VLSI Logic Test, Validation and Verification Lecture 1, Aug 25, 2004

Instructor: Priyank Kalla
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT 84112
Email: kalla@ece.utah.edu

I. Introduction to VLSI Testing, Validation & Verification

VLSI Testing, Design Validation and Computer-Aided Verification are three separate courses in their own right. In many universities, they are indeed treated separately and there are dedicated courses that cover these topics individually. In spite of teaching these courses separately, many instructors feel that there was so much material to cover and so little time that they had to exclude some topics from the syllabus. So you must be wondering why is yours truely trying to do the impossible by teaching everything all at once? Well, I am really not trying to do the impossible. My intention is not to teach you everything related to VLSI Testing, Validation and Verification - I cannot, because that is not possible. I only wish to introduce to you the important fundamental problems in these areas and *highlight the similarities among them.*

Testing, validation and verification problems are often addressed by separate communities - VLSI testing is often considered to belong to the Electrical Engineering domain, while Validation and Verification are often thought of as Computer Science subjects. In industry, the methodologies employed by testing, validation and verification engineers are also different. While the application domain of these problems is certainly different, there is one similarity among them - and that is the underlying mathematical framework. Many test, validation and verification problems, when considered within the VLSI-CAD paradigm, are essentially different applications/implementations of fundamental switching theory concepts. This is one important issue that I wish to highlight through this course. But why do I want to do that?

As the size and complexity of digital designs increases, the difficulty to test, validate and verify the designs also increases in scope. Some of the problems in these areas are very well understood and efficient CAD techniques have been developed to solve the respective problems. Increasingly, many solutions to different problems have begun to borrow concepts from one another, and these fields have begun to overlap. For example, testing a fabricated chip for manufacturing defects is one of the more successfully tackled problems in VLSI CAD. Automatic Test Pattern Generation (ATPG) algorithms with highly accomplished search techniques are widely available. Particularly for sequential circuits, where the state space of designs is prohibitively large, a lot of research has been devoted to analyze the state space to excite the faulty behaviour in the defective circuit. As you will learn in the course, fundamentally, the problem of verifying *functional equivalence* of two different implementations of a sequential circuit also requires state space analysis. Why not borrow the concepts from sequential ATPG techniques to solve sequential equivalence verification? Why not, of course, but a more important question is, “What to borrow from VLSI testing to make design verification more efficient?” You cannot understand this issue unless you are well aware of the ATPG problem as well as the equivalence verification problem. Recently (in the last 5-6 years) there has been a lot of work in both test and verification where solutions have been borrowed from one application domain to the other - and this trend is very welcome if it enables development of robust, automated CAD frameworks. I hope that through this course, not only would you understand individual test and verification issues but (hopefully) also contribute to any and all of these fields.

So what is VLSI testing, what is design validation and what is verification? At what point in the VLSI circuit realization process do we employ these techniques? How are these problems different? And how are these problems similar? At the moment, I will only attempt to answer the first three questions. You will get the answer to the last question as this course progresses. Let me first describe the VLSI realization process and then explain where the Validation, Verification and Testing problems are employed.

II. The VLSI Realization Process

The VLSI realization process begins with a “customer” detailing the product specifications to the designers. These product specifications are written, usually, in English: they may describe only the desired input-output behaviour, or in some cases they may describe the entire instruction-set of a computer. Since the specification is in English, it is impossible to operate upon it. It is the designers job to understand the specifications and write them in a Hardware Description Language (HDL), so as to be able to synthesize and realize a functioning circuit from it. In transforming the “English version” of the specs into, say, a VHDL version, the designer may have inadvertently introduced a bug in it. This process of transforming from English to VHDL is rather tedious for very large designs - bugs may creep in because of some misunderstandings, omission of certain behaviours, improper modeling of don’t care conditions, carelessness, etc. It is difficult to have a flawless spec on the first go; designers keep on writing (and augmenting) test-benches to *validate the specs via simulation.*

