

# Verity - a Formal Verification Program for Custom CMOS Circuits \*

Andreas Kuehlmann      Arvind Srinivasan      David P. LaPotin

IBM Thomas J. Watson Research Center  
Yorktown Heights, N.Y.

## Abstract

In an effort to fully exploit CMOS performance, custom design techniques are used extensively in commercial microprocessor design. However, given the complexity of current generation processors and the necessity for manual designer intervention throughout the design process, proving design correctness is a major concern. In this paper we discuss Verity, a formal verification program for symbolically proving the equivalence between a high-level design specification and a MOS transistor-level implementation. Verity applies efficient logic comparison techniques which implicitly exercise the behavior for all possible input patterns. For a given register-transfer level (RTL) system model, which is commonly used in present-day methodologies, Verity validates the transistor implementation with respect to functional simulation and verification performed at the RTL level.

---

\*Copyright ©1994 International Business Corporation  
This document has been published in the IBM Journal on Research and Development, January 1995.

# 1 Introduction

The design of complex digital systems requires verifying the correctness of the implementation with respect to the intended function. For example, large computer designs integrating many individual circuit components must be checked for numerous characteristics including static function, timing, testability, and manufacturability. A complete verification strategy is not only important for lower development cost and shorter design duration, it is a prerequisite for successful system design.

A verification technique proves a set of user-defined design properties in terms of specific modeling criteria. The accuracy of the model and the complexity of the algorithms determine the practical limitations of a given technique. Typically, the trade-off between accurate results and efficient usage leads to a range of different verification methods applied at different levels of abstraction. Techniques for verifying detailed models of smaller circuit pieces are complemented by more abstract methods working on a larger scale. This hierarchical approach is especially important for practical usage of verification algorithms with exponential computing time or memory complexity.

To validate the functional behavior of large system designs, the complete system is usually modeled at an abstract level and exposed to the intended environment by simulation. For microprocessors, simulation typically includes executing a kernel subset of the designated operating system, running selected software applications, or testing random sequences of processor instructions. Much effort has been spent in improving the coverage of the validation process by accelerating existing algorithms for software simulation [1], applying hardware simulation techniques [2], or using prototype implementations based on programmable logic devices. Recent research in the verification area is focused on applying formal techniques for testing higher-level system properties [3]. However, except for limited results, no breakthrough for general applications has yet been made.

After validation through simulation, the abstract model becomes the definition of the intended system function. Therefore, this model is often referred to as the *golden specification*. Starting with this high-level model a detailed *implementation* can be derived automatically, manually, or (as is often the case) by a combination of both methods. A design technique based on automatic synthesis significantly limits the possible implementation styles and therefore compromises the area and timing performance of the results. However, assuming that the applied algorithms are correct, synthesis preserves the functionality of the abstract specification. Besides reducing the design time, this is a major advantage of automated implementation techniques. Optionally, functional verification of the final design is applied to confirm the correctness of the synthesis algorithms.

To maximize the performance of CMOS processors, custom design exploits elaborate, manual circuit and layout techniques. Since the circuit design is done independently of the golden specification, a separate functional verification step for the final implementation is necessary. There are two approaches to custom circuit verification:

1. The system-level simulation is repeated on the switch-level model of the CMOS circuit. The smaller granularity of this model causes a significant increase in simulation complexity. This drastically reduces the number of simulation patterns which can be applied in a given time, resulting in a corresponding decrease in overall verification

coverage. This problem has been addressed by abstracting a gate-level model from the transistor representation [4, 5, 6, 7, 8] and by using hardware accelerators for switch-level simulation [9, 10]. However, even with these improvements, the repeated functional simulation on the circuit level is highly CPU-time-intensive and impractical in an interactive debugging environment.

2. The functional behavior of the transistor-level implementation and the high-level specification are exhaustively compared. Formal verification techniques can be used to symbolically prove the equivalence between the input/output behavior of two circuit representations for all possible input sequences. The comparison indirectly validates the transistor-level implementation with respect to all results obtained from the functional simulation of the specification.

In this paper we present Verity, a formal verification tool which follows the second approach. It is applicable for gate-level designs as well as for transistor-level circuits with a wide variety of implementation styles including static and dynamic techniques. Verity is part of the design methodology for several microprocessors developed within IBM, including PowerPC<sup>1</sup> implementations. It is being used for the following verification tasks:

- Verity compares the CMOS circuit implementation of the system hierarchically with the high-level specification. Although the specification is declared as golden, in practical design scenarios miscompares typically uncover errors in both representations. Therefore, besides checking the implementation, formal comparison also provides additional confidence in the correctness of the golden model.
- Verity performs a variety of consistency checks on the transistor circuits. These tests verify that a specific design style is obeyed. For example, floating net conditions can be detected in which the logical state of a net is undefined.
- Verity can use and test functional boundary conditions provided as logical assertions by the designers. For example, an orthogonality assertion might be imposed on the input signals of a circuit for which exactly one signal is active at a given time. Verity uses these assertions to constrain the verification process. Further, the circuit generating these input signals is tested for orthogonality to check whether the assumption is correct.

The verification of sequential designs by Verity requires the same state encoding for the circuit implementation and the high-level specification. Essentially, the sequential circuit elements of both sides must be identified and matched.

The paper is structured as follows: In the next section, the verification problem is characterized, and previous approaches in this area are summarized. In subsequent sections the general verification methodology for applying Verity is outlined and specific extraction algorithms are discussed. Finally, results and conclusions are presented.

---

<sup>1</sup>PowerPC is a trademark of International Business Machines, Incorporated.

## 2 Verification Problem

The verification of transistor-level circuits can be divided into two subproblems. The first problem is to extract a Boolean interpretation of the transistor-level network. The second problem is the verification of the Boolean representation by some formal method. In this section we characterize both subproblems and introduce preliminary concepts needed for their solution.

### 2.1 Logic Verification of Boolean Networks

We refer to formal verification as a technique which exhaustively proves certain functional design properties. For example, formal verification might be used to show the equivalence of two circuit representations. We limit our discussion to static functional behavior, neglecting any delay of circuit elements. Initially, we also assume that the circuit representations are based on a synchronous single-clock, finite-state machine (FSM) model. Extensions of the basic model to more elaborate design styles such as multiphase dynamic circuits are discussed in the subsection on time-sliced extraction for dynamic circuits.

Assume that two FSMs,  $A$  and  $B$ , are to be compared. Intuitively,  $A$  and  $B$  are functionally equivalent if they have an identical interface and if, from a given pair of equivalent initial states, they produce the same sequence of output values for any valid sequence of input values. Figure 1 illustrates the equivalence check for two synchronous FSMs. Let  $C^A$  and  $C^B$  denote the combinatorial part and  $S^A$  and  $S^B$  denote the set of state registers of machines  $A$  and  $B$ , respectively. Further, let  $\underline{x} = \{x_1, \dots, x_n\}$  be the set of inputs,  $\underline{y} = \{y_1, \dots, y_m\}$  be the set of outputs,  $\underline{z} = \{z_1, \dots, z_k\}$  be the set of present-state variables and  $\underline{z}' = \{z'_1, \dots, z'_k\}$  be the set of next-state variables. Superscripts  $A$  and  $B$  distinguish between the two machines.  $X[t]$ ,  $Y[t]$ ,  $Z[t]$ , and  $Z'[t]$  are used to denote the vectors of values at clock cycle  $t$  for the inputs, outputs, present-state variables, and next-state variables, respectively.

For the sake of functional comparison, a product machine  $A \times B$  is built. Inputs  $\underline{x}^A$  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 combined to form signal  $c$ . The two machines are said to be functionally equivalent if and only if, after  $S^A$  and  $S^B$  are initialized to their corresponding initial states  $Z^A[0]$  and  $Z^B[0]$ , respectively, any input sequence  $(X[0], \dots, X[t])$  produces a constant value of 0 at  $c$ .

If FSMs  $A$  and  $B$  do not have state registers, each circuit implements a combinatorial function where the output values  $Y[t]$  do not depend on past input values  $X[t-i]$ ,  $1 \leq i \leq t$ . In this case, successful comparison of the two circuits for a single clock cycle proves their equivalence for any input sequence. This case is classified as *combinatorial logic verification*. The more general case, where  $A$  and  $B$  contain arbitrary state registers, is referred to as *sequential logic verification*.

#### 2.1.1 Combinatorial Logic Verification

Most published work in the area of combinatorial verification can be classified into two basic approaches:

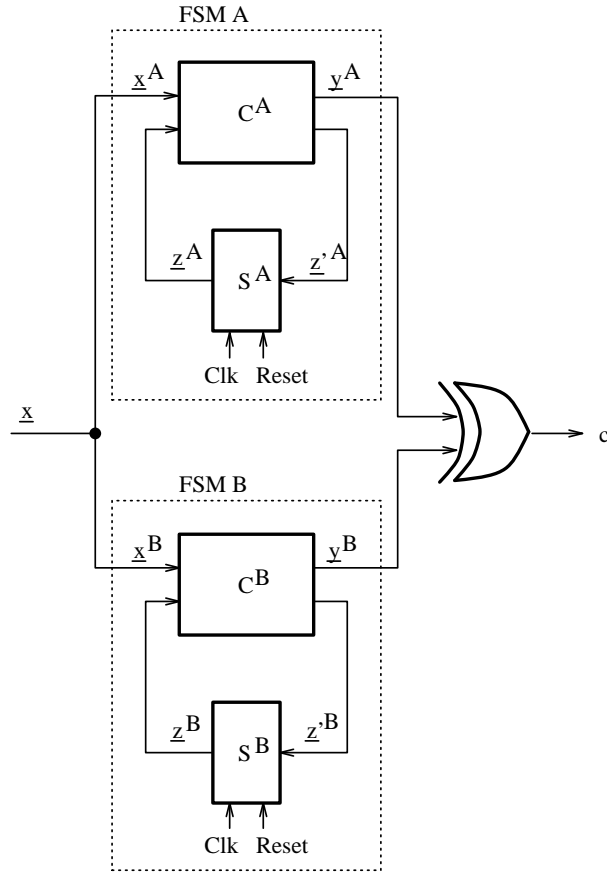


Figure 1: General configuration to prove the equivalence between two FSMs.

