudell's sifting algorithm, which has been incorporated into the package, can be found in [15].

2.3.2 Implementation of a BDD Package for Sequential Verification

A BDD package for sequential verification has been developed based upon the theory presented in the previous sections. The package has been implemented as an extension to the BDD package mentioned in section 2.3.1. It provides specific functions for verification of sequential circuits. In particular, state variables have been introduced to incorporate state information into BDDs. State variables are treated in exactly the same way as combinatorial variables except they have additional information about their initial state and related next-state functions. The hash table structure of the unique table has been adapted to store this information for each state variable. The package uses an ordering for the BDD variables in which present and next-state variables are interleaved. State variables are introduced by a call to the function *bdd_create_state_var* which automatically creates a pair of state variables in such a way that every present state variable s is adjacent to its corresponding next-state variable s'. The introduction of state variables in pairs is beneficial for image computation (see chapter 3). One of our current tasks involves adapting Rudell's sifting algorithm [18] in such a way that corresponding state variables are kept in pairs. Several algorithms and routines of the package are discussed in more detail in the next chapter. The integration into Verity is described in chapter 5.

Chapter 3

Algorithms for FSM Verification

In this chapter we consider the problem of verifying the equivalence of two FSM's. The algorithms we discuss are based on an implicit enumeration of all the reachable states in the STG of the FSM. The state enumeration techniques are not only applicable to sequential verification but can also be used for state extraction and state minimization [12].

The functional equivalence of two sequential circuits, i.e., two Mealy machines $\mathcal{M}_A$ and $\mathcal{M}_B$, can be verified by traversing the STG of the product machine $\mathcal{M}_x = \mathcal{M}_A \times \mathcal{M}_B$ and searching for a transition where the two machines generate different output values [5]. If no such transition is found after exploring the whole reachable part of the STG, equivalence of $\mathcal{M}_A$ and $\mathcal{M}_B$ is proved. Thus, sequential verification of a FSM requires representing the STG of the product machine $\mathcal{M}_x$ and to traverse it either backwards of forwards to find the set of reachable states. The *forward traversal* starts from a set of initial states and repeatedly computes the set of states reachable from the previous calculated set. This requires an image evaluation of the transition function. The *backwards traversal* starts from a given set of states and computes the states from which that set is reached. This requires a pre-image evaluation of the transition function.

The explicit construction of the STG of the product machine is a very memory consuming operation. Therefore, it is restricted to machines with only a few states. Other approaches traverse the product machine without explicitly building its STG. Basically, there are two, well-known strategies to traverse the STG of the product machine: *depth-first* and *breadth-first traversal* [5]. Depth-first traversal suffers from the fact that the number of steps for the traversal is linear in the number of reachable states. This is the reason that the depth-first strategy is only suited for machines with a relative small number of states. The second strategy performs a breadth-first traversal of the STG. This strategy exercises multiple transitions in the STG simultaneously (implicit enumeration) and sets of states (e.g. the set of states visited thus far) are stored in the form of characteristic functions (BDDs). The breadth-first traversal algorithm is presented in detail in section 3.1. In section 3.2, we look at two algorithms for image computation. Section 3.3 presents experimental results comparing the two image computation algorithms. Improvements are discussed in section 3.4.

3.1 Finding Reachable States

Reachable state computations are the fundamental part of state machine verification. Let S_0 be the set of initial states, represented by the BDD $S_0(V)$. Our goal is to derive an algorithm which computes the BDD $S(V)$, representing the set of all reachable states. Consider first the set S_1, of states reachable in at most one step from S_0. Clearly, a state s_i belongs to S_1 if $s_i \in S_0$ or there exists a state $s_i \in S_0$ and a transition (s_i, s_j) in the next-state relation N. Thus, the set S_1 is given by

$$S_1 = S_0 \cup \{s' \mid \exists s \, [s \in S_0 \wedge (s, s') \in N]\} \tag{3.1}$$

Given the BDDs $S_0(V)$ and $N(V, V')$, the BDD representing S_1 can be computed by performing the logical operations corresponding to expression 3.1.

$$S_1(V') = S_0(V') \vee \underset{v \in V}{\exists} [S_0(V) \wedge N(V, V')] \tag{3.2}$$

Similarly, the states reachable in at most two steps from S_0 are represented by

$$S_2(V') = S_1(V') \vee \underset{v \in V}{\exists} [S_1(V) \wedge N(V, V')] \tag{3.3}$$

In general, the states reachable in at most $k+1$ steps are represented by

$$S_{k+1}(V') = S_k(V') \vee \underset{v \in V}{\exists} [S_k(V) \wedge N(V, V')] \tag{3.4}$$

Notice that each set of states is a superset of the previous one. Since the total number of states is finite, a fixed point exists[1], i.e. at some point $S_{k+1} = S_k$. At this point, no further states are reachable. The set of all reachable states is represented by $S(V) = S_k(V)$. The breadth-first traversal function in algorithm 3.1 repeatedly computes $S_{k+1}(V)$ from $S_k(V)$ to obtain the set of all reachable states. This process converges when no new states can be reached. Note that testing for convergence is easy, since testing BDDs for equivalence is a constant time operation. At each iteration, the set of states visited so far, $S_k(V)$ is updated by adding the set of newly reached states $S_k(V) - S_{k-1}(V)$. In addition, it is verified that both machines produce the same output value. This is done by intersecting the set of newly reached states with the output of the product machine. A result different from constant zero indicates the existence of a miscompare state. Most of the computational effort of the algorithm goes into the image computation of the set of reachable states. Two image computation algorithms will be discussed in section 3.2.

[1] A fixed point of a function is some value x such that $f(x) = x$.

```
function bdd_bfs_traversal (out, init: BDD) : Boolean
    BDD new;       /* Newly reached states */
    BDD reached;   /* States seen so far */
    BDD old_reached;
    BDD current;   /* States to be treated */

    reached := old_reached := current := init;

    do {

        old_reached := reached;

        new := bdd_image (current)
        new := bdd_rename (new);
        new := bdd_and_not (new,reached);
        reached := bdd_or (reached, new);

        if (BDD_0_P (bdd_and (out,new)) return false;

        current := new;
    } while (!BDD_EQUAL_P (reached, old_reached));

    return true;
```

Algorithm 3.1: Breadth-first traversal algorithm

The set of newly reached states are obtained in terms of the next-state variables. In order to update the set of reached states and to use the set $S_k(V) - S_{k-1}(V)$ as a starting point for the next iteration, it is necessary to rename all next-state variables to present state variables. Algorithm 3.2 shows the basic algorithm to perform the substitution of all next-state variables by their corresponding present state variables. The *bdd_rename* function creates a copy of the original BDD by replacing each node (v', T, E) with a node (v, T, E). The function is called at the top level with the root node as argument and the mark fields of the nodes being either all true or all false. The value of a node's mark field is complemented as it is visited in order to avoid solving subproblems more than once. As each node is visited exactly once, the number of recursive calls to *bdd_rename* is $|f|$.

The performance of the *bdd_rename* function can be improved in two ways. First, it is beneficial to use a variable order in which the present and next-state variables are interleaved with corresponding present and next-state variables adjacent to each other. This specific order guarantees that the ranknumber of a variable being substituted is smaller than the ranknumbers of sub-results from recursive calls. As a result, it is possible to perform a call to the *find-or-add* operation instead of a full *ite* computation[2]. Empirically, this provides a substantial savings in time.

[2]The *find-or-add* operation either finds an existing BDD node in the unique table or creates a new node if the given node does not exist.

```
function bdd_rename (BDD f): BDD
    BDD g,h;

    if f = BDD_0 ∨ f = BDD_1
        return f;
    else
        [ f is a triple (v', T, E) ]
        g := bdd_rename (T);
        h := bdd_rename (E);

        return bdd_ite (v,g,h);
    endif
```

Algorithm 3.2: Rename algorithm

A second improvement is to use a result cache. This cache maps the three arguments v', T and E to the result node $R = (v, T, E)$ once it has been computed.

3.2 Image Computation

Image computation is the basic operation in the breadth-first traversal algorithm. Given a set of present states, the set of states reachable in one step is computed using either next-state functions or relations. Several research groups proposed BDD-based techniques for image computation. Here, the methods developed by Burch, Clarke et al. [4] and Touati et al. [19] are discussed.

