A COMMON APPROACH TO TEST GENERATION AND HARDWARE VERIFICATION BASED ON TEMPORAL LOGIC

Thomas Kropf, Hans-Joachim Wunderlich
University of Karlsruhe, Institute of Computer Design and Fault Tolerance
P.O. Box 6980, 7500 Karlsruhe, Germany

Hardware verification and sequential test generation are aspects of the same problem, namely to prove the equal behavior determined by two circuit descriptions. During test generation, this attempt succeeds for the faulty and fault free circuit if redundancy exists, and during verification it succeeds, if the implementation is correct with regard to its specification. This observation can be used to cross-fertilize both areas, which have been treated separately up to now. In this paper, a common formal framework for hardware verification and sequential test pattern generation is presented, which is based on modeling the circuit behavior with temporal logic. In addition, a new approach to cope with non resetable flipflops in sequential test generation is proposed, which is not restricted to stuck-at faults. Based on this verification view, it is possible to provide the designer with one tool for checking circuit correctness and generating test patterns. Its first implementation and application is also described.

1. INTRODUCTION

The increased use of VLSI especially in safety critical systems demands a high confidence in the correct functioning of these systems. Thus, it has to be ensured that circuits contain *neither* design errors *nor* fabrication faults.

Hardware verification copes with errors, which occur in the circuit during the design process. Verification is performed by formally proving that an implementation meets the specification, i.e. the behavioral requirements. Usually this is done by modeling the functional behavior in terms of logic and using a theorem proving tool to support the proof process (e.g. [1], [2], [3]).

Test generation on the other hand addresses the problem of finding input stimuli for a circuit in such a way that fabrication defects are found. Generally, this is accomplished by injecting faults given by an appropriate fault model into the description of the correct circuit and comparing the behavior of the resultant faulty circuit and the correct circuit to find input sequences such that the output values of the two circuits eventually differ. Most of the algorithms for test generation perform the behavioral reasoning mainly on the given structural circuit information (e.g. [4], [5], [6], [7]). Sequential test generation is known to need exponential worst case effort, which is reduced by design modifications like a complete scan path ([8], [9]), a partial scan path ([10], [11], [12]), a pseudo-exhaustive technique [13] or an appropriate synthesis [14]. Sometimes such design modifications are not feasible due to area and speed restrictions and test generation has to be done for the original circuit, or the modifications only reduce the complexity of the sequential test generation process.

Although hardware verification *and* testing are needed to achieve fault free systems, both have been treated in isolation up to now. Since both have to cope with propositions about the behavioral equivalence of circuit descriptions, it is possible to combine test generation and verification which results in appropriate benefits for both.

In this paper temporal logic is used to capture the circuit behavior. This logic is often used for hardware verification, since circuit specifications may be described naturally and more complex properties may be expressed than in normal FSM verification approaches ([15], [16], [17]). Additionally, using temporal logic, the behavior of arbitrary faults at gate level may be easily modelled. Moreover, using a formal logic leads to a reduction of the hardware verification and testpattern generation problem to a satisfiability and validity problem and hence new and different approaches are possible, which are based on a separation between problem formulation and solution methods [18].

Our approach leads to a tool which supports the designer at two stages of the design process. When creating a circuit implementation, he ensures design correctness by describing his design in terms of temporal logic and verifying it against a given specification. The very same circuit description and tool is then used to generate test pattern sequences. Moreover, as a spin-off, design for testability is automatically supported, since a verifiable implementation leads automatically to better testable designs.

A related approach has been presented by Cho, Somenzi et al. which relies on similar implementation principles ([19], [20]). However, it is directly based on FSM equivalence checking and is not able to cope with circuits without reset line.

The paper is organized as follows: In section 2 some fundamentals of temporal logic are introduced. The next section shows how to model hardware behavior using this formal language. In chapter 4 an approach for hardware verification with temporal logic is presented. Then the verification problem is extended to sequential ATPG. Chapter 6 points out some optimizations which accelerate the approach. The paper ends with experimental results and a conclusion.