1. In the first approach, the Boolean function of all outputs of  $C^A$  and  $C^B$  is converted into some unique (canonical) form [11, 12, 13]. A structural comparison of this unique representation is used to draw conclusions about their functional equivalence. Since the worst-case size of canonical representations of Boolean functions grows exponentially with the number of inputs, excessive memory requirements limit their applicability to general Boolean functions.
2. The second approach is adopted from test pattern generation. An input pattern is determined which causes conflicting output values for the two circuits [14, 15, 16]. If a pattern does not exist, the circuits are functionally equivalent. The search for a counter-example is performed on the circuit structure. In the worst case, this technique may require an exhaustive enumeration of all possible input patterns, thus causing run-time problems for general network structures.

One of the first practical programs used in industry for verifying large logic designs was SAS (Static Analysis System) [17]. SAS is based on the DBA (Differential Boolean Analyzer) and ESP (Equivalent Sets of Partials) algorithms, which are similar to unordered BDD (Binary Decision Diagrams) [11] in their unreduced and reduced forms, respectively.

Although SAS had significant restrictions on the applicable design size, it was successfully applied to complex computer designs within IBM [18].

The Reduced Ordered Binary Decision Diagram (ROBDD) developed by Bryant [19] is one of the most popular canonical structures for representing Boolean functions. The reasons for the success of ROBDDs are their compact structure, which can be manipulated efficiently, and their wide applicability for many practical problems. Ordered BDDs employ a global ordering of the input variables. Depending on the function, the ordering sequence greatly affects the total size of the overall data structure. Therefore, heuristics based on the circuit structure are often used to determine a good ordering before BDD construction starts [20]. Other approaches incorporate ordering algorithms into the BDD software by dynamically reordering the variables during construction of the BDD [21]. Various modifications of the basic BDD structure attempt to enlarge the set of Boolean functions for which BDDs can be efficiently built [22].

Another fundamental approach for comparing combinatorial functions uses probabilistic methods by hashing Boolean functions to integer values [13, 23]. Since this is a one-to-many mapping, unequal functions might be recognized as equal. The probability of false positives can be greatly reduced by repeating this process with different hashing functions. One problem with probabilistic approaches is the calculation of the hash value. It can be done with polynomial time complexity only if the network representation of the Boolean function complies with certain constraints. However, obtaining this specific structure might be as complex as building an equivalent BDD representation.

### 2.1.2 Sequential Logic Verification

There are a variety of approaches for functional verification of general machines. A comprehensive overview can be found in [24, 25]. The different techniques fall 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 [26, 27]. They iteratively modify the hypothetical theorem by applying axioms or other previously proven theorems until it becomes tautologic. Because of their generality, these methods can model almost any behavioral system property. However, these approaches also require a great deal of user knowledge and guidance in order to successfully verify practical designs.
2. State exploration techniques follow a bottom-up approach by explicitly or implicitly visiting all reachable states of the product machine  $A \times B$  [3, 28, 29]. For all outgoing transitions from these states, they check that output  $c$  is logically constant zero, thus proving the functional equivalence of  $A$  and  $B$  with respect to the pair of initial states. The introduction of BDDs for representing sets of states combined with a symbolic depth-first or breadth-first traversal of the FSM graph made this approach applicable for designs with a large number of states [29, 30]. In contrast to theorem proving, state exploration techniques can be automated. This 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 their general

application. This is a major problem since, other than dynamic variable ordering, no practical variable reordering technique for this application is known.

If the two FSMs to be compared use the same state encoding, with a known one-to-one correspondence between the state bits  $(z_1^A, \dots, z_k^A) \rightarrow (z_1^B, \dots, z_k^B)$ , the sequential verification task becomes far more tractable. In this case, the following three steps inductively prove the equivalence of the two machines:

$$\begin{aligned} Z^A[0] &= Z^B[0] , \\ \forall X[t] : Z^A[t] &= Z^B[t] \Rightarrow Y^A[t] = Y^B[t] , \\ \forall X[t] : Z^A[t] &= Z^B[t] \Rightarrow Z^A[t+1] = Z^B[t+1] . \end{aligned}$$

This effectively reduces the sequential verification problem to the comparison of the combinatorial functions implemented by  $C^A$  and  $C^B$ . In other words, if the machines start from the same initial state, a sufficient condition for the equivalence of  $A$  and  $B$  is the equivalence of their next-state and output functions:

$$\begin{aligned} \underline{z}'^A(\underline{z}^A = \underline{z}, \underline{x}^A = \underline{x}) &= \underline{z}'^B(\underline{z}^B = \underline{z}, \underline{x}^B = \underline{x}) , \\ \underline{y}^A(\underline{z}^A = \underline{z}, \underline{x}^A = \underline{x}) &= \underline{y}^B(\underline{z}^B = \underline{z}, \underline{x}^B = \underline{x}) . \end{aligned}$$

## 2.2 Functional Circuit Extraction

In order to perform a Boolean equivalence check against a transistor level implementation, a functional extraction step is necessary. This can be approached in two different ways:

1. Using graph isomorphism algorithms, known substructures of the original circuit are identified and replaced with an equivalent Boolean network [31].
2. The Boolean behavior of the MOS circuit is completely calculated on the basis of a switch-level model [6].

For a practical application of the first approach, a complete set of known circuits must be maintained. Unidentified structures remaining in the circuit would require manual translation into a Boolean model. This technique is applicable for library-based design methodologies. However, it is not adequate for custom design styles where each circuit piece is specifically designed for the intended function and exotic circuit structures are often used.

The switch-level model for MOS circuits has become quite popular in simulation applications because of its practical trade-off between accuracy and efficiency. The model is based on an undirected graph, where the nodes and the branches model the nets and the MOS transistors, respectively. The nodes and branches are weighted by strength values which reflect the net capacities and the driving conductance of transistors. The steady-state response of the MOS circuit for a particular input stimulation is expressed by digital values (usually  $\in \{0, 1, X\}$ , where  $X$  is unknown) assigned to the nodes and branches. The subsequent calculation of these values for a sequence of input patterns models the digital behavior of the actual circuit.

An elegant definition of a switch-level model was introduced by Bryant and used in the simulator MOSSIM II [32]. It restricts the possible strengths of nodes and branches to a bounded set of integer values and applies simplified operations for calculating the combined strengths of interconnected branches. For parallel connections, the maximum-strength value is taken. Similarly, for serial connections the minimum-strength value is applied. The application of that model and a clever grouping scheme for the strengths of nodes and branches results in a highly efficient simulation algorithm.

Several techniques based on switch-level simulation algorithms have been adopted for deriving a functionally equivalent Boolean network from a transistor-level representation. These approaches extend the applicability of gate-level simulation techniques to the switch-level domain [5, 7].

A straightforward approach for switch-level extraction can be formulated by introducing independent Boolean variables for each circuit net. The circuit branches establish relations among these variables which can be modeled by a system of Boolean equations. The solution of that system encodes the complete static behavior of the circuit. However, because of the resulting large number of variables and the inability of handling node and branch strengths, this approach has limited applicability to practical design verification.

Path-based extraction techniques [4, 6, 7] use the concept of driven and controlling nets in transistor circuits. Driven nets include primary inputs or internal nets which hold charge. The set of controlling nets consists of all primary outputs and nets which directly drive the gate of some transistor. Path-based extraction explicitly or implicitly enumerates all transistor paths from driven nets to controlling nets. ANAMOS [6] employs a path-based extraction scheme which is fully consistent with Bryant's switch-level model developed for MOSSIM II [32]. Based on implicit path traversal, it generates a gate-level circuit representation which models the entire switch-level behavior. ANAMOS is applied primarily for gate-level simulation of transistor circuits [33, 34].

A direct approach for functional verification of transistor circuits, described in [35], is based on ANAMOS for switch-level extraction. A state register is assigned to each functional circuit net, resulting in an FSM which models the entire sequential circuit behavior. After computing the steady-state response for a given set of input values, state enumeration techniques are applied for the verification step. However, the generality of this approach compromises the applicable circuit size.

## 3 Verification Approach of Verity

### 3.1 Overview

Verity was designed for the functional verification of large CMOS designs such as complete microprocessor systems. It uses ROBDDs to represent the Boolean function of the networks being compared. Various ordering algorithms working on the circuit structure, in conjunction with dynamic variable ordering, are applied to manage the storage requirements of the BDDs.

Verity does not address the general sequential verification problem. It is based on a verification model in which corresponding state registers are to be identified. Although this

generally limits the applicability of the program, the restriction to combinatorial equivalence enables the verification of more complex circuits. Further, because of the maturity of combinatorial verification, it significantly improves the ability to predict whether a given circuit block can be handled by the verification tool, thus simplifying the overall design partitioning process.

Verity employs a general data representation for mixed circuit designs at the gate level or switch level. A general path-based functional extraction algorithm can handle any combination of these models. The extraction step is tightly coupled with the actual verification step. This makes it possible to efficiently handle special circuit structures such as pass-transistor logic, false CMOS paths, or circuits which contain combinatorial loops. Further, circuit structures which potentially violate the combinatorial verification model, such as dynamic circuit nets or structural network loops, are modeled and handled in a general way. The approach can immediately identify whether a given circuit causes undesired sequential circuit behavior.