Touati's technique for computing the image of a set of states uses a conjunction of next-state functions to represent the complete next-state relation. However, they combine this technique with Coudert's *Constrain* operation to restrict the result of the next-state functions to the set of reached states. This reduces the problem of computing the image of a set to that of computing the range of a function. The set S_{k+1} is computed from the set S_k using the expression:

$$S_{k+1}(V') = S_k(V') \vee \exists_{v \in V} \prod_{1 \le i \le n} [v_i' \equiv Constrain(f_i(V), S_k(V))] \tag{3.5}$$

where the image of the set $S_k(V)$ is computed using the product

$$\exists_{v \in V} \prod_{1 \le i \le n} [v_i' \equiv Constrain(f_i(V), S_k(V))] \tag{3.6}$$

The product in expression 3.6 can be computed using a simple iteration loop. However, it is more efficient to use a recursive algorithm which decomposes the Boolean *and* operation of the n equivalence functions $v_i' \equiv Constrain(f_i(V), S_k(V))$ into a binary tree of Boolean *and* operations. The equivalence operations are performed at the leaves of the tree and form the terminal cases of the recursion. After computing a binary *and* on the two sub-results of the recursive calls, say p, variables from the quantification set V which appear

only in p can be quantified out. Notice that this essentially requires a specification of the quantification order of the variables. The quantification and the *and* operation can be done in one pass over the BDDs with an algorithm called *AndExists* [16]. This reduces the storage requirements. Algorithm 3.3 outlines this approach. The function is called at the top level with three BDD vectors f, v' and q as arguments. The first BDD vector, f, represents the vector of next-state functions. The second BDD vector v', represents the next-state variables. The BDD vector q represents the vector of quantification sets. The BDD S_k represents the current set of states.

```
function bdd_image (BDD *f, BDD *v', BDD*q, BDD S_k, int left, int right): BDD
    BDD r,h_1,h_0;

    if (left == right)
        r := bdd_constrain (f[left], S_k);
        return bdd_xnor (v'[left], r);
    else
        int split_pos := (right -left +1)/2;
        h_1 := bdd_image (f, v', q, S_k,left, left + split_pos -1);
        h_0 := bdd_image (f, v', q, S_k,left + split_pos, right );
        return bdd_and_exists (h_1,h_0, q);
    endif
```

Algorithm 3.3: Image computation algorithm

Another BDD based technique to compute the image of a set of states has been reported by Burch, Clarke et al [4]. Their algorithm repeatedly computes the set S_{k+1} from S_k and checks the equivalence of S_k and S_{k+1} in order to determine whether a fixed point has been reached, similar to Touati's approach. However, the fixed point iteration loop is based on an expression which is slightly different from expression 3.4. The set S_{k+1} of states reachable in $k+1$ or fewer steps is given by

$$S_{k+1}(V') = S_0(V') \vee \underset{v \in V}{\exists} [S_k(V) \wedge N(V, V')] \tag{3.7}$$

The image of a set is computed with the relational product

$$S'_{k+1}(V') = \underset{v \in V}{\exists} [S_k(V) \wedge N(V, V')] \tag{3.8}$$

which performs the Boolean *and* operation on the set of states reached so far, S_k, and the next-state relation N together with the existential quantification over all variables in the set V. Note that the set S_k contains all states which have been reached so far, including states reached in previous iterations. As a result, transitions which have been calculated in previous iterations are computed again. Afterwards, the set S'_{k+1} contains all states which are reachable in $k+1$ or fewer steps except for the initial set of states. The set S_{k+1} can be obtained by taking the union of the initial set of states S_0 and S'_{k+1}.

Algorithm 3.4 shows the relational product function *bdd_rel_prod* which can be used to perform the image computation of expression 3.8. The computation is done in one pass over the BDDs $S_k(V)$ and $N(V,V')$. The motivation for this algorithm is to avoid producing the entire BDD for $S_k(V) \wedge N(V,V')$ which is often very large. This is done by applying existential quantification to the results of the subproblems as soon as they become available. Empirically, this provides a substantial savings in space and time. The BDD $S'_{k+1}(V')$ is computed with the call *bdd_rel_prod* $(S_k(V), N(V,V'), V)$.

The performance of the algorithm can be improved by using a result cache. In this case, entries in the cache are of the form (f,g,E,r), where E represents the set of quantification variables and f,g and r are the BDDs. If such an entry is in the cache, it means that a previous call to *bdd_rel_prod (f,g,E)* returned r as its result. The algorithm, as shown, is independent of assumptions on the BDD variable ordering. In section 3.4 we will discuss an improved algorithm for the case that present and next-state variables are interleaved. Other possible implementation details which improve the performance of the algorithm include:

1. If the top variable is being quantified out and the result from the recursive *then* step is true, the result of the *or* operation is true and it is not necessary to do the else recursion.

2. If the top variable among the operands is below the last variable in the set of quantification variables, it is possible to perform the *and* operation on the current operands and return this result.

3. The size of the result cache is extended dynamically if the number of occurrences in the cache exceeds a given upperbound.

The set of quantification variables E can be represented in several ways. Possible options are a linked list, a bit-vector or a BDD. We use a BDD representation. This reduces the test if the cache contains an entry of the form (f,g,E) to simple pointer address comparisons which can be performed in constant time. Speed is an important consideration here as the cache is checked $|S|.|N|$ times in the worst case. The complexity of the relational product algorithm can be stated as $\mathcal{O}(|S|.|N|.2^{2n})$ which is the number of disjunctions to be performed ($|S|.|N|$ in the worst case) times the square of the largest possible BDD size 2^n.

```
function bdd_rel_prod (f,g: BDD, E: set of variables): BDD
    BDD h_0, h_1, r;

    if f = BDD_0 ∨ g = BDD_0
        return BDD_0
    else if f = BDD_1 ∧ g = BDD_1
        return BDD_1
    else if (f,g,E,r) is in result cache
        return r
    else
        let x be the top variable of f
        let y be the top variable of g
        let z be the topmost of x and y
        h_0 := bdd_rel_prod(f|_{z=0}, g|_{z=0}, E)
        h_1 := bdd_rel_prod(f|_{z=1}, g|_{z=1}, E)
        if z ∈ E
            r := bdd_or(h_1, h_0)
        else
            r := bdd_ite(z, h_1, h_0)
        endif
        insert (f,g,E,r) in the result cache
        return r
    endif
```

Algorithm 3.4: Relational product algorithm

3.3 Experimental results

The image computation algorithms described in the previous section have been implemented in C and integrated into the sequential BDD package mentioned in section 2.3.2. This section presents experimental results comparing Burch's relational product method with Touati's approach for image computation. The comparison was done early in the project. At that point both image computation algorithms were implemented without their specific optimizations. More specific, we used the relational product implementation shown in algorithm 3.4. In Touati's algorithm, present state variables were quantified after each image computation instead of on-the-fly. The results obtained from the comparison were used to select one method as the default algorithm for image computation. Table 3.1 shows the results of performing reachability analysis using 23 sequential benchmark circuits from the IWLS'91 benchmark set[3]. The first three columns give the number of state variables, the depth of the STG and the number of reachable states. The next three columns list cpu time, memory, the size of the largest intermediate BDD generated by the *bdd_rel_prod* function and the size of the next-state relation using Burch's algorithm. The last three columns report the cpu time, memory and the size of the largest intermediate

[3]The IWLS'91 benchmark set includes a directory of 40 sequential circuits (smlexamples). This includes the ISCAS'85 and ISCAS'89 benchmarks. The circuits s208.1, s420.1, s838.1 and s9234.1 are corrected with respect to the original versions in the ISCAS'89 benchmark set

BDD generated by the *bdd_image* function using Touati's algorithm. All experiments were run on a IBM RS6000 workstation, model 370, a 60 MIPS machine. The initial state was set to all zeros in each case. The time figures reported include parsing of the input file and building the BDDs for the next-state functions of the state variables. The reported numbers for memory usage represents the maximum amount of memory needed by the BDD package to perform the computations. The figures include overhead resulting from the hash tables used in the BDD package. Both methods use the same variable order. Garbage collection was invoked when more than 25% of the nodes in the unique table were dead. The dynamic variable ordering option was disabled during these experiments. During BDD processing, a maximum of one million BDD nodes was placed on the BDD package, i.e., the memory was limited to approximately 24 Mb. In four cases, (s838.1, s1423, s5378 and s9234.1), it was not possible to build the BDDs for the next-state functions within this limitation. 19 of the 23 example circuits were able to complete both with Burch's and Touati's algorithm. For these examples, the runtime increased by an average[4] of 1.6 when Touati's algorithm was used. This can be explained by comparing the size of the intermediate BDDs generated by both image computation algorithms. For all examples, the intermediate BDDs generated by the *bdd_image* function are larger than the intermediate BDDs generated by the *bdd_rel_prod* function. In addition, Burch algorithm benefits from its result cache. We experienced that for this set of examples, on the average 20% of the calls to the *bdd_rel_prod* function could be handled by the cache. Extending Touati's algorithm with a result cache is not straightforward due to the variable length of the BDD vector arguments

3.4 Improvements

Image computation is a key operation in the sequential verification algorithms discussed in the previous sections. Two refinements to perform image computation more efficiently are discussed in this section. A technique called *partitioned next-state relations* will be discussed in the next chapter.