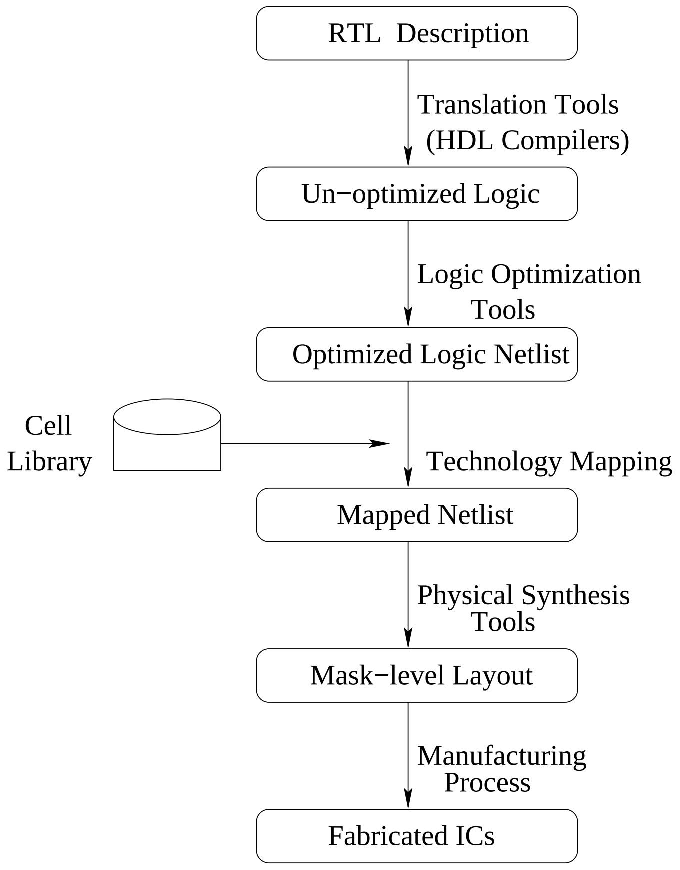


Fig. 1. The VLSI Realization Process: A Typical design synthesis flow.

A. *Validation via Simulation*

So here is the first process of "testing" (oops, not testing, but actually validating) the specification. Validation of the initial HDL description is a major bottleneck in the design process. Because the HDL description is usually the first description of the design, *simulation* is the primary methodology for validating it. Simulation based validation, however, is necessarily incomplete because it is not computationally feasible to exhaustively simulate the entire design. It is therefore important to measure the degree of verification coverage, both qualitatively and quantitatively. Coverage measures such as code (line) coverage, branch coverage, transition coverage, etc., have been used extensively in industry. A "high" degree of simulation coverage provides confidence to the designer regarding the correctness of the design. A "low" degree of coverage implies that certain functionalities of the design may not have been simulated. In which case, should there be an error in the non-simulated portion of the design, it may not have been observed during the validation process. It is therefore important to simulate the design functionality as much as possible. This requires automatic generation of a large number of simulation vectors to excite all the components in the design.

Pseudo-random techniques for automatically generating simulation vectors are widely in use. Simulation tools may use random number generators, and randomly generate **1**s and **0**s, collect a vector of inputs and apply them to the design and perform the simulation. (Ever seen C language random number generators? Try reading the man-pages for random(), randu(), drand48(), etc. These are the most primitive random number generators....) While they do produce a virtually endless stream of simulation vectors, enhanced coverage is achieved only infrequently. Often, a situation arises where certain areas of code (or parts of the design) are *not* excited by the generated set of vectors. Validation tools can monitor every statement of code and keep a record of the number of times it was executed in response to the input stimulus. They can thus identify parts of the HDL design that have not been excited at all by the simulation vector set. After identifying the parts of the RTL description that were not excited by the simulation vectors, validation engineers have to generate the vectors that would excite these portions of the design.

It is desirable to have validation tools that can automatically compute a set of vectors which would cover the un-exercised parts of the design, thus enhancing the over-all coverage measures. Consider the circuit shown in Fig. 2. In order to simulate this design, we randomly generate a set of test vectors. Suppose that our test vector set, being in-exhaustive, failed to excite the following output values: $\{u = 1, v = 1, w = 1\}$. The simulation test bench can detect that the combination of values $\{u = 1, v = 1, w = 1\}$ was not excited, and ask the validation tool to automatically generate those test vectors that would excite the required output values.