The verification method for a particular circuit design style is fully customizable by a user defined set of extraction rules. For example, rules are used to specify the detailed clocking scheme used in dynamic circuit implementations. The rule set also includes tests for unwanted circuit situations, such as nets which might have floating or undefined states. Because of this flexible verification approach, Verity can be adapted to the specific requirements of various projects. Verity is also capable of handling circuits using ratioed logic (transistor strength dependent logic). The corresponding algorithms are not covered here.

## 3.2 Verification Methodology

The ultimate goal of functional verification is to achieve exhaustive coverage across the entire design. However, because of their computational complexity, verification algorithms cannot be applied directly on the entire chip. Using design partitioning, a two-part verification methodology for the use of Verity 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. Often valid input patterns are also referred to as the *care-set*. Input assertions are used as verification constraints, whereas output assertions are validated.
2. The composition of macros to form the complete design is verified for both correctness and consistency. This essentially checks the integrity of macro interconnection, including the correct wiring and the consistency of the assertions between the individual macros.

The following two sections elaborate on these two verification steps.

### 3.2.1 Hierarchical Design Verification

Because of the complexity associated with Boolean function representation, Verity cannot handle large systems as one piece. Therefore, the verification of complex circuits requires an identical partitioning of the two design representations being compared. From a verification point of view, the granularity of the partitioning must guarantee that each piece successfully passes Verity. This process can be referred to as *design for verification*. The two representations are usually developed independently. Changes made to the high-level model typically invalidate the results of the functional system validation. Also, any circuit modification at a late stage might cause significant effort to update the results of timing verification or layout implementation. Therefore, an early confirmation of the applicability of Verity to all pieces of the design partitioning is advantageous.

Verity is usually applied at an early stage to confirm that the circuit partitioning can be handled. Applicability to a set of commonly used macros, including 64-bit data-path units such as adders, shifters, and rotators, is predictable. Depending on the function, macros containing up to 25,000 transistors can be handled by Verity. The sequential verification problem was explicitly excluded to avoid additional uncertainty. Further, Verity can be applied to test the BDD construction for a single design representation. For example, an early version of the macro specification is usually available before the implementation. Successful BDD construction for the specification practically implies that a comparison with any correct implementation is feasible.

Figure 2(a) shows a simple example of a hierarchical design description. A line divides the set of macros into two groups: 1) The set of leaf macros is defined as the set of all hierarchy nodes for which the corresponding subcircuit can be verified as one piece (macros F, G, H). For the sake of functional verification, these subcircuits are completely flattened. This grants complete freedom for their hierarchical description. 2) All remaining macros (A, B, C, D, E) form the set of supermacros 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. Besides calls to other macros, supermacros might also contain actual logic.

The basic idea of hierarchical verification is to reduce the circuit complexity by excluding instances of submacros from the verification of the calling supermacro. The circuits of the excluded submacros are removed from the hierarchical design description and replaced by *black boxes*. For example, when supermacro C of the circuit in Figure 2(c) is being verified, leaf macros F, G, and H are black-boxed.

The hierarchical verification is controlled by a *verification skeleton*, which defines all macros for which Verity is actually used. Clearly, the complete comparison of two design representations requires the same verification skeleton on both sides. Depending on the overall design methodology, Verity can be applied on each skeleton macro in a top-down or bottom-up manner. As shown in Figure 2(a), the verification skeleton consists of two supermacros (A, C) and the three leaf macros (F, G, H). The resulting five verification tasks for Verity are illustrated in Figures 2(b) through 2(d).

When verifying a supermacro by black-boxing submacros, the following verification steps are performed to ensure completeness:

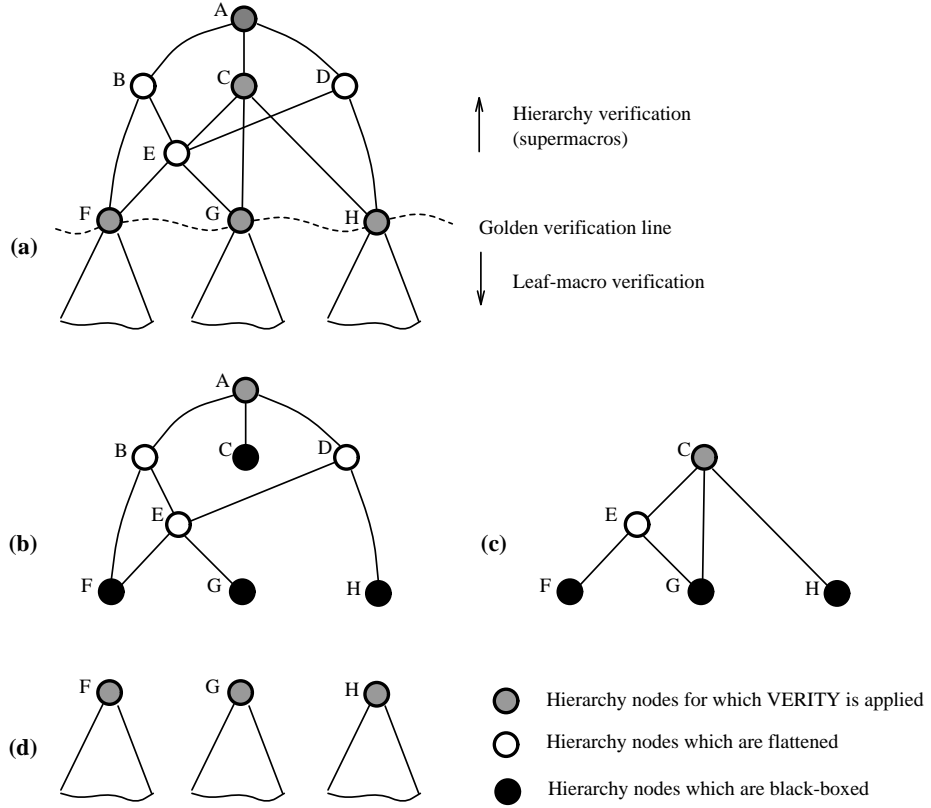


Figure 2: Hierarchy example: (a) verification skeleton, (b-d) set of resulting verification tasks.

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

The verification view of a particular supermacro  $M1$ , which calls two instances,  $I1$  and  $I2$ , of submacro  $M2$ , is given in Figure 3. Figure 3(b) shows the corresponding control files for  $M1$  and  $M2$  describing 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 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.

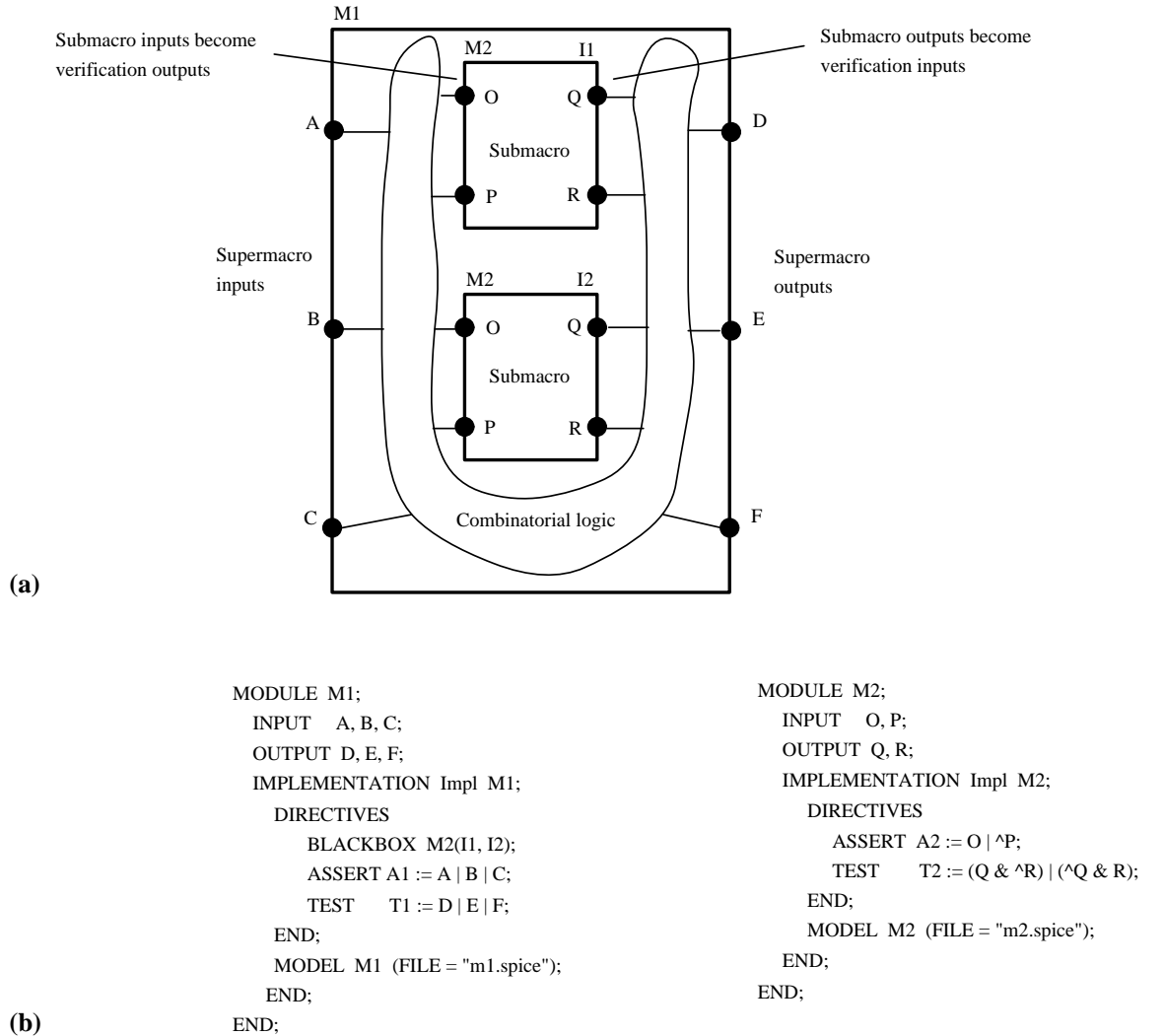


Figure 3: Verification of supermacro  $M1$  where two instances of submacros  $M2$  are black-boxed: (a) hierarchy structure, (b) corresponding control files.