2. TEMPORAL LOGIC

Propositional temporal logic is frequently used in hardware verification ([16], [21], [17]). Traditional propositional logic is extended by temporal operators, which allow the expression of time varying properties as the sequential behavior of digital circuits. Moreover, propositional temporal logic is decidable, and there are constructive, fully automated decision procedures available, which are the main advantages compared to approaches based on first or higher order logics ([2], [3]). There, mechanized theorem proving is slower and often requires user guidance.

Two approaches to temporal logic theorem proving are mainly used – Computation Tree Logic (CTL) and Propositional Temporal Logic (PTL). CTL is a propositional, branching time logic, i.e. in the future many computation paths are possible [22]. A specification is given by CTL formulas and the implementation of the system to be verified is given by a state graph [17]. PTL is based on a linear sequence of discrete time points. In contrast to CTL, no explicit state graph is given and both, specification and implementation are described in PTL. It has been proposed by Manna and Pnueli as a means for verifying concurrent programs [23] and its usefulness for describing and verifying hardware has also been established ([24], [25], [16]).

Since our approach is based on PTL, its operators as well as the underlying decision procedure are explained in the following.

Formulas in PTL are constructed in the usual way of the propositional calculus. The semantics of PTL is explained based on the propositional operators $\neg$ and $\rightarrow$, but other logical connectives are used as abbreviations ($\wedge$, $\vee$, $\leftrightarrow$, $\oplus$ as *and*, *or*, *equivalence* and *exclusive-or* respectively). A formula F is built from a set $\mathbf{A}$ of variables and it is called atomic, if $F \in \mathbf{A}$ or $F = \neg p$ with $p \in \mathbf{A}$. In addition to the propositional connectives, temporal relationships are expressed by three operators. The formula $\bigcirc p$ indicates, that formula p holds in the next time instance, $\Box p$ means, that p is true in this and all following time points and $\Diamond p$ means that p is true in this or one of the following time points. The *until* operator is omitted, since it is not used in this context.

PTL formulas are defined as follows [23].

Definition 1:
a) An atomic proposition is a PTL formula.
b) If F and G are PTL formulas, then $\neg F$, $F \rightarrow G$, $\bigcirc F$, $\Box F$ and $\Diamond F$ are PTL formulas.

In the following, small letters denote atomic propositions (e.g. p) and capital letters denote compound PTL formulas (e.g. F, UC).

A formula F is called elementary, if it is atomic or $F = \bigcirc G$, i.e. F contains the next operator as its outermost connective.

Definition 2: Let $\sigma := (s_0, s_1, s_2, \ldots)$ be a sequence of truth assignments s_i: $\mathbf{A} \rightarrow \{0, 1\}$, and be $\sigma_i := (s_i, s_{i+1}, \ldots)$ the ith truncated suffix of σ. We call σ a *model* of a formula H or *H is true under* σ (denoted as $\sigma \models H$) according to the following rules:

$\sigma \models p$	iff	$s_0(p) = 1$ when $p \in \mathbf{A}$
$\sigma \models \neg F$	iff	$\sigma \not\models F$
$\sigma \models F \rightarrow G$	iff	$\sigma \not\models F$ or $\sigma \models G$
$\sigma \models \bigcirc F$	iff	$\sigma_1 \models F$
$\sigma \models \Box F$	iff	$\forall i, i \in \mathbb{N}_0 . \sigma_i \models F$
$\sigma \models \Diamond F$	iff	$\exists i, i \in \mathbb{N}_0 . \sigma_i \models F$

Definition 3: A formula is satisfiable if there exists a model for it. A set of formulas is called satisfiable, if every formula from this set is satisfiable. A formula F is called *valid* (or *tautology*, denoted by $\models F$) if $\sigma \models F$ holds for every σ.