3.4.1 Frontier Set Simplification

An optimization introduced by Coudert and Madre [5] called *frontier set simplification* can often be used to speed up image computations. This technique tries to reduce the size of the BDD representing the set of states on the search frontier, i.e., the set of states in S_{i+1} but not in S_i. Consider the set S_2 of states reachable in at most two steps from S_0:

$$S_2(V') = S_1(V') \vee \exists_{v \in V} [S_1(V) \wedge N(V, V')] \tag{3.9}$$

Notice that it is also possible to obtain S_2 with an image computation from the set of states on the search frontier $S_1 - S_0$ as this will yield a superset of $S_2 - S_1$ (it may also include some states in S_1):

$$\{s' \mid \exists s \, [s \in S_1 - S_2 \wedge (s, s') \in N] \} \tag{3.10}$$

[4] Averages are computed as arithmetic mean of the ratio computed for each example individually.

				Burch's approach				Touati's approach		
Example	latches	states	depth	cpu (sec)	mem (Kb)	inter (Nodes)	Next-state (Nodes)	cpu (sec)	mem (Kb)	inter (Nodes)
s27	3	6	3	0.1	373	6	10	0.1	373	31
s208.1	8	256	256	0.5	378	13	52	0.9	443	19
s298	14	218	19	0.5	454	59	293	0.6	390	210
s344	15	2625	7	2.7	590	695	637	2.7	589	2529
s349	15	2625	7	1.7	590	694	636	2.7	589	2531
s382	21	8865	151	18.0	594	273	1163	18.0	594	348
s386	6	13	8	0.2	379	14	64	0.7	379	51
s400	21	8865	151	20.0	594	273	1163	18.4	594	348
s420.1	16	65536	65536	161.0	594	34	118	392.0	588	35
s444	21	8865	151	3.5	594	204	388	9.7	796	257
s510	6	47	47	0.5	390	18	150	0.5	390	19
s526	21	8868	151	4.1	594	244	507	11.1	594	360
s641	19	1544	7	4.2	1265	168	2454	10.0	917	13207
s713	19	1544	7	7.9	1265	168	2454	13.4	917	13207
s820	5	25	11	0.5	387	15	107	0.5	387	78
s832	5	25	11	0.5	387	15	107	0.5	387	78
s838.1	32	?	?	-	-	-	-	-	-	-
s1196	18	2616	3	26.0	3337	1103	7983	15.0	3933	29506
s1423	74	?	?	-	-	-	-	-	-	-
s1488	6	48	22	0.6	445	25	179	0.7	445	47
s1494	6	48	22	0.6	445	25	179	0.7	445	47
s5378	179	?	?	-	-	-	-	-	-	-
s9234.1	211	?	?	-	-	-	-	-	-	-
Ratio				1.0				1.6		

Table 3.1: Comparison between Burch's and Touati's image computation algorithm

The set S_2 can be obtained by adding all the states in S_1. Thus, the expression for S_2 can be rewritten as:

$$S_2 = S_1 \cup \{s' \mid \exists s \, [s \in S_1' \wedge (s, s') \in N] \} \tag{3.11}$$

where S_1' is the frontier $S_1 - S_0$. It is sufficient to take any S_1' satisfying $S_1 - S_0 \subseteq S_1' \subseteq S_1$. Given this freedom, we like to choose S_1' so that its BDD representation is small. This can be done using the *generalized co-factor* operation defined in section 2.3. The generalized co-factor function takes two BDDs $S(V)$ and $C(V)$ as input. $S(V)$ can be viewed as the state set and $C(V)$ as the care set. It produces an output BDD $S'(V)$ such that $S(V) \wedge C(V) = S'(V) \wedge C(V)$. That is, $S(V)$ and $S'(V)$ evaluate to the same value for the states in the care set $C(V)$. Intuitively, the representation of the set $S(V)$ is simplified by adding or removing states not in $C(V)$. Usually, the size of $S'(V)$ is less than the size of $S(V)$. Using this idea, the algorithm for computing the set of reachable states is modified as follows. Let S_i be the set of of states reachable after i steps, then

$$\begin{aligned} S_{i+1} &= S_i \cup \{s' \mid \exists s \, [s \in S_i \wedge (s, s') \in N] \} \\ &= S_i \cup \{s' \mid \exists s \, [s \in \mathit{Generalized\_CoFactor}(Si, \neg S_{i-1}) \wedge (s, s') \in N] \} \end{aligned}$$

Notice that using frontier set simplification does not result in memory savings; all of the BDDs in the original reachability algorithm are still computed. The potential advantage of frontier set simplification is that smaller BDDs are used in the image computation. However, the results in table 3.2 indicate that for the set of examples used here, the frontier set simplification technique, combined with Burch's approach, does not lead to a major reduction of computation time.

3.4.2 Improved Relational Product Algorithm

This section describes an optimized relational product algorithm. It can be used if the corresponding present and next-state variables are ordered in pairs. In that case, the next-state to present state renaming can be done during the relational product computation rather than as a separate operation afterwards. Section 3.2 described how the *bdd_rel_prod* function can be used to compute a relational product of the form:

$$S(V') = \underset{v \in V}{\exists} [S(V) \wedge N(V, V')] \tag{3.12}$$

In the standard breadth-first traversal algorithm, this operation is followed by the next-state to present state renaming in order to express the set of reached states in terms of present state variables. A slight modification allows computing the relational product and the substitution in a single recursive pass over the BDDs $S(V)$ and $N(V)$.

Recall from section 3.2 that the basic step in the *bdd_rel_prod* routine is to perform the normal conjunction except when an element of the quantification set V has to be built. In that case, existential quantification is performed. Notice that the resulting BDD $S(V')$ only depends on next-state variables, i.e., all present state variables are existentially quantified. In other words, the conjunction operation generates BDD nodes of the form (v', T, E). Due

				Burch's approach		Frontier set simpl.	
Example	latches	states	depth	cpu (sec)	mem (Kb)	cpu (sec)	mem (Kb)
s27	3	6	3	0.1	373	0.1	373
s208.1	8	256	256	0.5	378	0.6	443
s298	14	218	19	0.5	454	0.5	354
s344	15	2625	7	2.7	590	1.7	589
s349	15	2625	7	1.7	590	1.7	589
s382	21	8865	151	18.0	594	10.3	639
s386	6	13	8	0.2	379	0.2	379
s400	21	8865	151	20.0	594	10.2	641
s420.1	16	65536	65536	161.2	594	173.0	4696
s444	21	8865	151	3.8	594	2.6	594
s510	6	47	47	0.5	390	0.5	390
s526	21	8868	151	4.1	594	3.3	594
s641	19	1544	7	4.2	1265	4.5	1256
s713	19	1544	7	7.9	1265	8.0	1256
s820	5	25	11	0.5	387	0.5	387
s832	5	25	11	0.5	387	0.5	387
s838.1	32	?	?	-	-	-	-
s1196	18	2616	3	26.0	3337	47.4	4696
s1423	74	?	?	-	-	-	-
s1488	6	48	22	0.6	445	0.6	445
s1494	6	48	22	0.6	445	0.6	445
s5378	179	?	?	-	-	-	-
s9234.1	211	?	?	-	-	-	-
Ratio				1.0		0.92	

Table 3.2: Comparison between Burch's algorithm and relational product algorithm using frontier set simplification

to the recursive implementation of the algorithm, where subproblems are solved before the disjunction or conjunction operation and the interleaved variable ordering, it is safe to create a triple (v, T, E) instead of (v', T, E). In this way, the renaming of next-state by present state variables can be avoided. The impact of this optimization has been measured by computing the set of reachable states using Burch's relational product algorithm and our improved relational product algorithm. The results are detailed in table 3.4.2. The benchmark circuits are taken again from the IWLS'91 benchmark set. The runtime for this set of examples is reduced by an average of 32% when our improved relational product algorithm is used. The optimization obviously has more impact if the depth of the STG is larger.