The above problem can be formulated as follows: Given a set of *output value requirements*, say $\{u = 1, v = 1, w = 1\}$,

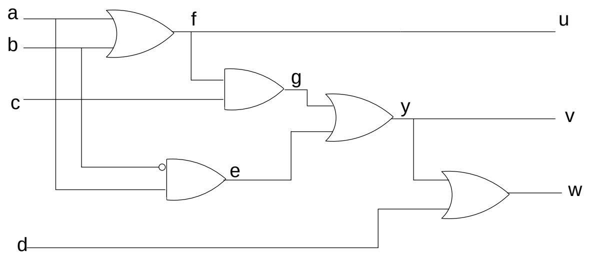


Fig. 2. An example circuit.

how do we find an *input assignment* that satisfies these requirements? This problem can be solved using Boolean satisfiability (SAT). Constructing the product (conjunction) of the output requirements in terms of the primary inputs results in all satisfying solutions. Let u, v, w represent the primary outputs of the circuit, the Boolean equations for which are as follows:

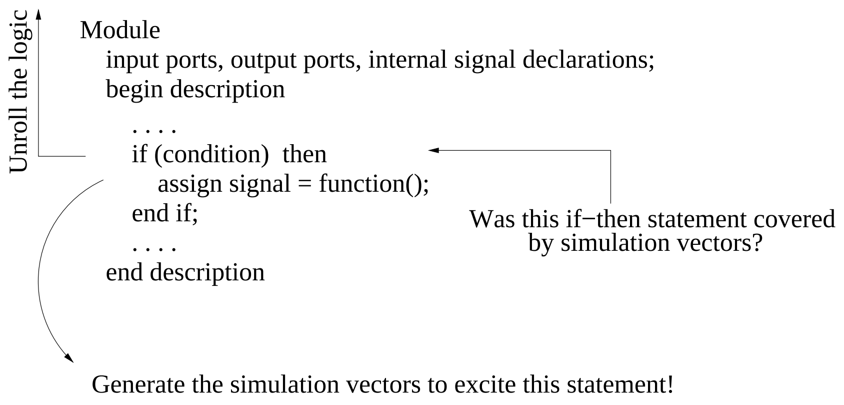
$$u = a + b \tag{1}$$
$$v = (a+b)\cdot c + a\overline{b} \tag{2}$$
$$w = ((a+b)\cdot c + a\overline{b}) + d \tag{3}$$

Let f represent the product of the outputs.

$$\begin{aligned} f &= u \cdot v \cdot w && (4)\\ &= ((a+b)\cdot((a+b)\cdot c + a\overline{b}))\cdot((a+b)\cdot c + a\overline{b} + d) && (5)\\ &= ac + bc + a\overline{b} && (6) \end{aligned}$$

It follows that any of the cubes ac, bc, or $a\overline{b}$ provide the satisfying assignments. A SAT engine can be used to generate the required test vectors $\langle 1-1-, -11-, 10--\rangle$ which can be used for enhanced simulation coverage.

The vector generation problem is often not as rudimentary as the one described above. Consider the pseudo-RTL description shown in Fig. 3. While simulating the design, suppose, the **if-then** statement was not excited by the set of simulation vectors. The validation tool needs to generate test vectors that would excite this statement. This manifestation of the vector generation problem can be solved as follows. First, the logic corresponding to the **if-then** statement needs to be extracted and represented in terms of the primary inputs. Satisfiability checking can then be applied on the resulting Boolean function. SAT solutions to this problem would provide the simulation vectors to excite the statement in question.



```
Module
   input ports, output ports, internal signal declarations;
   begin description
      . . . .
      if (condition)  then
         assign signal = function();
      end if;
      . . . .
   end description
```

Unroll the logic
Was this if−then statement covered
by simulation vectors?
Generate the simulation vectors to excite this statement!

Fig. 3. RTL code coverage.

For real-world HDL-designs, the extracted logic for which we need to find a satisfying solution can become prohibitively large. For this reason, we need to develop robust, automated CAD-tools that can solve these problems for Very Large Scale Integrated Circuits (VLSI). Now do you see the connection - VLSI & Design Validation? Or are you confused? What are cubes? What is Boolean Satisfiability (SAT)? Don't worry - just fasten your seatbelts - this is what we will cover in the course.