Automated theorem proving can be done based on tableau methods in a similar way as they are used for the propositional calculus [26]. A property of proving procedures required for test generation is their constructiveness, i.e. the ability to explicitly generate models for a satisfiable formula and counterexamples if a tautology check for a given formula fails.

However, for the problems treated in this paper not the whole expressiveness of temporal logic is needed. Hence methods known from FSM equivalence checking ([27], [19], [28]) may be used to fasten the temporal logic proving process similar to suggestions of Burch, Clarke, McMillan and Dill ([17], [29]). This is described in more detail in chapter 6.3.

3. HARDWARE MODELING

To describe hardware with PTL, the model of discrete time points is mapped onto real time events. Two approaches are conceivable. Either every discrete time point is defined by a fixed time schedule or the time points mark the clock ticks of a synchronous system. Although the former possibility allows the expression of asynchronous behavior, it complicates the circuit descriptions and limits its use to small circuits. In this paper, the latter method is used. The "next"-operator indicates the values of circuit variables after the next clock signal. This modeling is not restricted to single clock systems, complex clocking schemes are allowed provided that clock transitions only occur at time points describable with PTL.

A simple example of a single clock circuit is shown in figure 1 [30].

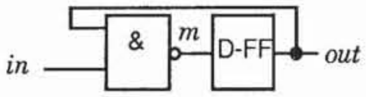


Figure 1: Example Circuit

Its behavior is described by the following set of formulas:

$$\Box\,(m \leftrightarrow \neg\,(in \wedge out))$$
$$\Box\,(\bigcirc out \leftrightarrow m)$$

The "always"-operator indicates, that the functional relationship between the input and output of the elements must hold forever. For better readability the AND-operator is omitted between the subformulas.

A netlist description of a circuit is translated into temporal logic in linear time, the PTL formulas to model the behavior of basic cells have to be stored in a library.

4. HARDWARE VERIFICATION

For performing hardware verification, both circuit specification and circuit implementation can be described as indicated in the last section. If a specification $\mathcal{S}$ of only certain circuit properties, i.e. a partial specification, as well as an implementation $\mathcal{I}$ are given in temporal logic, the formula

$$\models (\mathcal{I} \rightarrow \mathcal{S}) \tag{1}$$

must be proven [15]. Often the specification describes the complete behavior of the circuit. In that case, the behavioral equivalence of two circuit descriptions $\mathcal{S}$ and $\mathcal{I}$ must be shown

$$\models (\mathcal{I} \leftrightarrow \mathcal{S}) \tag{2}$$

The behavioral equivalence of two circuit implementations $\mathcal{I}^1$ and $\mathcal{I}^2$ has to be checked, if a design has been modified, e.g. by minimization. The necessary verification can also be performed with formula (2), but for an easier processing with a temporal proof system, a slightly different approach, based on the definition of behavioral equivalence (see e.g. [31]) is used.

Definition 4: Two circuits are behaviorally equivalent, if for arbitrary input values the correspondent primary outputs carry the same values for all time points, provided that both circuits have been initialized correctly.

If the variables o_i^1 $(i = 1, \ldots, no)$ denote the primary outputs of the first circuit and o_i^2 denote the correspondent outputs of the second circuit, the property P as defined in formula (3) states behavioral equivalence directly in terms of temporal logic ($\bigwedge$ denotes a conjunction of its arguments).

$$P := \Box \bigwedge_{i=1}^{no} \left(o_i^1 \leftrightarrow o_i^2 \right) \tag{3}$$

The equivalence $\neg\Box x \leftrightarrow \Diamond\neg x$ leads to the following uncover condition UC ($\bigvee$ denotes a disjunction of its arguments), which, if satisfiable, indicates a different circuit behavior.

$$UC := \Diamond \bigvee_{i=1}^{no} \left(o_i^1 \oplus o_i^2 \right) \tag{4}$$

Since the two circuits to be verified must be in the same starting state at the beginning of the verification process, there must be a unique reachable initial state, which is guaranteed, if all flipflops are resetable. This is expressed by the following PTL initialization condition IC, where d_i^1 and d_i^2 denote the state variables of both circuits.

$$IC := \bigwedge_{i=1}^{nd^1} (\neg d_i^1) \wedge \bigwedge_{i=1}^{nd^2} (\neg d_i^2) \tag{5}$$

The correspondence between satisfiability in temporal logic and circuit equivalence is stated in lemma 1, which is an immediate consequence of definition 4 and the definition of satisfiability in PTL.