The hierarchical verification of the supermacro consists of two tasks: First, the verification of submacro  $M2$  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  $T2$ . In a second step, macro  $M1$  is verified with the two instances of  $M2$  black-boxed. The black-boxing imposes four additional equivalence tests for submacro inputs  $I1.O$ ,  $I1.P$  and  $I2.O$ ,  $I2.P$  of instances  $I1$  and

$I2$ , respectively. Further, the submacro outputs  $I1.Q, I1.R$  and  $I2.Q, I2.R$  are treated as independent verification inputs, constrained by the test expressions  $I1.T2$  and  $I2.T2$ .

### 3.2.2 Leaf Macro Verification

As defined previously, the set of leaf macros consists of all of the subtrees of the design hierarchy which can be verified as one unit. Except for sequential circuit pieces, such as latches or registers, these leaf macros are completely flattened. Thus, no restrictions are imposed on their hierarchical description. After flattening, Verity extracts the Boolean function of the outputs for the two design representations and compares them with respect to the input constraints. The details of the circuit extraction algorithm are described in the following section.

Sequential circuit elements or other design pieces which cannot be modeled by Boolean logic, such as analog or semi-analog circuits, must be excluded from the macro verification process. These subcircuits are black-boxed, and corresponding instances of the two design representations are identified. The black-boxing scheme is identical to that used for hierarchical verification.

A customized design style necessitates hand-crafted implementations of most of the chip, including storage elements. To optimize performance for each individual instance, the designer usually makes modifications to these circuits which might affect the corresponding interface or the functionality of the circuit implementation. However, once the validation of the golden specification is finished, it is frozen, and interface modifications at a late stage are unacceptable. Thus, for practical design projects a general mechanism for matching differences between interfaces is needed. To meet this practical requirement, Verity allows the user to specify any combinatorial relation between the interfaces of different implementations.

A possible matching of two representations for a simple latch example is shown in Figure 4. They differ in two aspects: First, the specification (*Spec*) includes the gating of the global clock signal *Sys\_clk* by a separate select signal *Select*. In the implementation (*Impl*), the clock gating is handled externally, which virtually moves the circuit interface into the latch. Second, the implementation generates two polarities for the output signal *Out*, whereas the specification produces only one.

To match the different interfaces between representations, a generic method of black-boxing is used in Verity. The idea is to map the actual interfaces of the various implementations to a common generic interface by applying user-defined combinatorial *glue logic*. For each macro this combinatorial glue logic must be specified by the designer. During verification, the glue logic block replaces the actual black box producing the generic interface common to all implementations. Figure 4(b) shows the Verity control file which contains the user-specified glue logic in the INTERFACE section.

### 3.3 Functional Extractor

Verity uses a unified data structure for storing both gate-level and switch-level circuits. The circuit representations being compared, the given input constraints, and any additional out-

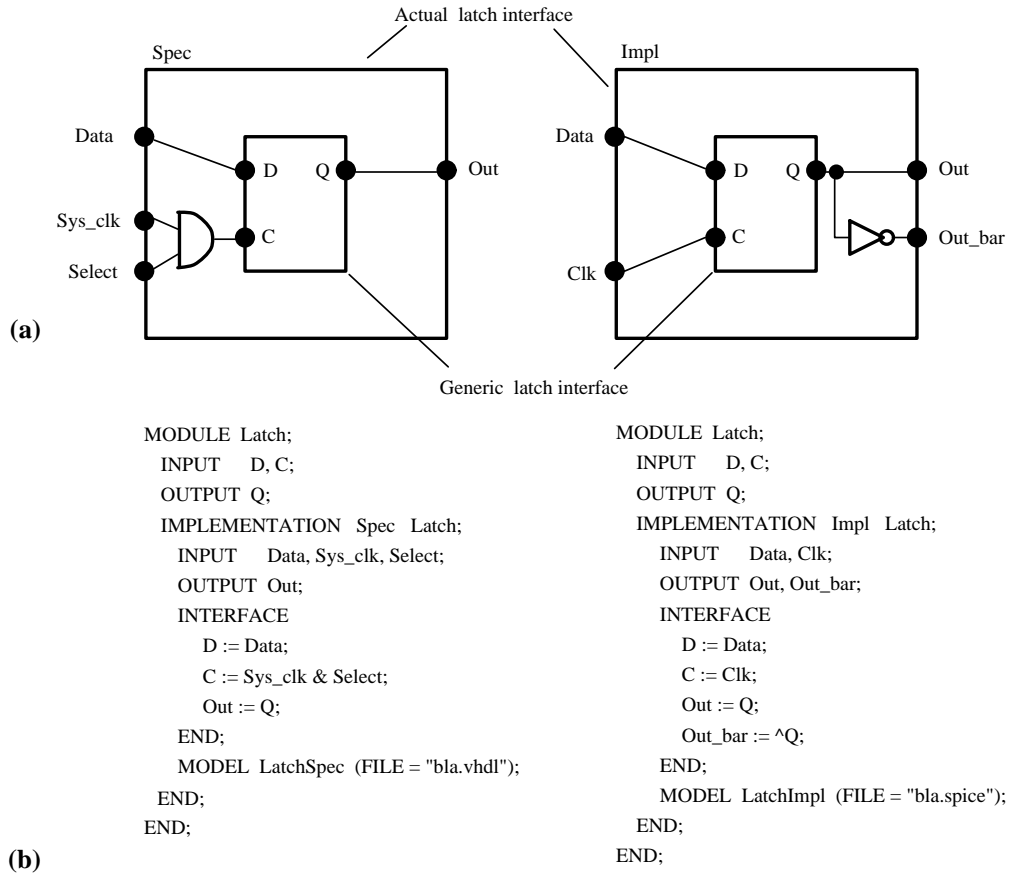


Figure 4: Application of glue logic to match two different latch interfaces: (a) glue logic structure for two latch representations, (b) corresponding control files.

put tests are converted into that common data representation. This allows the extraction algorithm to work on the unified representation without distinguishing between the actual verification tasks to be performed. Moreover, this general extraction approach can be applied to mixed representations where gate-level designs include transistor-level circuits, and vice versa. This feature is useful for incremental verification of incomplete design implementations. In other words, given a complete specification, Verity can be applied to partial circuit implementations where the missing design parts are replaced by the specification. Details of the extraction technique are described in the following sections.

### 3.3.1 Path-based Extraction Scheme

A channel-connected component is defined to be the maximal set of transistors and nets such that every net in the component is reachable from every other net by traversing source-drain connections of transistors within this component.

Verity uses an explicit path enumeration for extracting the Boolean function of a channel-connected component. For verification, the function of all channel-connected com-

ponents must be computed with respect to the primary inputs. This is done in a recursive manner, starting from the outputs. The advantage of this approach is that false paths are eliminated during traversal because the functions at controlling transistor gates along that path are known with respect to inputs.

In contrast, the implicit enumeration approach of ANAMOS is applied to a single channel-connected component. It can be used to extract the function even if the number of paths grows exponentially. In contrast, fewer and less complex Boolean function operations are required to obtain the final function using explicit enumeration. For all practical cases in which Verity has been applied, the potential problem of an exponential number of true paths has not been encountered.

In the extraction model, MOS transistors are represented by switches which have two switched terminals and a control terminal corresponding to the MOS drain, source, and gate connection, respectively. Let  $N(G, S, W)$  denote a circuit with a set of logical gates  $G = \{G_1, \dots, G_g\}$ , a set of switches  $S = \{S_1, \dots, S_s\}$ , and a set of nets  $W = \{W_1, \dots, W_w\}$  interconnecting the logical gates and switches. Further, let  $I_{G_i} \subseteq W$  be the set of input nets and  $O_{G_i} \in W$  be the single output net of logical gate  $G_i$ . Similarly,  $C_{S_i} \in W$  and  $D_{S_i} \subseteq W$  denote the control net and the two switched nets of switch  $S_i$ , respectively. For consistency, we assume that each net connected to a primary circuit input or constant voltage source is driven by a specific logical gate with no inputs. Similarly, let each primary circuit output drive a logical gate without an output.

On the basis of this circuit definition, two specific groups of nets can be identified. First, the *controlling nets* include the inputs of logical gates and the controlling nets of all switches. Second, the outputs of logical gates form the set of *driven nets*. More formally, the set of controlling nets is defined as  $CW = \{W_i \mid (\exists G_j : W_i \in I_{G_j}) \vee (\exists S_j : W_i = C_{S_j})\}$ . Similarly, the set of driven nets is defined as  $DW = \{W_i \mid \exists G_j : W_i = O_{G_j}\}$ . The set of switches is partitioned into two groups: The set  $POS \subseteq S$  contains all switches which close when a 1 is applied at their control input. Similarly, the set of switches that close when a 0 is applied at the control input is included in  $NEG \subseteq S$ . Physically, these two sets correspond to NMOS and PMOS transistors, respectively.