				Burch's alg.		Improved alg.	
Example	latches	states	depth	cpu (sec)	mem (Kb)	cpu (sec)	mem (Kb)
s27	3	6	3	0.1	373	0.1	373
s208.1	8	256	256	0.5	378	0.3	378
s298	14	218	19	0.5	454	0.4	390
s344	15	2625	7	2.7	590	1.0	528
s382	21	8865	151	18.0	594	6.1	594
s386	6	13	8	0.2	379	0.2	379
s400	21	8865	151	20.0	594	6.2	594
s420.1	16	65536	65536	161.2	594	58.4	595
s444	21	8865	151	3.5	594	1.2	594
s510	6	47	47	0.5	390	0.4	390
s526	21	8868	151	4.1	594	1.4	594
s641	19	1544	7	4.2	1265	3.6	1265
s713	19	1544	7	7.9	1265	7.4	1265
s820	5	25	11	0.5	387	0.5	387
s832	5	25	11	0.5	387	0.4	387
s838.1	32	?	?	-	-	-	-
s1196	18	2616	3	26.0	3337	19.9	2689
s1423	74	?	?	-	-	-	-
s1488	6	48	22	0.6	445	0.6	381
s1494	6	48	22	0.6	445	0.6	381
s5378	179	?	?	-	-	-	-
s9234.1	211	?	?	-	-	-	-
Ratio				1.0		0.68	

Table 3.3: Comparison between Burch's relational product algorithm and our improved relational product algorithm.

Chapter 4

Partitioned Next-state Relations

4.1 Problem Statement

In the previous chapter, we discussed the breadth-first traversal technique along with two methods for image computation. Experimental results indicated that the relational product method developed by Burch, Clarke et al [4] performed better than Touati's [19] image computation algorithm. The basic step of Burch's algorithm involves performing relational product computations of the form

$$\exists_{v \in V} \left[S(V) \wedge N(V, V') \right] \tag{4.1}$$

Thus, it is crucial to perform this step as efficiently as possible. For example, a special algorithm has been presented which computes expression 4.1 without building the BDD for $S(V) \wedge N(V, V')$ which could become impractically large. Unfortunately, the BDD representing the next-state relation itself can become very big. Being forced to construct this BDD has been a major limitation in trying to verify complex circuits. In the remainder of this chapter we describe a technique for overcoming this problem by using more than one BDD to represent the next-state relation.

Recall from section 2.2 that the next-state relation of a synchronous sequential circuit can be written as a conjunction of next-state relations

$$N(X, V, V') = N_0(X, V, V') \wedge \cdots \wedge N_{n-1}(X, V, V')$$

Each individual next-state relation $N_i(X, V, V')$ determines the new state of one register as a function of the old state and the inputs. From this expression, it is easily deduced that the number of nodes in the BDD representing the complete next-state relation in the worst case grows as the product of the number of nodes in the individual next-state relations, yielding exponential growth. If the complete next-state relation is represented as a list of implicitly conjuncted next-state relations, the number of BDD nodes is the sum of the nodes in the component next-state relations, yielding linear growth. This observation was made by Burch, Clarke et al. [4] who used the term *partitioned next-state relations* for it.

Since the next-state relation for a synchronous circuit is a conjunction of relations, the relational product has the form

$$S(V') = \exists_{\substack{v \in V \\ x \in X}} \left[S(V) \wedge (N_0(X,V,V') \cdots \wedge N_{n-1}(X,V,V')) \right] \tag{4.2}$$

The main problem in computing $S(V')$, without building the full conjunction of next-state relations, is that existential quantification does not distribute over conjunction. However, it is possible to divide the computation of a full relational product into a sequence of smaller steps by using properties of the conjunction operation.

The basic technique to compute the relational product in equation 4.2 is as follows. Note that the next-state relations can be combined in any order as conjunction is associative and commutative. In addition, although conjunction does not commute with existential quantification, sub-formulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. As a result, it is possible to conjunct the $N_i(X,V,V')$ with $S(V)$ one at a time and quantifying out each variable v when none of the remaining $N_i(X,V,V')$ depend on v. Since quantification tends to reduce BDD size by reducing the number of variables, the strategy is to combine the next-state relations in such an order that the variables can be quantified out as soon as possible. This technique is referred to as *early quantification* [1].

The next-state ordering problem can be defined formally as:

Next-state Ordering Problem (NOP)

Instance: Given a conjunctive partitioned next-state relation with n state variables. In this case, the relational product computation can be written as:

$$S'(V') = \exists_{\substack{v \in V \\ x \in X}} \left[S(V) \wedge N_0(X,V,V') \wedge \cdots \wedge N_{n-1}(X,V,V') \right] \tag{4.3}$$

Minimization Task: Find an ordering of the next-state relations $N_i(X,V,V')$, $i = 0, \cdots, n-1$ which minimizes the intermediate BDD size during the relational product computation.

An ordering of the next-state relations $N_0, \cdots, N_{n-1}$ can be described by the permutation $\rho = \{N_{\rho(0)}(X,V,V'), \cdots, N_{\rho(n-1)}(X,V,V')\}$. The solution space is given by

S = {all permutations π on n next-state relations}

with $|S| = n!$. The problem of computing an ordering that minimizes the size of the intermediate BDDs during the relational product computation is **NP**-hard [8].

[1] The relational product algorithm described in section 3.2 which combines conjunction and quantification in a bottom up manner is also an example of early quantification. Another example includes the use of the *And_Exist* operation in Touati's image computation algorithm.

The permutation describes the order in which the next-state relations $N_i(X, V, V')$ are combined. For each i, let D_i be the set of variables in V and X that $N_i(X, V, V')$ depends on. The set of variables which can be quantified out at the i-th step of a relational product computation is given as:

$$E_i = D_{\rho(i)} - \bigcup_{k=i+1}^{n-1} D_{\rho(k)}$$

Thus, E_i is the set of variables contained in $D_{\rho(i)}$ that are not contained in $D_{\rho(k)}$ for any k larger than i. The E_i are pairwise disjoint and their union is equal to V. Given a permutation ρ, the relational product can be computed as

$$\begin{aligned}
S_1(X, V, V') &= \exists_{x,v \in E_0} \left[S(V) \wedge N_{\rho(0)}(X, V, V') \right] \\
S_2(X, V, V') &= \exists_{x,v \in E_1} \left[S_1(V) \wedge N_{\rho(1)}(X, V, V') \right] \\
&\vdots \\
S(V') &= \exists_{x,v \in E_{n-1}} \left[S_{n-1}(V) \wedge N_{\rho(n-1)}(X, V, V') \right]
\end{aligned}$$

The ordering ρ has a significant impact on how early in the computation state variables can be quantified out. This affects the size of the BDDs constructed and the efficiency of the verification procedure. Thus, it is important to choose ρ carefully. As a practical example consider the next-state relations of the 3-bit counter described in section 2.2. The next-state relation of the least significant state bit depends only on v_0 and v_0'. The next-state relation of the most significant bit depends on v_2' and the whole set of v_0, v_1 and v_2. The set $S(V')$ of states reachable from the set $S(V)$ can be computed as follows:

$$S(V') = \exists_{v_0} \exists_{v_1} \exists_{v_2} \left[\left[\left[S(V) \wedge N_0(V, V') \right] \wedge N_1(V, V') \right] \wedge N_2(V, V') \right] \tag{4.4}$$

Here, the next-state relations are ordered from N_0 to N_2. In this case it is not possible to quantify out any present state variable until the last iteration and a BDD that depends on $2n$ variables is generated. If the next-state relations are ordered from N_2' to N_0', the set $S(V')$ can be computed as

$$S(V') = \exists_{v_0} \exists_{v_1} \exists_{v_2} \left[\left[\left[S(V) \wedge N_2(V, V') \right] \wedge N_1(V, V') \right] \wedge N_0(V, V') \right] \tag{4.5}$$

Now, it is possible to eliminate one state variable per iteration and the maximal number of state variables in the BDD will be $n + 1$. So, without a good order ρ, the relational product computation will usually be much slower and consume more space than using the full next-state relations.

A final remark is that it is usually not beneficial to completely partition the next-state relation. Instead, it is more useful to combine several next-state relations into one BDD. Fewer BDD nodes may be needed in this representation due to the node sharing mechanism. Combining some of the BDDs in a partitioned next-state relation can also speed up the verification algorithms.

4.2 Evaluation

The use of partitioned next-state relations has been evaluated using an encryption circuit called KEY as a benchmark circuit[2]. The choice of this particular circuit enables us to compare our results with the ones reported by Burch, Clarke et al. [4] who describe a similar experiment. The KEY circuit has 228 state variables (*count0* through *count3*, C_0 through C_{111} and D_0 through D_{111}), 258 inputs (*start, encrypt* and *key0* through *key255*) and 193 outputs. The next-state functions for each of the $count_j$ state variables depend on $start, encrypt$ and $count_i$ for $i \leq j$. The next-state function for each of the C_j depend on $start, encrypt, C_j, D_j, count_0$ through $count_3$ and two of the key_i inputs. The same is true for the next-state function for each D_j. The particular support for each state variable indicates that the KEY circuit can be viewed as 113 communicating finite sub-automata. One automata for the $count_0$ through $count_3$ state variables, and for each i from 0 to 111 one automata containing the C_j and D_j variables.