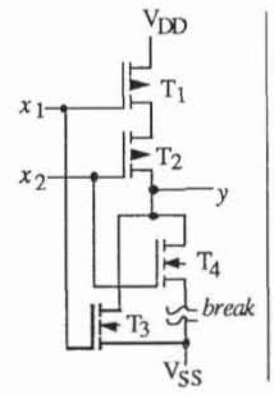
Lemma 1: Let $\mathcal{I}^1$ and $\mathcal{I}^2$ be two circuits, let UC be an uncover condition according to (4) and IC an initialization condition according to (5) in PTL. If the conjunction $\mathcal{I}^1 \wedge \mathcal{I}^2 \wedge IC \wedge UC$ is not satisfiable, then the two circuits have identical behavior.

This lemma reduces verification to theorem proving, moreover, as the proof system is constructive, an input sequence is generated automatically, which uncovers the different behavior. This input sequence may be used by the circuit designer as a hint to identify and eliminate the design errors.

5. TEST GENERATION

Test generation is performed by injecting faults given by an appropriate fault model into the description $\mathcal{I}$ of the correct circuit to get a faulty description $\mathcal{I}^F$. The behavior of $\mathcal{J}$ and $\mathcal{J}^{\varepsilon}$ is then compared to find input sequences so that the output values of both descriptions eventually differ.

The following approach is not restricted to stuck-at faults, an arbitrary erroneous behavior can be handled, if it is describable in PTL. This includes for instance stuck open faults (see figure 2).



$$\Box(\neg(x_1 \vee x_2) \leftrightarrow y)$$

Behavior of the correct circuit

$$\begin{aligned}
&\Box((\neg x_1 \wedge \neg x_2) \rightarrow y)\\
&\Box(x_1 \rightarrow \neg y)\\
&\Box((\neg x_1 \wedge x_2) \rightarrow (y \leftrightarrow state))\\
&\Box(\bigcirc\, state \leftrightarrow y)
\end{aligned}$$

Behavior of the faulty circuit

Figure 2: NOR-gate with stuck-open fault

More effort is required for considering the impact of hazards and charge-storing on transition faults. These mechanisms and delay faults can be handled by refining the grid of the time points of PTL and by adding timing specifications to the library elements. Overall this leads to a considerable increase of complexity and is not incorporated in our first implementation. Moreover, for conciseness and comparability with other approaches [32], we restrict ourselves in the following to the stuck-at fault model.

Unlike verification, the test generation must not be based on the assumption of a reset state, since an initialization sequence may be altered by the fault, the reset line may be affected or there exists a stuck-open fault with unknown starting value. Hence one must be able to deal with unknown values of storage elements at the beginning of the test generation process. For easier understanding, the case that the faulty circuit still has a reset state is discussed first, and then the general case is treated.

5.1 Circuits with Reset State

The uncover condition (4) and the initialization condition (5) defined in the former section still hold for the faulty and fault free circuit. A *test pattern sequence* is a truth assignment for the primary input variables so that at least one of the outputs of the correct and faulty circuit eventually carries a different value. If such an assignment does not exist, the fault is called *undetectable*.

As an immediate consequence of lemma 1 the following fact is proven:

Lemma 2: Let $\mathcal{J}$ be a correct circuit and let $\mathcal{J}^{\varepsilon}$ be a faulty circuit, let UC be an uncover condition according to (4) and let IC be according to (5). A satisfying variable sequence T^{ε} for the conjunction $\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge UC \wedge IC$ is a test pattern sequence for the fault.