III. Formal Design Validation/Verification

Design descriptions have traditionally been validated by extensive simulation. However, simulation based validation offers no guarantees for correctness. Why? Because even if you have simulated the entire code - all **if-then, case, for, while**, etc., statements - you have NOT simulated (and CANNOT simulate) every permutation and combination of those if-then-else and other statements. For this reason, verification (or should I say validation?) engineers increasingly rely upon mathematical techniques to *formally prove the correctness of designs.*

Any digital system can be described mathematically, perhaps by a set of mathematical equations - in our case, by Boolean equations. A set of rules are used to generate formulas describing the *system*, and semantic functions are used that assign some meaning to the formulas. But what are you verifying (or validating)? You wish to verify certain properties of the system. For example, suppose that you are asked to design a traffic light controller for a 4-way intersection. You want to ensure, over the entire design space of the controller, that the traffic lights controlling two perpendicularly intersecting streets (*e.g.* Wakara Way and Foothill) should **never be green at the same time**. Otherwise there will be an accident! If g_1 and g_2 are the green signals for the intersecting streets, your design should satisfy the property $\overline{g_1 \cdot g_2}$ throughout the design space. In other words, we need a set of rules to generate formulas defining the *properties* that are to be checked, and semantic functions that define satisfiability of that property by the system. Now that we have a mathematical description of the system, as well as a mathematical representation of the properties that we wish to verify, all we need to do is formally verify whether or not our system satisfies the property. Since we are in the world of VLSI-CAD, we need an *algorithm* to verify the satisfaction relation. This is what is so dramatically called "formal verification".

Traditional approaches to formal verification attempt to show that there exists a formal proof of the formulae defining the correctness criteria. The proof is obtained from the formulas characterizing the design and the axioms underlying the associated logical system [1]. This process is referred to as *theorem proving.* The benefit of this approach is its generality and completeness. However, generating the proof automatically is cumbersome in both theory and practice. Existing heuristics to automatically generate the proof are both memory and compute intensive. As a result, theorem proving lacks the level of automation that is desirable for a CAD framework to be practically useful.

Model checking [2] provides a different approach to the formal verification problem. The system is characterized by a finite *state transition graph* (STG) where the vertices represent the configurations (or the *states*) that the system can reside in, and the edges represent the transition between the states. Properties of the system that are to be verified for satisfaction are represented using *temporal logic formulae*. Temporal logics are essentially equivalent to various special fragments of *linear time* logics or of *branching time* logics. Algorithmically, the verification is performed by traversing the state transition graph; starting from the initial state, the set of reachable states is found where all the states in the set satisfy the desired property. Model checking tools [3] [4] have achieved a significant level of automation and maturity and are widely in use in both academia and industry. One of the factors behind the success of model checking is that the STG of the underlying system can be relatively easily extracted from the designs described in either conventional hardware description languages or from circuit level netlists. Hence, the designers find it straightforward to include property verification within their design/synthesis methodology, as a common HDL-framework can be used for simulation, synthesis and verification.

Now that I have confused you even more, let me give you the good news. Theorem Proving and Model Checking are NOT the subjects that are going to be covered in this course. The only formal verification technique that we will study is the one described below.

IV. Implementation Verification

Once an HDL model is validated (by extensive simulation or property verification), it is transformed into a gate-level representation so that logic synthesis tools can be used to optimize the design according to the desired area/delay constraints. As shown in Fig. 1, the design proceeds through a varied set of optimization and transformation operations. During various transformation stages, different implementations of the design, or parts of the design, are examined depending upon the constraints, such as area, performance, testability, etc. As the design is modified by replacing one of its components by another equivalent implementation, it needs to be verified whether or not the modified design is *functionally equivalent* to the original one.

For example, after the HDL representation is transformed into a gate-level netlist, it is important to verify that the functionality of the gate-level netlist is equivalent to that of its HDL representation. Similarly, after logic optimization is performed on the gate-level netlist, the designers have to ensure that the optimization process indeed did not introduce a bug in the design. In order to verify equivalence of different *implementations* of the same design, equivalence checking tools are used in various stages of the design cycle. Fig. 4 depicts a typical implementation verification flow during various design phases.