An example of a mixed gate and switch-level circuit is given in Figure 5. As shown,  $CW = \{a, b, c, d, f, h\}$  includes all nets connected to transistor gates ( $a, b, f$ ) or to inputs of logical gates ( $b, c, d, h$ ). The outputs of logical gates form the set of driven nets  $DW = \{V_{DD}, GND, a, b, c, d, f, g\}$ .

The extraction of a Boolean function from a mixed circuit representation is based on the concept of paths. A path  $P_{p_1, p_n} = \{W_{p_0}, S_{p_1}, W_{p_1}, \dots, S_{p_n}, W_{p_n}\}$  is defined to be a subset of the switches  $S$  and nets  $W$  such that  $D_{S_{p_i}} = \{W_{p_{i-1}}, W_{p_i}\}, 1 \leq i \leq n$ . In other words, a path  $P_{p_0, p_n}$  from source net  $W_{p_0}$  to sink net  $W_{p_n}$  is defined to be a loop-free interconnected sequence of switches between these two nets. For example, nets  $e$  and  $h$  of the circuit in Figure 5 are connected by two paths:  $P_{e, h}^1 = \{e, S_3, h\}$  and  $P_{e, h}^2 = \{e, S_4, h\}$ . Note that this definition includes paths containing a single net without switches, e.g.,  $P_{a, a} = \{a\}$ . Such single-net paths define a self connection.

For the sake of functional extraction, two Boolean functions  $f_{W_i}^0$  and  $f_{W_i}^1$  are assigned to each controlling net  $W_i \in CW$ . These functions describe the conditions for which the net is driven by 0 and 1, respectively. The four possible value combinations ( $f^0, f^1$ ) =

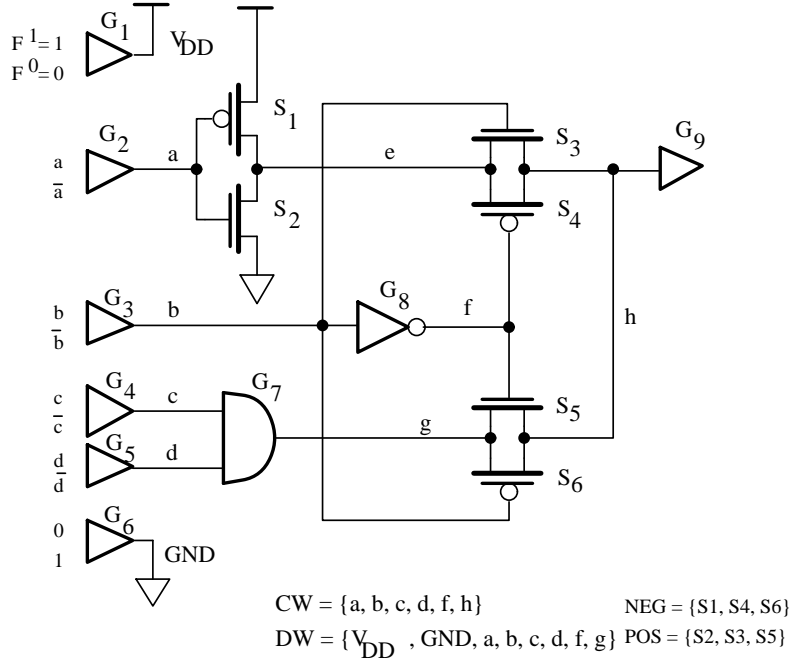


Figure 5: Example of a mixed gate and switch-level circuit.

$\{(0, 0), (0, 1), (1, 0), (1, 1)\}$  correspond to net status: *high impedance* (floating), *logical-1*, *logical-0*, and *collision*. Similarly, two driving functions  $F_{W_i}^0$  and  $F_{W_i}^1$  are assigned to each driven net  $W_i \in DW$ . They are computed by the driving gate on the basis of the functions of the gate inputs. The computation rules for a representative set of basic gate types is provided in Table 1. For example, function  $F^0$  and  $F^1$  of the  $V_{DD}$  driver  $G_1$  are set to 0 and 1, respectively. An independent variable and its complement are assigned to  $F^1$  and  $F^0$  of the primary inputs, respectively (e.g.,  $F_a^1 = a, F_a^0 = \bar{a}$ ). Further, internal logical gates compute the gate function for  $F^1$  and the dual gate function for  $F^0$ . As an example, the driving functions of net  $g$  in Figure 5 are  $F_g^0 = f_c^0 \vee f_d^0 = \bar{c} \vee \bar{d}$  and  $F_g^1 = f_c^1 \wedge f_d^1 = c \wedge d$ .

Gate type	$F^0$	$F^1$
Constant-0 (GND)	1	0
Constant-1 ( $V_{DD}$ )	0	1
Primary input	$\overline{variable}$	$variable$
AND	$\bigvee_{\forall \text{ inputs } W_i} f_{W_i}^0$	$\bigwedge_{\forall \text{ inputs } W_i} f_{W_i}^1$
OR	$\bigwedge_{\forall \text{ inputs } W_i} f_{W_i}^0$	$\bigvee_{\forall \text{ inputs } W_i} f_{W_i}^1$
NOT	$f_{W_i}^1$	$f_{W_i}^0$

Table 1: Computation of the driving functions for various gate types.

In the following, we assume that floating conditions at controlling nets do not occur in a correct circuit implementation. A generalization of the extraction scheme which includes dynamic circuit techniques is discussed in the subsection on time-sliced extraction for dynamic circuits. According to the assumption that floating controlling nets violate the chosen circuit technique, the logical values at any controlling net  $W_i$  are completely determined by the set of paths which connect  $W_i$  to driven nets. Moreover, the functions  $f_{W_i}^0$  and  $f_{W_i}^1$  can be calculated as follows:

$$f_{W_i}^0 = \bigvee_{\forall P_{i,j}} (f_{P_{i,j}} \wedge F_{W_j}^0), \quad (1)$$

$$f_{W_i}^1 = \bigvee_{\forall P_{i,j}} (f_{P_{i,j}} \wedge F_{W_j}^1), \quad (2)$$

where  $f_{P_{i,j}}$  encodes the conditions for which path  $P_{i,j}$  conducts.

Using the formulas for calculating the functions  $(f^0, f^1)$  and  $(F^0, F^1)$  of controlling nets and driven nets, respectively, a recursive extraction algorithm can be formulated. Starting from the output and test points, the procedures shown in Figure 6 calculate the functions for the entire input fan-in cone of these nets.

**Algorithm Compute\_Function (  $W_i$  )**

```

FOR all paths  $P_{i,j}$  from a driven net  $W_j$  to  $W_i$  DO
  Compute_Path_Function ( $P_{i,j}$ );
  Compute_Drive_Function ( $W_j$ );
END;
Calculate  $f_{W_i}^0$  and  $f_{W_i}^1$  according to Equations (1) and (2);
END;
```

**Algorithm Compute\_Path\_Function (  $P_{i,j}$  )**

```

FOR all switches  $S_k \in P_{i,j}$  DO
  Compute_Function ( $C_{S_k}$ );
END;
 $f_{P_{i,j}} = \bigwedge_{\forall S_k \in P_{i,j} \cap POS} f_{C_{S_k}}^1 \wedge \bigwedge_{\forall S_k \in P_{i,j} \cap NEG} f_{C_{S_k}}^0$ ;
END;
```

**Algorithm Compute\_Drive\_Function (  $W_i$  )**

```

FOR all inputs  $W_j$  of logical gate  $G_k$  feeding  $W_i$  DO
  Compute_Function ( $W_j$ );
END;
Combine input functions according to Table 1;
END;
```

Figure 6: Extraction algorithm for the Boolean function.

The calculation of the path functions  $f_{P_{i,j}}$  is based on the control nets of the switches forming this path. Depending on the switch type,  $f^0$  or  $f^1$  is taken for PMOS or NMOS

transistors, respectively. The path function of single-net paths without switches is defined as constant 1.

### 3.3.2 False-path Elimination

In general, switch-level structures can contain an exponential number of paths. Typical problematic structures are arithmetic shift or rotate operations implemented by flow trees of pass transistors. Since the extraction algorithm is based on an explicit enumeration of all paths, these circuits would result in an exponential run-time complexity. As an example, consider the four-bit barrel shifter [36] shown in Figure 7. The structure, formed by 16 pass-transistors, contains 1313 paths from the data outputs to the data inputs.

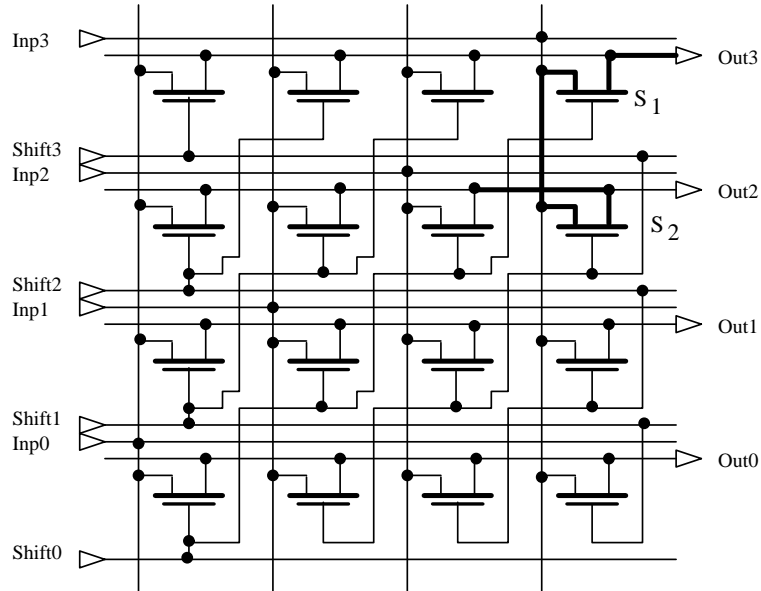


Figure 7: Barrel shifter circuit which contains an exponential number of paths.