The variable ordering used in the reachable state computation was selected manually. The $start, encrypt$ and $count_i$ variables were put at the top of the ordering as each next-state function depend on them. The C_j and D_j variables were interleaved and the key_i inputs were put near the C_j and D_j that depend on them. With this ordering, it was impossible to construct the BDD for the full next-state relation. However, it was possible to use three partitions: one for the $count_i$ variables, one for the C_j variables and one for the D_j variables. The BDDs for the partitions had 37, 3591 and 5200 nodes respectively. The reachable state set of the circuit contains $1.348 \cdot 10^{67}$ states. It took 1705 seconds to compute this set when a fully partitioned next-state relation was used and 142 seconds for the three partitioned case, a speed up of a factor 12. The runtimes were measured on a 130 Mips IBM RS6000 Model 390. Burch, Clarke et al. [4] reported runtimes (on a Sparc station +1) of 1019 and 41 seconds respectively. These times are obtained using a better variable order. Their BDD representation for the three partitioned case had 33, 2464 and 2566 nodes respectively. Although it is difficult to compare the results directly due to the difference in computer hardware, their result clearly benefits from the better ordering. We expect that a future implementation of dynamic variable ordering which keeps the state variables ordered in pairs will reduce this difference in runtime. The experimental results clearly indicate the benefits of partitioned next-state relations and of recombining individual partitions. In addition, it seems possible to verify synchronous circuits with an extremely large number of states.

4.3 Discussion

The major reason to use the partitioned next-state relation technique is to avoid the exponential growth of the BDD representing the next-state relation. A disadvantage of the technique is that one image computation on the complete next state relation becomes a series of image computations, one for each of the component next state relations yielding slower execution. The use of partitioned next-state relations represents a trade off between

[2]There are actually two sequential benchmark circuits called KEY, one with 228 latches and one with 56 latches. The one with 228 latches is used here.

spending more time or allocating more memory to the image computation. A possible way to control this trade off is to combine some of the $N_i(V, V')$ into one BDD by forming their conjunction. Notice that the technique is somewhat limited as existential quantification only distributes over conjunction in the special case when one of the conjuncts does not depend on the variable being quantified. Nevertheless, there are cases where the support of the component relations is sufficiently disjoint to make this technique effective.

The BDD representation of the next-state relation can be constructed in a straight forward iterative or recursive way. In our current implementation, we monitor the size of the partial generated results and stop the construction if the BDD exceeds a given upper bound (e.g. 5000 BDD nodes). In that case, image computation is performed using partitioned next-state relations.

A final observation is that the trick of combining the relational product computation with the next-state to present state renaming can be used together with partitioned next-state relations due to the disjoint character of the sets of quantification variables. A next-state variable can be renamed if the corresponding present state variable is contained in the current quantification set. Our current research focuses on finding an efficient algorithm to determine orders in which the next-state relations are conjuncted.

Chapter 5

Verification Approach of Verity

Verity is a formal verification tool developed in IBM for the verification of CMOS processor designs. It verifies the logical equivalence between a high-level description of a design such as Verilog and a transistor-level net-list. Flexibility is allowed in the transistor-level implementation as a wide variety of implementation styles including static, dynamic and self-timed logic are supported. Verity has a logic debugger built into it. In case of a verification failure between the high-level specification and the implementation, the debugger will pinpoint regions where the error is most likely.

The current version of Verity addresses the verification of sequential circuits in a strict way. It is based upon a verification model in which corresponding registers must be identified and matched a priori in both designs. This requires that both designs have the same number of registers and use an identical state encoding. As a result, the sequential behaviour of the design can not be verified if the number of registers in both designs do not match or a different state encoding is used. This chapter considers the extension of Verity for sequential verification. Section 5.1 presents the general verification methodology which has been applied within Verity[1]. In section 5.2, we describe how state enumeration techniques can be used for sequential circuit verification and error diagnosis.

5.1 Verification Methodology

The ultimate goal of the functional verification is to achieve exhaustive coverage across the entire design. However, because of the computational complexity, verification algorithms can not be applied directly on the entire chip. Using design partitioning, a two-part hierarchical verification methodology has been developed:

1. The individual pieces of the design (referred to as *macros*) are verified independently. Specific logical boundary conditions associated with macro input and output signals are asserted by the designers. These assertions describe the set of signal patterns which can occur at the inputs of a particular macro. The valid input patterns are referred to as the *care-set*. Input assertions are used as verification constraints, whereas output assertions are validated.

[1]The text in section 5.1 has been reprinted from [11] with permission of the authors.

2. The composition of macros to form the complete design is verified for both correctness and consistency. This essentially checks the integrity of the macro interconnection including the correct wiring and the consistency of the assertions between the individual macros.

Figure 5.1 shows a simple example of a hierarchical design description. A line divides the set of macros in two groups: (1) The set of leaf macros is defined as the set of all hierarchy nodes for which the corresponding sub-circuit can be verified as one piece (macros F, G, H). (2) All remaining macros (A,B, C, D, E) form the set of super-macros which compose the complete design in terms of the set of leaf macros. Functional verification is applied to confirm the correctness of this composition and to check the consistency between all macro assertions.

The basic idea of hierarchical verification is to reduce the complexity of the verification task by excluding instances of sub-circuits from the verification of the calling super-macro. The circuits of the excluded sub-macros are removed from the hierarchical design description and replaced by *black boxes*. For example, when super-macro C of the circuit in figure 5.1c is being verified, leaf macros F, G and H are black-boxed.
The hierarchical verification is controlled by a skeleton which defines all macros for which Verity is actually applied. The complete comparison of two design representations requires an identical verification skeleton on both sides. The verification skeleton shown in figure 5.1a consists of two super-macros (A,C) and the three leaf macros (F,G,H). The resulting five verification tasks for Verity are illustrated in figure 5.1b through figure 5.1d. When verifying a super-macro by black-boxing sub-macros, the following verification steps are performed to ensure completeness:

- All inputs of sub-macros are considered as verification outputs which are, in addition to all primary outputs, functionally compared between the two representations.
- Sub-macro outputs are considered as verification inputs which are driven by independent variables, common for the two design representations.
- Verification constraints asserted at sub-macro inputs are tested on the super-macro level. Since the sub-macros are verified only with respect to those constraints, their test on the higher level effectively validates this assumption.
- Assertions at sub-macro outputs are used to constrain the input space for super-macro verification. The correctness of these assertions is confirmed during sub-macro verification.

The verification view of a particular super-macro $M1$, which calls two instances, $I1$ and $I2$, of sub-macro $M2$, is given in Figure 5.2. Figure 5.2(b) shows the corresponding control files for $M1$ and $M2$ which describes the verification tasks to be performed by Verity. Each control file contains the port definition, which is common to all representations of a particular macro, and other details specific to the representation. For $M1$, these details include the black-boxing directive for both instances of macro $M2$, a constraint for the possible input values, and a test on the outputs. The input constraint describes the care-set for verification, which in this specific example includes all input patterns with at least