The question that you should now be asking is, how do we verify equivalence of two representations of the same design? Can we not generate simulation vectors for one representation and use the same set to simulate both design representations and observe whether or not their behaviour is identical? Can we simulate the *entire* design space of both designs practically? It is infeasible to simulate the entire design space. In order to prove equivalence efficiently, we describe the designs in some mathematical form, and then prove that their respective mathematical forms are equivalent. How do we do that? Welcome to the class....

But before we proceed any further, did you understand the difference between validation and verification?

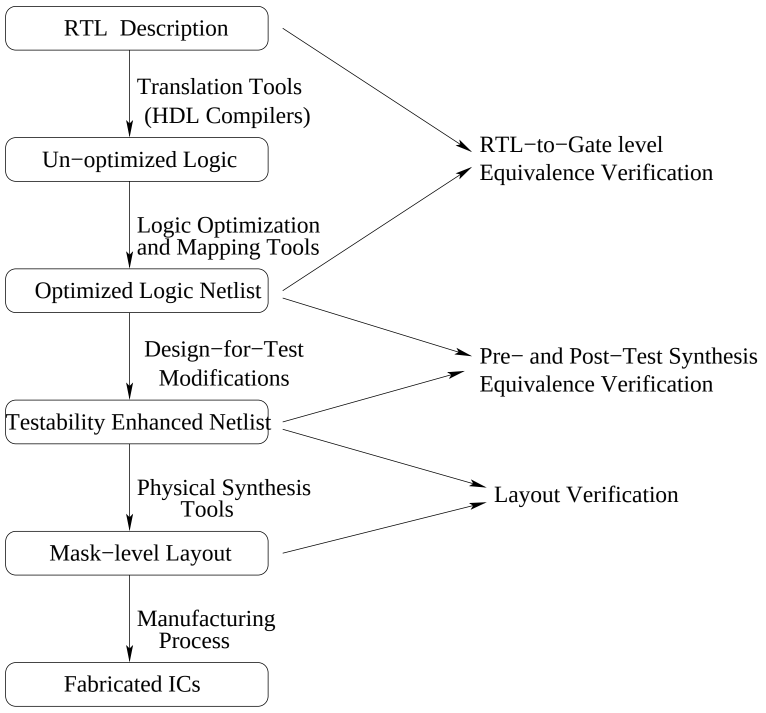

RTL Description
Translation Tools (HDL Compilers)
Un−optimized Logic
Logic Optimization and Mapping Tools
Optimized Logic Netlist
Design−for−Test Modifications
Testability Enhanced Netlist
Physical Synthesis Tools
Mask−level Layout
Manufacturing Process
Fabricated ICs
RTL−to−Gate level Equivalence Verification
Pre− and Post−Test Synthesis Equivalence Verification
Layout Verification

Fig. 4. Typical implementation verifi cation flow.

In VLSI-CAD nomenclature, Validation is generally employed on specifications, such as *validation of a specification*. On the other hand, verification generally relates to implementations, such as *verification of the implementation with respect to the specification*. Some folks use Validation to mean "informal verification" which is in contrast to "formal verification". This differentiation isn't totally incorrect either. Combine the two definitions, and you can interpret the meaning of *formally validating the specification using property/model checking*. Here, you are validating the spec - not verifying it w.r.t. any implementations. You are *validating* a particular property of the spec and you are doing that by *mathematically proving satisfaction* of the property by the system; hence the formality. Now, can there be such a thing as "design equivalence validation"? A point to ponder!

V. Testing for Manufacturing Defects

Lexicographically speaking, Testing, Validation and Verification really mean the same thing. Now I did highlight the difference between validation and verification (w.r.t. VLSI-CAD) in the last paragraph - even though the distinction between them is not sacrosanct. However, the meaning of VLSI Testing, it being a well researched subject since 1960s, is crystal-clear. It corresponds to *testing for manufacturing defects in a fabricated chip.*

Once the high-level (VHDL?) model is validated, and the description has been synthesized according to the design constraints (delay, area, power, etc.), the circuit goes for fabrication. Semiconductor processing is a technologically complicated process. With feature sizes going down rapidly (usually, every year), the new process may not be as mature and reliable as the old ones. This means that we may not be able to make all the chips fault-free. Before sending the chips out to the customer, you need to test whether or not there are any manufacturing defects in them. You validated the design descriptions, and also verified the implementations, so hopefully the chips do not have logical errors. Any error in the chip would now mean that the fabrication process wasn't flawless. Perhaps two parallel running aluminum wires got shorted; perhaps an aluminum wire connection between two gates was open because a dust particle fell on the spot when metalization was going through; perhaps the SiO_2 insulation didn't go through well and latch-up occurred; perhaps transistors were not fabricated properly and......