Example: The modeling of the correct and faulty circuit behavior is demonstrated by the circuit, given in figure 3.

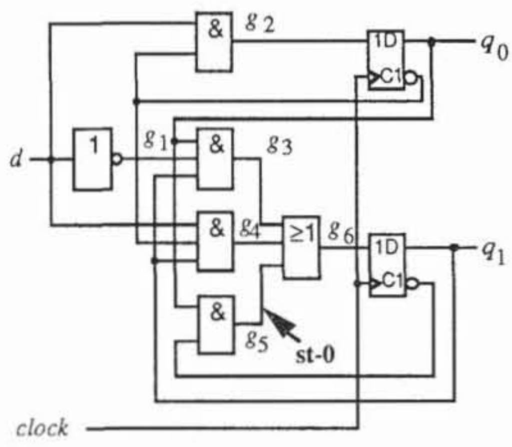


Figure 3: Example circuit taken from [33]

Given a stuck-at-0 fault at the output g_5, the behavior of the correct circuit $\mathcal{J}$ and faulty circuit $\mathcal{J}^{\varepsilon}$ is modelled by the PTL formulas, depicted in figure 4.

$$\begin{aligned}
&\Box(g_1 \leftrightarrow \neg d) && \Box(g_1^{\varepsilon} \leftrightarrow \neg d)\\
&\Box(g_2 \leftrightarrow (\neg q_0 \wedge d)) && \Box(g_2^{\varepsilon} \leftrightarrow (\neg q_0^{\varepsilon} \wedge d))\\
&\Box(g_3 \leftrightarrow (q_0 \wedge g_1 \wedge q_1)) && \Box(g_3^{\varepsilon} \leftrightarrow (q_0^{\varepsilon} \wedge g_1^{\varepsilon} \wedge q_1^{\varepsilon}))\\
&\Box(g_4 \leftrightarrow (d \wedge \neg q_0 \wedge q_1)) && \Box(g_4^{\varepsilon} \leftrightarrow (d \wedge \neg q_0^{\varepsilon} \wedge q_1^{\varepsilon}))\\
&\Box(g_5 \leftrightarrow (q_0 \wedge \neg q_1)) && \Box(g_5^{\varepsilon} \leftrightarrow 0)\\
&\Box(g_6 \leftrightarrow (g_3 \vee g_4 \vee g_5)) && \Box(g_6^{\varepsilon} \leftrightarrow (g_3^{\varepsilon} \vee g_4^{\varepsilon} \vee g_5^{\varepsilon}))\\
&\Box(\bigcirc q_0 \leftrightarrow g_2) && \Box(\bigcirc q_0^{\varepsilon} \leftrightarrow g_2^{\varepsilon})\\
&\Box(\bigcirc q_1 \leftrightarrow g_6) && \Box(\bigcirc q_1^{\varepsilon} \leftrightarrow g_6^{\varepsilon})
\end{aligned}$$

Figure 4:Behavior of the circuit $\mathcal{J}$ and $\mathcal{J}^{\varepsilon}$

The uncover condition UC is as follows:

$$\Diamond\big((q_0 \oplus q_0^{\varepsilon}) \vee (q_1 \oplus q_1^{\varepsilon})\big)$$

Since all flipflops are resetable, the following initialization condition holds

$$IC := \neg q_0 \wedge \neg q_1 \wedge \neg q_0^{\varepsilon} \wedge \neg q_1^{\varepsilon}$$

The formulas from figure 4 a) and b) as well as the uncover and initialization condition are now input to a temporal proof system to perform a satisfiability check, i.e. a variable sequence for the conjunction

$$\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge IC \wedge UC$$

has to be found. In the example, the proof system will find the following solution

$$T^{\varepsilon} := d \wedge \bigcirc\neg d \wedge \bigcirc\bigcirc \neg d.$$

This formula corresponds to a test pattern sequence (d, $\neg d$, $\neg d$) (end of example).