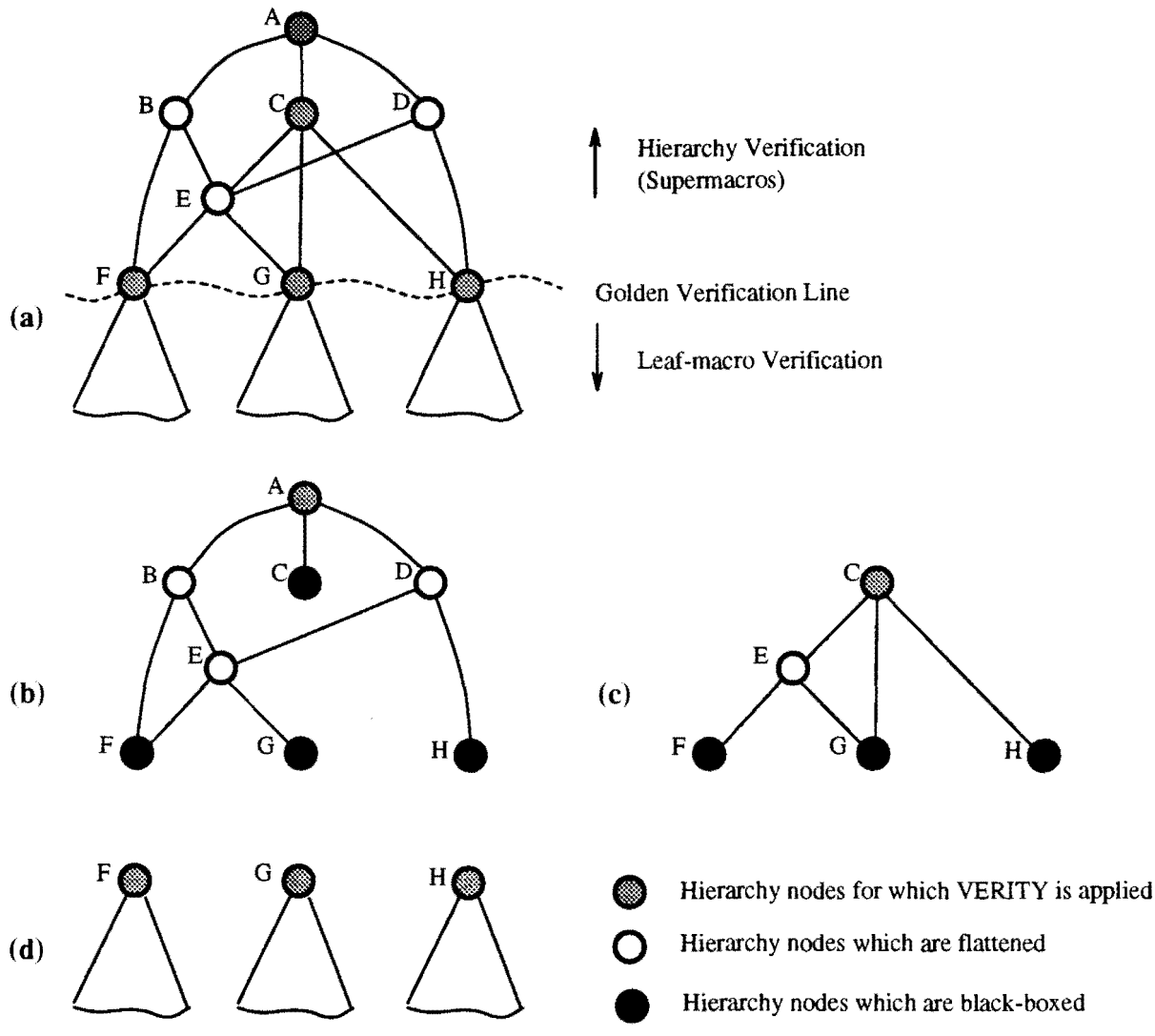
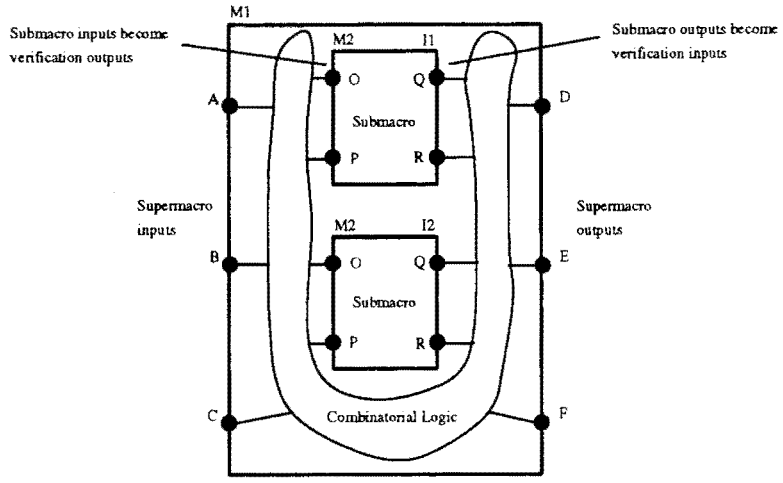


Figure 5.1: Hierarchy example: (a) verification skeleton, (b-d) set of resulting verification tasks.



(a)

(b)

```
MODULE M1;
  INPUT   A, B, C;
  OUTPUT D, E, F;
  IMPLEMENTATION Impl M1;
    DIRECTIVES
      BLACKBOX M1(I1, I2);
      ASSERT A1 := A | B | C;
      TEST    T1 := D | E | F;
    END;
    MODEL M1 (FILE = "m1.spice");
  END;
END;
```

```
MODULE M2;
  INPUT   O, P;
  OUTPUT Q, R;
  IMPLEMENTATION Impl M2;
    DIRECTIVES
      ASSERT A2 := O | ^P;
      TEST     T2 := (Q & ^R) | (^Q & R);
    END;
    MODEL M2 (FILE = "m2.spice");
  END;
END;
```

Figure 5.2: Verification of super-macro $M1$ where two instances of sub-macros $M2$ are black-boxed: (a) hierarchy structure, (b) corresponding control files.

one input having a logical value of 1. Output tests are checked for tautology. The control file for $M2$ is used in a similar way.
The hierarchical verification of the super-macro consists of two tasks:

1. the verification of sub-macro $M2$ proves the equivalence of the various implementations of that macro with respect to the input constraint $A2$. This includes the test for functional equivalence of outputs Q and R and the validation of test $T2$.

2. macro $M1$ is verified with the two instances of $M2$ black-boxed. The black-boxing imposes four additional equivalence tests for sub-macro inputs $I1.O$, $I1.P$ and $I2.O$, $I2.P$ of instances $I1$ and $I2$, respectively. Further, the sub-macro outputs $I1.Q$, $I1.R$, $I2.Q$, and $I2.R$ are treated as independent verification inputs, constrained by the test expressions $I1.T2$ and $I2.T2$.

As already defined, the set of leaf macros consists of all of the sub-circuits of the design hierarchy which can be verified as one unit. Excluding sequential circuit parts, such as latches, these leaf macros are flattened. Thus no restrictions are imposed on their hierarchical description. After flattening, Verity extracts the Boolean function of the

outputs for both design representations and compares them with respect to the input constraints. Sequential circuit elements need to be excluded from the macro verification process. They have to be black boxed and corresponding instances must be matched between the design representations being compared. This requires that both designs have the same number of registers and use identical state encoding. The goal of the work described in this thesis is to extend Verity in such a way that sequential leaf macro circuits can be verified. The restriction of sequential verification to leaf macro cells in combination with a hierarchical verification approach reduces the complexity of the verification task and makes it possible to use the techniques described in chapter 3 in a practical design environment.

5.2 Verification of Sequential Leaf-Macros

In this section, we show how the sequential verification techniques discussed in chapter 3 can be used to verify sequential leaf macro circuits. To simplify the discussion and to focus on the main idea of sequential leaf macro verification, a simple gate-level circuit instead of a transistor-level circuit is used as a practical example. However, the verification procedure described here reflects the fact that Verity specifically addresses transistor-level verification.

The application of Verity to transistor-level designs proceeds in two steps. First, a general path based extraction algorithm is used to extract a functionally equivalent gate-level network from the transistor design. Second, the verification step proves the correct implementation of the circuit against a given specification. The extraction of the gate-level network is based on *functional nets* in the transistor circuit. These nets include all primary inputs, primary outputs and nets which control gates of MOS transistors. The extraction procedure assigns two Boolean functions f^1 and f^0 to each functional net. For a primary input, both polarities of the corresponding input variable are assigned to f^1 and f^0 (e.g. $a^1 = a, a^0 = \overline{a}$). For primary outputs and internal nets, the ON sets of f^1 and f^0 describe the set of input patterns for which the net is driven to VDD and GROUND, respectively. In other words, $f^1(X_i) = 1$ means that there is a path of interconnected transistors from that net to VDD such that all transistors are conducting if pattern X_i is applied at the circuit inputs. In a similar manner, $f^0(X_i) = 1$ denotes some path to GROUND. The verification step compares f^1 and f^0 of each output against the specified functions F^1 and F^0. In addition to the comparison of the circuit outputs, a set of consistency checks can be formulated for each functional net. For example the intersection of f^1 and f^0 detects collisions where the net is driven simultaneously to VDD and GROUND. Similarly, the union of both functions specifies a condition for which a net is floating.
As a practical example, we consider the modulo 8 counter circuit described in section 2.2. This circuit has three state-bits s_0, s_1 and s_2. The subscripts A and B are used to distinguish between the implemented and the specified design, respectively. The two driving functions of the most significant state bit s_3 are

$$f^1 = (s^1_{0_A} \wedge s^1_{1_A}) \oplus s^1_{2_A} \tag{5.1}$$

$$f^0 = \overline{(s^0_{0_A} + s^0_{1_A}) \oplus s^0_{2_A}} \tag{5.2}$$

The corresponding specified functions F^1 and F^0 are defined as:

$$F^1 = (s^1_{0_B} \wedge s^1_{1_B}) \oplus s^1_{2_B} \tag{5.3}$$

$$F^0 = \overline{(s^0_{0_B} + s^0_{1_B}) \oplus s^0_{2_B}} \tag{5.4}$$

The functional equivalence of f^1 against F^1 is established by a call to the function *bdd_seq_equal*. This function accepts two BDD arguments f and g and returns a Boolean value determining the equivalence of its arguments. It can be used for both the combinatorial and the sequential equivalence test. The combinatorial equivalence test is covered by the macro BDD_EQUAL_P (f,g) which performs a simple pointer equivalence check in order to find out if the BDDs f and g are canonical. For the sequential verification case, the logical *xor* of f and g is created. The state variables are extracted recursively from the output function and their corresponding next-state functions. If no state variables are found combinatorial verification is applicable and we can safely return false. The state variables are used to construct the next-state relation and the initial state of the circuit. The *bdd_bfs_traversal* function computes the set of reachable states and determines at each iteration if the product function $f^1 \oplus F^1$ produces a logical 0.

```
function bdd_seq_equal (f,g: BDD): Boolean
    BDD out, next, init, s;

    if BDD_EQUAL_P(f,g)
        return true;
    else
        out := bdd_xor (f,g);
        s := bdd_extract_state_variables (output);
        if (BDD_VOID_P (s)) return false;
        next := bdd_next_state_relation (s);
        init := bdd_get_init_state (s);
        return bdd_bfs_traversal (s,next,init,out);
    endif
```