Before sending the chip to the customer, we need to ensure that the chip is not defective. Don't you hate it when you order a computer and find that the motherboard is faulty? Does that mean that the manufacturer didn't test it? Or does it mean that testing was performed, but their test strategy could not develop a good test vector suite that would have excited each and every fault? Unfortunately, there are so many failures to test for: wires, transistors, switching delay, excess current or no current, shorts, opens, unknown logic values (neither a 0 not a 1), etc. Each and all of them can fail individually or collectively. How can you possibly test everything throughout the entire chip? Well, that is where engineering comes into play.

In general, direct mathematical treatment of physical failures and fabrication defects is not feasible. Thus, test engineers model these faults by *logical faults*, which are a convenient representation of the effect of the physical faults on the operation of a system. Such a model assumes that the components of a circuit are fault-free and only their interconnections are defective. These logical faults can represent many different physical faults, such as, opens, shorts with power or ground, and internal faults in the components driving signals that keep them stuck-at a logic value. A *short* results from unintended interconnection of points, while an *open* results from a break in a connection. A short between ground (or power) and a signal line can result in a signal being *stuck* at a fixed value. A signal line when shorted with ground (power) results in its being *stuck-at-0* (*stuck-at-1*) and the corresponding fault is called a *s-a-0* (*s-a-1*) fault.

Figure 5 illustrates the effect of an *s-a-0* fault at line A on the operation of a circuit. An input signal to an AND gate when shorted to ground (*s-a-0*) results in its output always being *s-a-0*. Thus, if line A is *s-a-0*, irrespective of the values at all other inputs of the circuit, output Z will always evaluate incorrectly to logic value 0. In such a way, the presence of a stuck-at fault may transform the original circuit to one with a different functionality.

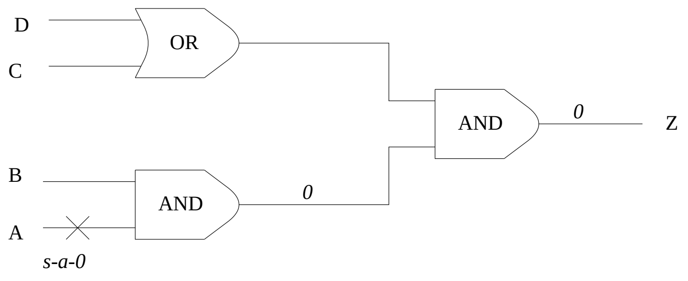


Fig. 5. Faulty operation of a circuit due to a $stuck - at - 0$ fault at line A.

Why did line A get stuck at logical value 0? Was there a short between line A and ground? Or was the pull-down section of the CMOS gate faulty such that there was a permanent connection to ground from the A input (Gate-to-source shorted)? This issue is beyond our scope and falls into the domain of *fault diagnosis*. We only care whether or not the circuit operation is correct.

By modeling such physical defects as logical faults, we can apply switching algebra concepts and ascertain whether or not the circuit is correct. Such an abstract view allows us to identify a fault relatively easily and ignore its electrical causes - its cause could be anything. This makes our job easier and more efficient.

All said and done, how do we generate tests for the entire circuit? What type of fault models do we employ? How do we interpret the tests? As I said, welcome to this class! And Good Luck to all of you!

References

[1] Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems", in Springer-Verlag, editor, *LCNS*, 1992.
[2] E. A. Emerson, "Temporal and Modal Logic", in J. van Leeuwen, editor, *Formal Models and Semantics*, vol. B of *Handbook of Theoretical Computer Science*, pp. 996–1072. Elsevier Science, 1990.
[3] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vencentelli, F. Somenzi, A. Aziz, S-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, G. Shiple, S. Swamy, and T. Villa, "VIS: A System for Verifi cation and Synthesis", *in Computer aided Verifi cation*, 1996.
[4] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.