5.2 General Case

A test pattern sequence T^{ε} is called *generally valid* (denoted as T^{ε}_{V}), if it is a truth assignment for the primary input variables so that at least one of the outputs of the correct and faulty circuit carries eventually a different value for arbitrary initial values of the storage elements of the circuit.

Cho and Bryant generated such sequences by introducing a third value X to assign an "unknown" signal value to the flipflops [30]. Since efficient multiple-valued logic theorem tools are not generally available, an encoding of every three-valued variable by two two-valued variables is required and leads to a significantly larger search space. Moreover, this approach leads to test pattern sequences which are often not minimal. Even worse, approaches based on such a representation of unknown values are inherently incomplete, since information may get lost [33].

The approach, presented in the following avoids an explicit representation of unknown signal values and hence this drawback of incompleteness. It is based on the trivial observation, that if a test pattern sequence is generally valid, it is also a test pattern sequence for a circuit with resetable flipflops. Vice versa, in many cases the determined sequence for circuits with reset state is also a valid sequence for *arbitrary* initial values of the flipflops. This property is formally provable by using the following lemma, which states directly general validity.

Lemma 3: Let $\mathcal{J}$ and $\mathcal{J}^{\varepsilon}$ be the PTL descriptions of a fault free and faulty circuit, let UC be an uncover condition according to (4) and T^{ε} a PTL formula describing a test pattern sequence. The sequence is generally valid, if formula (6) holds.

$$\models (\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge T^{\varepsilon}) \rightarrow UC \qquad (6)$$

The following algorithm starts by generating a sequence for an arbitrary initial state. If a sequence T^{ε} has been found which is not generally valid, the test generation process is restarted to capture "missing" initial states by extending the sequence.

The function `isce` returns a formula describing the values of all state variables of the counterexample at the first time point.

Note, that the splitting into the two functions `check_sat` and `sat_seq` has been chosen only for clarification. When using a temporal proof system, both results are achieved by one pass of the system due to its constructiveness. This also holds for `check_val` and `isce`.

```
function atpg($\mathcal{J}$, $\mathcal{J}^{\varepsilon}$, UC);
{UC is determined according to (4)}
begin
valid := false;
IC := 1;{first init. state arbitrary   }
while not val do
  begin
  sat := check_sat($\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge$ IC $\wedge$ UC);
        {check satisfiability}
  if sat then
    $T^{\varepsilon}$ := sat_seq($\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge$ IC $\wedge$ UC)
        {satisfying sequence              }
  else
    return("fault undetectable!");
  val := check_val(($\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge T^{\varepsilon}$) $\rightarrow$ UC)
        {check validity                   }
  if not val then
    begin
    ISCE := isce(($\mathcal{J} \wedge \mathcal{J}^{\varepsilon} \wedge T^{\varepsilon}$) $\rightarrow$ UC);
      {initial state of counterexample}
    IC := ISCE $\wedge$ $T^{\varepsilon}$
    end
  else
    return($T^{\varepsilon}$);
  end;
end.
```

Theorem: The algorithm `atpg` finds a test pattern sequence, which is generally valid, if one exists.

Proof: The correctness of `atpg` follows immediately from lemma 3, since this property is explicitly proven in the algorithm. For proving completeness, the termination condition must be checked. The algorithm stops, if no further test sequence with the given initialization condition IC is found. At the beginning, IC leads to a sequence for an arbitrary, but fixed starting state. If no such sequence is found, trivially no generally valid sequence exists. In the second and further iterations of the algorithm, the general validity property is checked. If a sequence is not generally valid, a counterexample is generated. The values of the state variables at the first time point indicate an initial state for which the determined sequence T^{ε} is unable to uncover the fault. The proof process is restarted with this state and the already generated test sequence as an additional constraint. Therefore, a new sequence is generated which extends the old sequence to uncover the fault for this new state. The algorithm only fails, if the sequence is not extendable to comprise all possible initial states. However, this is only the case, if it is impossible to find a subsequence beginning at the *endstate* of the circuit after applying the already generated sequence, which uncovers the fault. Hence if the circuit would have been in this endstate at the beginning of the generation process, no sequence would have been possible either. Therefore no generally valid test pattern sequence exists. ■