Algorithm 5.1: BDD function for the functional equivalence test of BDDs containing state variables.

Verity's extraction procedure will perform a call to the *bdd_seq_equal* function for every output comparison or consistency check. During each call, a different subset of the STG of the product machine is traversed, depending on the state variables in the two argument BDDs. The *bdd_bfs_traversal* function uses the improved relational product algorithm described in section 3.4.2 to perform the necessary image computations. This algorithm uses a result cache to improve its performance. In general, it is not necessary to clean the result cache after each fixed point computation. In fact, it is better to keep the results and use them in consecutive computations. We observed a significant caching across different calls to the *bdd_rel_prod* function. For example, when verifying the modulo 8 counter example, the top-level of the *bdd_rel_prod* function was called 125 times. It was possible to return

the correct result in 56 times (46%) directly from the top-level.

The extraction of sequential loops in transistor-level circuits has not been implemented yet. The appendices shows some preliminary verification results obtained from gate-level circuits. Appendix A contains the report file generated by Verity for the verification of the modulo 8 counter example described in section 2.2. Appendix B shows the corresponding control file. Appendices C and D contain the verification results for an 11-bit linear-feedback shift-register.

5.2.1 Error Diagnosis for Sequential Circuits

A formal verification tool which just verifies that an implemented design matches a given correct specification is not very useful in a practical design environment. From a usage point of view, designers are primarily interested in locating and correcting possible design errors. Thus a formal verification tool should provide the user with additional information in case of a miscompare between two designs. For combinatorial circuits this information consists of a set of counter examples in the form of input patterns for the erroneous outputs. For sequential circuits, the information consists of a complete error pattern trace, i.e., a set of error patterns specifying the values of primary inputs and state variables for each particular time frame from the initial state to the state where the miscompare occurs.

The algorithm to generate a sequential error trace proceeds as follows. The equivalence of two BDDs f and g containing state variables can be established by a call to the function *bdd_seq_equal* as described in section 5.2. This function will perform a forward traversal computation to calculate the set of reachable states. The state set $S_k(V)$ which contains exactly the states reachable from the initial state s_0, in k transitions or less, is stored at each iteration. The forward traversal stops whenever a miscompare state is reached. Let us assume that a miscompare state s_k is reached in the k^{th} time step. In that case, a backward traversal computation starting from s_k can be used to report an error trace. The backward traversal iteratively computes a sequence of minterms $(s_{k-1}, \cdots, s_0)$ such that s_{k-1} is reachable from s_{k-2} in one transition. To assure that s_i is in $S_i(V)$ an arbitrary minterm s_i contained in the intersection of the forward and the backward computed state set for that particular time frame is selected. The obtained sequence of minterms $(s_0, \cdots, s_{k-1})$ specifies an error trace leading to the miscompare state s_k.

The algorithm described above is currently being implemented in the sequential BDD package described in section 2.3.2.

Chapter 6

Conclusions and Future Directions

The main subject of this thesis is the development of automatic algorithms for the verification of synchronous sequential circuits. Several state enumeration based verification algorithms have been discussed in chapter 3. Experimental results indicated that the relational product method developed by Burch, Clarke et al. [4] performed better than Touati's [19] image computation algorithm. Additional improvements on the relational product method have been described.

Experimental results with the KEY benchmark circuit demonstrated the benefits of partitioned next-state relations and of recombining individual portions of the component next-state relations. The evaluation also indicated that state enumeration techniques can be used to verify synchronous circuits with an extremely large number of states.

It has also been shown how state enumeration based techniques can be used within Verity to verify gate-level sequential leaf-macro circuits. In addition, an algorithm for the generation of a sequential error trace in case of a verification failure has been described. Finally, a sequential BDD package has been implemented as an extension to the BDD package mentioned in section 2.3.1.

There are several questions that would benefit from future research. The first is the extraction of sequential loops in transistor-level circuits. This has not been implemented yet. As a result, it remains an open question how the algorithms discussed in this thesis perform on transistor-level circuits. A question of particular interest is the size of the transistor-level circuits that could be verified.

Another problem is finding automatic methods for determing efficient orders in which to process and combine parts of the next-state relation. A common problem of BDD-based verification methods is that a good variable ordering must be found. The problem here is that an ordering which minimizes the size of the BDD representing the next-state relation does not necessarily lead to an ordering which is good for the representation of the set of reached states. Ordering strategies for BDDs containing state variables have

been proposed by several authors. Touati [19] suggests an ordering heuristic based on minimizing the cumulative variable support of the latches. Jeong [9] describes an ordering algorithm based on the algebraic structure of the circuit. A recent paper addresses the problem of obtaining good variables orderings for the BDD representation of a system of interacting FSMs [1]. A general disadvantage of these approaches is that they are application specific, i.e., there is no heuristic which always gives good results. We intend to adapt Rudell's sifting algorithm for dynamic variable ordering [18] in such a way that the invariant that corresponding next and present-state variables are created in pairs is not violated.

Bibliography

[1] Adnan Aziz, Sedar Tasiran and Robert K. Brayton,
BDD variable Ordering for Interacting Finite State Machines,
Proc. 31st ACM/IEEE Design Automation Conference,
San Diego, June 1994, pp. 283-288.

[2] Karl S. Brace, Richard L. Rudell and Randal E. Bryant,
Efficient Implementation of a BDD Package,
Proc. 27th ACM/IEEE Design Automation Conference, June 1990, pp. 40-45.

[3] Randal E. Bryant,
Graph-Based Algorithms for Boolean Function Manipulation,
IEEE Transactions on Computers, C-35, 8, Aug 1986, pp. 677-691.

[4] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, D.L.Dill,
Symbolic Model Checking for Sequential Circuit Verification,
IEEE Transactions on CAD of Intergrated Circuits and Systems,
Vol. 13, 4, April 1994, pp. 401-424.

[5] O. Coudert, C. Berthet, J.C. Madre,
Verification of Synchronous Sequential Machines based on Symbolic Execution,
Proc. Workshop on Automatic Verification Methods for Finite State Machines,
June 1989, pp. 365-373.

[6] Michael C. McFarland,
Formal Verification of Sequential Hardware: A Tutorial
IEEE Transactions on Computer-Aided Design,
Vol. 12, 5, 1993, pp. 633-653.

[7] Aarti Gupta,
Formal Hardware Verification Methods: A Survey,
Formal Methods in System Design, Kluwer Academic Publisher, Boston,
Vol. 1, 1992, pp. 151 - 238,

[8] Somesh Jha, Carnegie Mellon University,
personal communication, May 1994.

[9] S.W. Jeong, B. Plessier, G.D. Hachtel and F. Somenz,
Variable Ordering and Selection for FSM Traversal,
Proc. IEEE Int Conf. Computer-Aided Design, 1991, pp. 476-479,

[10] K. Karplus,
Representing Boolean Functions with if-then-else Dags,
Computer Engineering UCSC-CRL-88-28, UC Sanata Cruz, December 1988.

[11] Andreas Kuehlmann, Arvind Srinivasan and David P. LaPotin,
Verity - a Formal Verification Program for Custom CMOS Circuits,
To be published in *IBM Journal of Research and Development*, 1995.

[12] B. Lin, H. J. Touati and A.R. Newton,
Don't care Minimization of Multi-level Sequential Logic Networks
Proc. IEEE Int Conf. Computer-Aided Design, 1990, pp. 414-417.

[13] David E. Long, AT & T Bell Laboraties,
Personal communication, August 1994.