In a proper implementation of such circuits, the majority of paths are false. That is, for all valid input patterns, most of the paths are non-conducting. According to Equations (1) and (2), false paths do not contribute to the logical function of controlling nets. Therefore, they can be eliminated from the enumeration process without affecting the extracted net functions. Consider the example in Figure 7. In normal operation, select inputs  $Shift0$ ,  $Shift1$ ,  $Shift2$ , and  $Shift3$  are mutually exclusive. Any path involving transistors controlled by different select inputs (e.g.,  $\{S_1, S_2\}$ ) is false.

For elimination of false paths, a pruning scheme is included in the extraction algorithm. Starting from the controlling net  $W_i$ , the set of driving paths is generated by recursively tracing connected switches to any driven net. The path function is built during the traversal process. If at any time during the traversal a given path becomes false, further recursion is terminated. In the given example, the path enumeration can backtrack after encountering  $S_1$  and  $S_2$  from output  $Out3$ , since in normal operation both transistors can never be active at the same time ( $Shift0 \wedge Shift3 = 0$ ).

### 3.3.3 Extraction of Combinatorial Loops

A digital circuit containing structural loops is characterized by some logical feedback from a gate output to its input. Such loops do not necessarily cause sequential behavior. It has been shown that certain combinatorial functions can be implemented efficiently by circuits containing loops [37]. However, for gate-level designs, combinatorial circuits with loops are usually not desired. For this description level, circular structures cause numerous problems for several design tasks, such as logic synthesis or automatic test pattern generation.

In contrast, because of their performance advantage, for customized transistor-level combinatorial circuits, circular structures are very popular. A customized implementation of a pulse-to-static converter circuit is shown in Figure 8. For the two valid input patterns  $a, b = [0, 1], [1, 0]$ , the output values of the cross-coupled NANDs are uniquely determined and do not depend on past circuit states. However, because of the circular network structure, the recursive circuit traversal described above is not applicable.

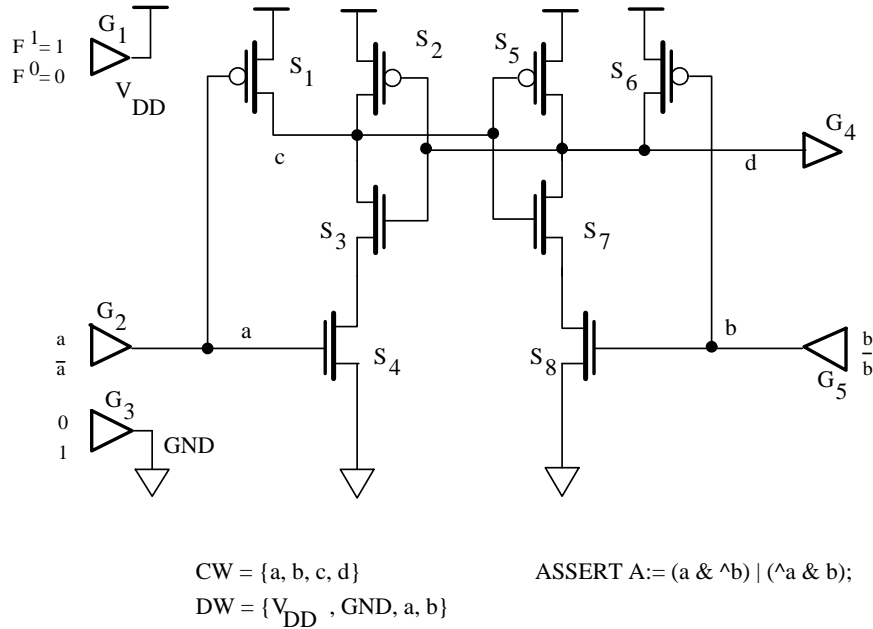


Figure 8: Example of a cyclic circuit structure.

The general configuration of circuits containing structural loops is given in Figure 9(a). The behavior of structural loops can be classified into two types: 1) False loops (also referred to as combinatorial loops) are not history-dependent, and they do not cause sequential behavior. 2) True loops (sequential loops) can store internal states and lead to sequential behavior.

Verity applies a loop examination technique related to that presented by Malik in [38]. The recursive network traversal identifies a structural loop by encountering a net which is marked as currently visited. The loop is then broken by inserting two new independent variables  $v^0$  and  $v^1$  [see Figure 9(b)]. After the backtracking reaches the broken net again, the extracted functions  $f_v^1$  and  $f_v^0$  are tested for the loop type according to the following

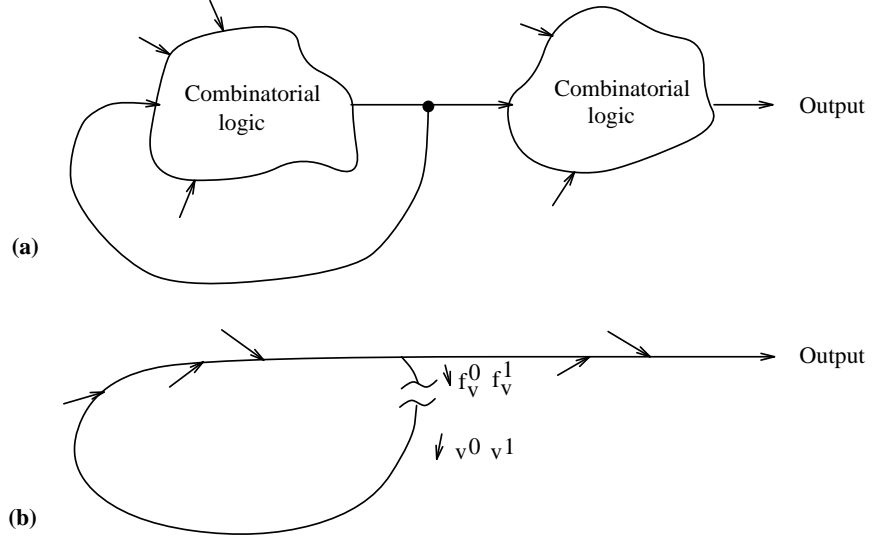


Figure 9: General structure of cyclic circuits: (a) circuit loop, (b) broken loop.

conditions:

$$\frac{\partial^2 f_v^0}{\partial v^0 \partial v^1} = 0 \quad \wedge \quad \frac{\partial^2 f_v^1}{\partial v^0 \partial v^1} = 0 \quad \Rightarrow \quad \text{false loop} ,$$

$$\frac{\partial^2 f_v^0}{\partial v^0 \partial v^1} \neq 0 \quad \vee \quad \frac{\partial^2 f_v^1}{\partial v^0 \partial v^1} \neq 0 \quad \Rightarrow \quad \text{true loop} ,$$

where  $\frac{\partial f}{\partial v} = f|_v \oplus f|\bar{v}$ .

As mentioned previously, Verity does not include sequential logic verification. True loops cause sequential circuit behavior and are flagged as violations of the extraction model. If a false loop is encountered, the loop must be revisited. For the extracted functions  $f_{W_i}$  of all loop nets  $W_i$ , the variables  $v^0$  and  $v^1$  are replaced by the corresponding functions  $f_v^0$  and  $f_v^1$ , respectively. This guarantees that the artificial variables  $v^0$  and  $v^1$  are properly eliminated for all other fan-outs from loop nets.

As an example, consider the extraction of the function of output  $d$  for the circuit shown in Figure 8. The path traversal starts to enumerate all paths driving net  $d$ , beginning with  $P_{V_{DD},d} = \{V_{DD}, S_5, d\}$ . This calls recursively the extractor for net  $c$ , which is controlling switch  $S_5$ . Assuming that path  $P_{V_{DD},b} = \{V_{DD}, S_2, c\}$  is traversed first, the extractor is called once more for net  $d$ , which was marked during the previous visit. Here, two variables  $v^0$  and  $v^1$  are created. After backtracking, the resulting net functions for  $d$  are  $f_d^0 = (v^0 \vee \bar{a}) \wedge b$  and  $f_d^1 = (v^1 \wedge a) \vee \bar{b}$ . After including the input constraint  $(a \wedge \bar{b}) \vee (b \wedge \bar{a})$  they simplify to  $f_d^0 = \bar{a}$  and  $f_d^1 = a$ . Since these functions do not depend on  $v^0$  and  $v^1$ , the structure is classified as a false loop.

### 3.3.4 Time-sliced Extraction for Dynamic Circuits

So far, high-impedance net conditions have been excluded from consideration in the previously described extraction techniques. Most standard static CMOS techniques do not

allow such undefined net conditions for proper circuit designs. However, dynamic CMOS techniques typically utilize precharging in order to achieve fast and compact circuit implementations. In dynamic circuits, controlling nets are not necessarily driven at all times during the clocking cycle.

As an example, consider the dynamic CMOS circuit in Figure 10. Net  $c$  is precharged to  $V_{DD}$  while  $reset$  is active. During the evaluation phase,  $reset$  is deactivated, and  $c$  might be discharged by a connection to  $GND$ , depending on the values of nets  $a$  and  $b$ . For the condition  $a \wedge b = 0$ , net  $c$  holds its precharged value 1; otherwise, it evaluates to 0.

Verity applies a time slicing approach to handle dynamic circuit techniques in a general way. The basic idea is to split the clock cycle into multiple slices and to extract an independent function for each slice. The slice functions for the nets are then combined to form the final net function.

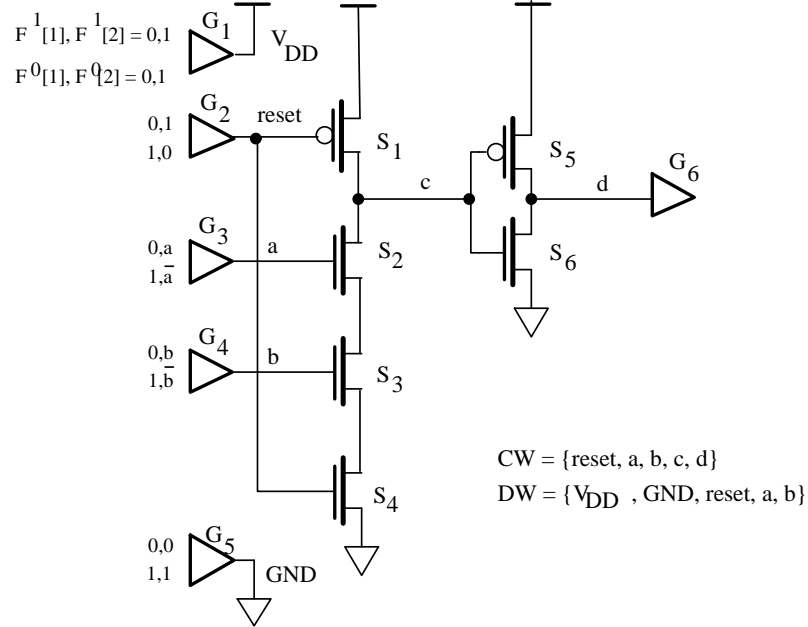


Figure 10: Example of a dynamic CMOS circuit.

Let us assume that the given circuit technique uses an  $n$ -phase clocking scheme. The driving functions  $(F_{W_i}^0[1], F_{W_i}^1[1]), \dots, (F_{W_i}^0[n], F_{W_i}^1[n])$  for the primary inputs and the clock inputs are specified according to a user-defined clocking scheme. For example, clock signal  $reset$  in the two-phase dynamic circuit of Figure 10 is activated during the precharge phase and deactivated during evaluation. The corresponding driving functions for precharge and evaluation are  $F_{reset}^0[1] = 1, F_{reset}^1[1] = 0$  and  $F_{reset}^0[2] = 0, F_{reset}^1[2] = 1$ , respectively. In this particular case it is assumed that the primary data inputs are inactive during precharge and driven by independent variables during evaluation.

In the general case, a set of  $n$  functions  $(f_{W_i}^0[1], f_{W_i}^1[1]), \dots, (f_{W_i}^0[n], f_{W_i}^1[n])$  is computed for each controlling net  $W_i$  by independently applying the extraction algorithm for each slice. The individual slice functions are combined to form the final combinatorial function

by the following scheme:

$$\begin{aligned} f^0 &= f^0[n] \vee (\overline{f^1[n]} \wedge f^0[n-1]) \vee \dots \vee \left( \bigwedge_{2 \leq i \leq n} \overline{f^1[i]} \wedge f^0[1] \right), \\ f^1 &= f^1[n] \vee (\overline{f^0[n]} \wedge f^1[n-1]) \vee \dots \vee \left( \bigwedge_{2 \leq i \leq n} \overline{f^0[i]} \wedge f^1[1] \right). \end{aligned}$$

This scheme effectively represents the electrical function of dynamic circuits, where for each clock phase the previous value is either kept or overwritten by an active path to a driven net. As an example, consider net  $c$  in Figure 10. The extracted functions are calculated as follows:

$$\begin{aligned} f_c^0[1] &= 0, \\ f_c^1[1] &= 1, \\ f_c^0[2] &= a \wedge b, \\ f_c^1[2] &= 0. \end{aligned}$$

The final functions are:

$$\begin{aligned} f_c^0 &= f_c^0[2] \vee \overline{f_c^1[2]} \wedge f_c^0[1], \\ f_c^1 &= f_c^1[2] \vee \overline{f_c^0[2]} \wedge f_c^1[1], \\ f_c^0 &= a \wedge b, \\ f_c^1 &= \bar{a} \vee \bar{b}. \end{aligned}$$

In Verity, the verification scheme for time-sliced extraction is customizable in a technology-dependent control file. This description includes the declaration of net types, their driving functions, the clocking scheme, the extraction method for each time slice, and the Boolean relations between the extracted slice functions. The control file also includes specific consistency checks to be performed for the net types at each time slice. For example, in the absence of strength-dependent logic, the expression  $f^0 \wedge f^1 = 0$  tests whether there exists a valid input pattern for which the net is pulled to 1 and 0 at the same time resulting in a collision. In the same manner,  $f^0 \vee f^1 = 1$  tests for floating conditions. The tests essentially validate the extraction model and alert the designer to specific undesirable circuit conditions.

### 3.4 Error Diagnosis

Functional verification proves the correctness of two different circuit representations including user-specified tests. In the case of a miscompare or failing test, the verification program must provide a detailed error report for finding and correcting the problem. For a productive application of formal verification in practical design projects, an effective debugging aid is as important as the actual comparison algorithm.

For each verification problem (i.e., miscomparing outputs, failing output tests, or failing consistency checks) Verity calculates a counter-example function representing all valid input patterns which exercise the undesired behavior. For example, for the given verification configuration of Figure 1, output value  $c = 1$  detects the cases in which FSMs  $A$  and  $B$  are functionally unequal. Since  $c$  is calculated on the basis of primary inputs  $\underline{x}$  and present state variables  $\underline{z}$ , each minterm of  $c$  represents a counter-example pattern for the inputs and state register that would exercise a functional miscompare between  $\underline{y}^A$  and  $\underline{y}^B$ .

Verity uses an arbitrary set of minterms from  $c$  as a basis for the counter-examples calculation. Let  $m$  be a minterm of the error function  $c$ . For each net  $W_i \in W$  of the erroneous implementation a counter-example value  $l_{W_i} \in \{0, 1, F, C\}$  is calculated as follows:

$$l_{W_i} = \begin{cases} 0 & \text{if } f_{W_i}^0(m) = 1 \wedge f_{W_i}^1(m) = 0 \\ 1 & \text{if } f_{W_i}^0(m) = 0 \wedge f_{W_i}^1(m) = 1 \\ F & \text{if } f_{W_i}^0(m) = 0 \wedge f_{W_i}^1(m) = 0 \\ C & \text{if } f_{W_i}^0(m) = 1 \wedge f_{W_i}^1(m) = 1 \end{cases},$$

where 0, 1,  $F$ , and  $C$  express logical zero, logical one, floating condition, and collision at the net, respectively.

In addition to the calculation of counter-example patterns, Verity applies an efficient error diagnosis algorithm which classifies the nets according to their probability of causing the error. Given a maximum number of assumed errors, this algorithm determines a circuit region which includes at least one erroneous net. Details of the diagnosis approach can be found in [39].

## 4 Practical Application and Results

Verity was developed in close collaboration with three microprocessor projects; it was intended to perform complete chip verification. The program is in daily use at five IBM sites and to date has verified more than 2000 logic macros, ranging in size from 100 to 25,000 MOS transistors. Although chip level verification has not been completed for any of these ongoing projects, large parts of the designs containing significant portions of the final chips have passed hierarchical verification.

Table 2 gives global usage statistics for the three design projects. The second and third columns provide the number of macros that have been verified and the total number of Verity runs for each project, respectively. As reported in the fourth column, about half of the macros passed functional verification on the first attempt. For the remaining macros, logical errors caused by an incorrect circuit implementation or an erroneous high-level specification were discovered. It is interesting to note that this ratio was consistently observed over the duration of these projects. The reported average number of Verity applications per macro does not necessarily reflect the total number of attempts to get the circuit functionally correct. Often, after small changes in the specification or implementation, verification for a particular macro is repeated to check that no errors were introduced.

Table ?? shows the performance of Verity for a set of randomly selected macros from a particular project. As a measure of the circuit complexity, the number of macro primary

Project	# Macros	# Verity runs	Percent of macros that passed first attempt	Average number of runs per macro
P1	933	6240	54.1 %	6.69
P2	704	4051	42.9 %	5.75
P3	262	1405	49.6 %	5.36

Table 2: Verification statistic for three ongoing microprocessors projects.

inputs and outputs, the number of MOS transistors and internal nets (excluding the black-boxed circuit parts), and the number of black-boxed submacros are given. The reported CPU times and memory requirements are taken from a RISC System/6000<sup>2</sup> Model 580 processor. For many circuits Verity can be used interactively producing a verification report within minutes. For larger macros, Verity runs are typically submitted to a pool of powerful workstations with more computing resources.

Macro	# Inputs / # Outputs	# Transistors / # Nets (excluding black boxes)	# Black boxes	CPU time (seconds)	Memory (MBytes)
M1	1487 / 499	7911 / 4068	83	435.2	37.3
M2	70 / 65	956 / 1129	128	40.3	4.6
M3	849 / 875	22944 / 8833	996	2526.5	644.8
M4	224 / 302	7876 / 4702	232	266.3	16.6
M5	66 / 65	802 / 341	0	3.6	4.8
M6	262 / 265	1585 / 1373	64	40.9	5.1
M7	194 / 64	800 / 474	0	3.9	3.7
M8	172 / 20	1212 / 823	48	202.4	10.4
M9	794 / 943	8775 / 4943	9	64376.7	154.6

Table 3: Verity performance for a set of randomly selected design examples.