[14] J.C. Madre and J.P Billon,
Proving Circuit Correctness using Formal Comparison between Expected and Extracted Behaviour,
Proc. 25th Design Automation Conference, 1988, pp. 205-210.

[15] Arjen A. Mets,
Dynamic Variable Ordering for BDD Minimization,
Student report, Eindhoven University of Technology,
Department of Electrical Engineering,
January 1994.

[16] Kenneth L. McMillan,
Symbolic Model Checking,
(Kluwer Academic Publishers, Boston, 1993).

[17] S. Minato, N. Ishuira and S.Yajima,
Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation,
Proc. 27th Design Automation Conference, June 1990, pp. 52-57.

[18] Richard Rudell,
Dynamic Variable Ordering for ordered Binary Decison Diagrams,
IWLS'93 Workshop Notes, Draft, May 1993, pp. 3a-1 3a-9.

[19] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli,
Implicit State Enumeration for Finite State Machines using BDD's,
Proc. IEEE Int Conf. Computer-Aided Design, 1990, pp. 130-133.

Appendix A

Report file modulo-8 counter

This appendix shows the report file generated by VERITY for the verification of the modulo-8 counter example described in section 2.2.

```
   +-------------------------------------------------------+
   | Verity 3.1 (C) Copyright IBM Corporation, 1993, 1994 |
   +-------------------------------------------------------+
                Customized version: TEST

Verification results for design "count3" run on Fri Sep 30 09:17:52 1994

Assertion file for golden model    : ./count3.verity
Version of golden model            : VIM
Assertion file for verified model  : ./count3.verity
Version of verified model          : BLA2

count3, BLA2: Correct outputs (0-function)
------------------------------------------
00
01
02

count3, BLA2: Correct outputs (1-function)
------------------------------------------
00
01
02

Verification summary of       : count3 / BLA2
against                       : count3 / VIM

Golden Model                  : count3
Golden Version                : VIM
  Number of Nets              : 4
  Number of Switches          : 0
```

```
  Number of Assertions        : 0
  Number of Tests             : 0

Compare Model                 : count3
Compare Version               : BLA2
  Number of Nets              : 4
  Number of Switches          : 0
  Number of Assertions        : 0
  Number of Tests             : 0

Results of comparison between outputs/cutpoints performed
  Total number of comparisons : 6
  Successful                  : 6
  Failed                      : 0

Results of tests performed
  Total number of tests       : 0
  Successful                  : 0
  Failed                      : 0

Results of consistency checks performed
  Total number of checks      : 12
  Successful                  : 12
  Failed                      : 0


Resulting path statistics for design "count3", version "VIM":
  Total number of paths       : 6
  Number of false paths       : 0
  Number of true  paths       : 6


Resulting path statistics for design "count3", version "BLA2":
  Total number of paths       : 6
  Number of false paths       : 0
  Number of true  paths       : 6

Total memory used             : 832 KBytes
Total CPU seconds             : 10.7 s

Verity Return Code: 0
```

Appendix B

Control file modulo-8 counter

This appendix shows the control file specifying the specification and the implementation of the modulo-8 counter example described in section 2.2.

```
MODULE count3;

  INPUT   IN;
  OUTPUT  O0,O1, O2;

  BEGIN
  IMPLEMENTATION VIM  count3 (MERGE="YES");
    EXTRACTOR "static.chk";

    INPUT   IN;
    OUTPUT  O0,O1,O2;

    DIRECTIVES
    BEGIN
      ORDERING (O0,O1,O2)=(IN);
    END;

    SPECIFICATION
    BEGIN
      O0 := (^O0[-1]);
      O1 := O0[-1] && O1[-1];
      O2 := (O0[-1] & O1[-1]) && O2[-1];
     END;
  END;

  IMPLEMENTATION BLA2  count3;
    EXTRACTOR "static.chk";

    INPUT   IN;
    OUTPUT  O0,O1,O2;

    SPECIFICATION
    BEGIN
```

```
      O0 := (~O0[-1]);
      O1 := O0[-1] && O1[-1];
      O2 := (O0[-1] & O1[-1]) && O2[-1];
    END;
  END;

END;
```

Appendix C

Report file 11-bit linear-feedback shift-register.

This appendix shows the report file generated by VERITY for the verification of an 11-bit linear-feedback shift-register. Appendix D contains the corresponding control file.

```
   +-----------------------------------------------------+
   | Verity 3.1 (C) Copyright IBM Corporation, 1993, 1994 |
   +-----------------------------------------------------+
                Customized version: TEST

Verification results for design "lfsr11" run on Fri Oct 14 23:16:27 1994

Assertion file for golden model    : ./lfsr11.verity
Version of golden model            : VIM
Assertion file for verified model  : ./lfsr11.verity
Version of verified model          : BLA2

lfsr11, BLA2: Correct outputs (0-function)
------------------------------------------
011

lfsr11, BLA2: Correct outputs (1-function)
------------------------------------------
011

Verification summary of       : lfsr11 / BLA2
against                       : lfsr11 / VIM

Golden Model                  : lfsr11
Golden Version                : VIM
  Number of Nets              : 12
  Number of Switches          : 0
  Number of Assertions        : 0
  Number of Tests             : 0

Compare Model                 : lfsr11
```

```
Compare Version               : BLA2
  Number of Nets              : 12
  Number of Switches          : 0
  Number of Assertions        : 0
  Number of Tests             : 0

Results of comparison between outputs/cutpoints performed
  Total number of comparisons : 2
  Successful                  : 2
  Failed                      : 0

Results of tests performed
  Total number of tests       : 40
  Successful                  : 40
  Failed                      : 0

Results of consistency checks performed
  Total number of checks      : 64
  Successful                  : 64
  Failed                      : 0


Resulting path statistics for design "lfsr11", version "VIM":
  Total number of paths       : 22
  Number of false paths       : 0
  Number of true  paths       : 22


Resulting path statistics for design "lfsr11", version "BLA2":
  Total number of paths       : 22
  Number of false paths       : 0
  Number of true  paths       : 22

Total memory used             : 3456 KBytes
Total CPU seconds             : 5707.25 s

Verity Return Code: 0
```

Appendix D

Control file 11-bit linear-feedback shift-register

This appendix shows the control file specifying the specification and the implementation of the 11-bit linear-feedback shift-register. Linear-feedback shift-register circuits are frequently used for internal pseudorandom test pattern generation.

```
MODULE lfsr11;
/* 11 bit lfsr pattern generator: X^11 + X^2 + 1 */
  INPUT   IN;
  OUTPUT  O11;

  BEGIN
  IMPLEMENTATION VIM lfsr11 (MERGE="YES");
    EXTRACTOR "static.chk";

    INPUT   IN;
    NET  O1,O2,O3,O4,O5,O6,O7,O8,O9,O10;
    OUTPUT  O11;

    DIRECTIVES
    BEGIN
      ORDERING (O1,O2,O3,O4,O5,O6,O7,O8,O9,O10,O11)
=(IN,O1,O2,O3,O4,O5,O6,O7,O8,O9,O10,O11);
    END;

    SPECIFICATION
    BEGIN
      O1[0] := 1;  O2[0]  := 1;  O3[0]  := 1;  O7[0] := 1;
      O4[0] := 1;  O5[0]  := 1;  O6[0]  := 1;  O8[0] := 1;
      O9[0] := 1;  O10[0] := 1;  O11[0] := 1;
      O1 := O9[-1] && O11[-1];
      O2 := O1[-1];
      O3 := O2[-1];
      O4 := O3[-1];
      O5 := O4[-1];
      O6 := O5[-1];
```

```
      O7 := O6[-1];
      O8 := O7[-1];
      O9 := O8[-1];
      O10 := O9[-1];
      O11 := O10[-1];
    END;
  END;

  IMPLEMENTATION BLA2 lfsr11;
    EXTRACTOR "static.chk";

    INPUT   IN;
    NET  O1,O2,O3,O4,O5,O6,O7,O8,O9,O10;
    OUTPUT  O11;

    SPECIFICATION
    BEGIN
      O1[0] := 1;  O2[0]  := 1;  O3[0]  := 1;  O7[0] := 1;
      O4[0] := 1;  O5[0]  := 1;  O6[0]  := 1;  O8[0] := 1;
      O9[0] := 1;  O10[0] := 1;  O11[0] := 1;
      O1 := O9[-1] && O11[-1];
      O2 := O1[-1];
      O3 := O2[-1];
      O4 := O3[-1];
      O5 := O4[-1];
      O6 := O5[-1];
      O7 := O6[-1];
      O8 := O7[-1];
      O9 := O8[-1];
      O10 := O9[-1];
      O11 := O10[-1];
   END;
  END;

END;
```