Designer feedback from the three projects indicates that the effort to use Verity effectively for complete chip verification depends highly on the overall methodology. Many designers who incorporated verification early in the development process have found Verity invaluable. Often, because of the short turnaround time, the actual circuit designs are done in a trial-and-error fashion, switching between verification and correction. For such design styles, Verity provides incremental verification in which the user can verify an incomplete schematic. Missing circuit implementations are replaced by their corresponding high-level specification. This capability has proved to be a powerful design verification framework that is particularly suited to manual circuit entry.

On the other hand, the later the verification tool is introduced in the macro design process, the more effort is necessary to make the circuit pass. Adjustments to the macro interface, the latch, and the register structure, and repartitioning of either design representation could be required. When introduced at a late stage, these changes are expensive and typically have an impact on the schedule.

---

<sup>2</sup>RISC System/6000 is a trademark of International Business Machines, Incorporated.

## 5 Conclusions

In an effort to optimize the performance of digital systems, designers of high-performance circuits are moving from correct-by-construction synthesized methodologies to hand-crafted custom design. This fundamental shift has necessitated more complete methods for verifying correct system behavior. Verity addresses the problem of formally proving the correctness of a system implementation with respect to the specification.

Verity applies symbolic comparison techniques which implicitly prove the functional equivalence between a CMOS circuit implementation and an RTL specification for all possible input patterns. Since the underlying extraction algorithms are based on a switch-level model of the MOS circuit, Verity effectively removes the need for expensive circuit and switch-level simulation for the purpose of determining correct Boolean behavior. Moreover, formal techniques provide an exhaustive comparison, which makes the generation of simulation patterns unnecessary.

The algorithms applied in Verity permit the verification of entire microprocessor systems. The success of formal methods on such a large scale requires a strict design-for-verification methodology. This significantly affects the overall design partitioning, the hierarchical circuit structure, the chosen register methodology, and the assertions of logical boundary conditions. Successful use of Verity requires an early consideration of such issues in the design cycle.

Future research on Verity will address two areas, extending the applicable macro size and incorporating sequential verification capabilities. An exploration of alternative representations of Boolean functions could result in a significant increase in the verifiable circuit size. This includes extensions to existing BDD algorithms, test-pattern approaches, probabilistic methods, and various combinations. The incorporation of sequential verification algorithms into Verity will permit checking of the actual registers during hierarchical verification. Further, in conjunction with algorithms to determine automatically the register/latch correspondencies between the two design representations, such techniques will help to relax the strict requirements on a design-for-verification methodology.

## 6 Acknowledgements

The authors would like to thank Florian Krohm at the IBM Thomas J. Watson Research Center, Geert Janssen and Arjen Mets of the Technical University Eindhoven, and David Cheng of the University of California at Santa Barbara for their significant contributions in the development of Verity. They also wish to thank Victor Rodriguez of IBM Austin, David Appenzeller of IBM Burlington, and Terry Chappell, Barbara Chappell, and Kenneth Shepard at the IBM Thomas J. Watson Research Center for their invaluable input.

## References

- [1] D. K. Beece, R. Damiano, G. Papp, and R. Schoen, "The EVE companion simulator," in *Proceedings of The European Conference on Design Automation*, (Glasgow,

- Scotland), pp. 290–295, IEEE, March 1990.
- [2] M. M. Denneau, “The Yorktown simulation engine,” in *ACM IEEE Nineteenth Design Automation Conference Proceedings*, (Las Vegas, Nevada), pp. 55–59, ACM/IEEE, June 1982.
  - [3] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, “Symbolic model checking for sequential circuit verification,” *IEEE Transactions on Computer-Aided Design*, vol. 13, pp. 401–424, April 1994.
  - [4] G. Ditlow, W. Donath, and A. Ruehli, “Logic equations for MOSFET circuits,” in *Proceedings of the IEEE International Symposium on Circuits and Systems*, (Newport Beach, CA), pp. 752–755, IEEE, May 1983.
  - [5] Z. Barzilai, L. M. Huisman, G. M. Silberman, D. T. Tang, and L. S. Woo, “Simulating pass transistor circuits using logic simulation machines,” in *Proceedings of the 20th Design Automation Conference*, pp. 157–163, ACM/IEEE, June 1983.
  - [6] R. E. Bryant, “Boolean analysis of MOS circuits,” *IEEE Transactions on Computer-Aided Design*, vol. 6, pp. 634–649, July 1987.
  - [7] D. T. Blaauw, D. G. Saab, P. Banerjee, and J. A. Abraham, “Functional abstraction of logic gates for switch-level simulation,” in *Proceedings of The European Conference on Design Automation*, (Amsterdam, The Netherlands), pp. 329–333, IEEE, February 1991.
  - [8] R. E. Bryant, “Extraction of gate level models from transistor circuits by four-valued symbolic analysis,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 350–353, IEEE, November 1991.
  - [9] W. J. Dally and R. E. Bryant, “A hardware architecture for switch-level simulation,” *IEEE Transactions on Computer-Aided Design*, vol. 4, pp. 239–249, July 1985.
  - [10] E. H. Frank, “Switch-level simulation of VLSI using special-purpose, data-driven computer,” in *Proceedings of the 22rd Design Automation Conference*, pp. 735–738, ACM/IEEE, June 1985.
  - [11] S. B. Akers, “Binary decision diagrams,” *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.
  - [12] J.-C. Madre and J.-P. Billon, “Proving circuit correctness using formal comparison between expected and extracted behaviour,” in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 205–210, ACM/IEEE, June 1988.
  - [13] J. Jain, J. Bitner, D. S. Fussel, and J. A. Abraham, “Probabilistic design verification,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 468–471, IEEE, November 1991.
  - [14] J. P. Roth, “Hardware verification,” *IEEE Transactions on Computers*, vol. C-26, pp. 1292–1294, December 1977.

- [15] D. Brand, "Verification of large synthesized designs," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 534–537, IEEE, November 1993.
- [16] W. Kunz, "HANNIBAL: an efficient tool for logic verification based on recursive learning," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 538–543, IEEE, November 1993.
- [17] G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean comparison of hardware and flowcharts," *IBM Journal of Research and Development*, vol. 26, pp. 106–116, January 1982.
- [18] M. Monachino, "Design verification system for large-scale LSI designs," *IBM Journal of Research and Development*, vol. 26, pp. 89–99, January 1982.
- [19] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.
- [20] M. Fujita, H. Fujisawa, and Y. Matsunaga, "Variable ordering algorithms for ordered binary decision diagrams and their evaluation," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 6–12, January 1993.
- [21] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 42–47, IEEE, November 1993.
- [22] A. Shen, S. Devadas, and A. Ghosh, "Probabilistic construction and manipulation of free Boolean diagrams," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 544–549, IEEE, November 1993.
- [23] M. Blum, A. K. Chandra, and M. N. Wegman, "Equivalence of free boolean graphs can be decided probabilistically in polynomial time," *Information Processing Letters*, vol. 10, pp. 80–82, March 1980.
- [24] M. C. McFarland, "Formal verification of sequential hardware: A tutorial," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 633–653, May 1993.
- [25] A. Gupta, "Formal hardware verification methods: A survey," *Formal Methods in System Design*, no. 1, pp. 5–92, 1992.
- [26] F. K. Hanna and N. Daeche, "Specification and verification using higher-order logic," in *Proceedings of 7th CHDL*, 1985.
- [27] A. Camilleri, M. Gordon, and T. Melham, "Hardware verification using high-order logic," in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, 1987.
- [28] S. Devadas, H.-K. T. Ma, and A. R. Newton, "On the verification of sequential machines at different levels of abstraction," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 271–276, IEEE, 1987.

- [29] O. Coudert, C. Berthet, and J. C. Madre, “Verification of sequential machines using Boolean functional vectors,” in *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 111–128, November 1989.
- [30] K. L. McMillan, *Symbolic Model Checking*. Boston, MA: Kluwer Academic Publishers, 1993.
- [31] C. Ebeling, “GeminiII: A second generation layout validation program,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, California), pp. 322–325, IEEE, November 1988.
- [32] R. E. Bryant, “A switch-level model and simulator for MOS digital systems,” *IEEE Transactions on Computers*, vol. 33, pp. 160–177, February 1984.
- [33] R. E. Bryant, D. Beatty, and K. Brace, “COSMOS: a compiled simulator for MOS circuits,” in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 9–16, IEEE, 1987.
- [34] A. Jain and R. E. Bryant, “Mapping switch-level simulation onto gate-level hardware accelerators,” in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 219–222, June 1991.
- [35] T. Kam and P. A. Subrahmanyam, “Comparing layouts with HDL models: A formal verification technique,” in *Proceedings of the IEEE International Conference on Computer Design*, (Boston, MA), pp. 588–591, IEEE, October 1992.
- [36] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley Publishing Company, 1980.
- [37] W. H. Kauz, “The necessity of closed loops in minimal combinational circuits,” *IEEE Transactions on Computers*, pp. 162–164, February 1970.
- [38] S. Malik, “Analysis of cyclic combinatorial circuits,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 618–625, IEEE, November 1993.
- [39] A. Kuehlmann, D. I. Cheng, A. Srinivasan, and D. P. LaPotin, “Error diagnosis for transistor-level verification,” in *Proceedings of the 31th ACM/IEEE Design Automation Conference*, (San Diego, CA), pp. 218–224, IEEE, June 1994, [PostScript] , [HTML]