Example: If the algorithm is applied to the example circuit from figure 3, with the new condition $IC := (\neg q_0 \wedge \neg q_1 \wedge \neg q_1^{\varepsilon})$, the result of table 1 is achieved. The flipflop q_0^{ε} can be omitted, since the fault may not propagate to that flipflop. "Chosen state" indicates the state, which has been chosen for test generation according to IC. The remaining initial states indicate the states for which T^{ε} is not a valid test pattern sequence.

Table 1: Variable Assignments for Example Circuit

IC	chosen state	T^{ε}	remaining initial states
1	$q_1^{\varepsilon} \neg q_1 \neg q_0$	$\neg d$	$q_1^{\varepsilon} q_1 \vee \neg q_1^{\varepsilon} \neg q_1$
$q_1^{\varepsilon} q_1 \vee \neg q_1^{\varepsilon} \neg q_1$	$\neg q_1^{\varepsilon} \neg q_1 q_0$	$\neg d \wedge \bigcirc d$	$q_1^{\varepsilon} q_1 \vee \neg q_1^{\varepsilon} \neg q_1 \neg q_0$
$q_1^{\varepsilon} q_1 \vee \neg q_1^{\varepsilon} \neg q_1 \neg q_0$	$\neg q_1^{\varepsilon} \neg q_1 \neg q_0$	$\neg d \wedge \bigcirc d \wedge \bigcirc^2 d \wedge \bigcirc^3 d$	$q_1^{\varepsilon} q_1 q_0$
$q_1^{\varepsilon} q_1 q_0$	$q_1^{\varepsilon} q_1 q_0$	$\neg d \wedge \bigcirc d \wedge \bigcirc^2 d \wedge \bigcirc^3 d \wedge \bigcirc^4 \neg d \wedge \bigcirc^5 \neg d$	0

Thus a generally valid test pattern sequence is found ($\bigcirc^i$ abbreviates i consecutive $\bigcirc$-operators)

$$T^{\varepsilon}_{v} = \neg d \wedge \bigcirc d \wedge \bigcirc^2 d \wedge \bigcirc^3 d \wedge \bigcirc^4 \neg d \wedge \bigcirc^5 \neg d$$

If the sequence $(\neg d, d, d, d, \neg d, \neg d)$ is applied to the circuit, a stuck-at-0 fault at the output g_5 for arbitrary initial values of the flipflops is uncovered (end of example).

6. OPTIMIZATIONS

The temporal proving process has an exponential worst case complexity with regard to the number of state variables. Optimizing the proving system, avoiding unnecessary proof runs and reducing the problem size are therefore crucial to obtain feasible runtimes.

6.1 Avoiding Unnecessary Proof Runs

In case of circuits without reset state, the number and length of the proof runs are reduced by taking advantage from the degrees of freedom in the initial condition IC: Especially when starting the algorithm, no constraints are imposed on the initial state. Hence, it is first checked, if there exists directly a state, which satisfies the given uncover condition. Thus is is always tried to extend the generated sequence by only one test vector. A real proof run is only performed, if IC forces it. Moreover, after each proof run a fault simulation is performed by a commercial fault simulator [34] to reduce the number of faults to be processed by dropping all faults, which have been also detected by the determined test pattern sequences. For this purpose, the test pattern sequences for all faults already processed are concatenated in case of not resetable flipflops. Due to the completeness of the presented approach, the fault simulator is only used for speed improvements and is *not* required for validating the test pattern sequences.

6.2 Reducing the Problem Size

There are situations in case of circuits without reset state as well as in case of circuits with reset state in which not all parts of the circuit have to be described by PTL formulas. Hence, the input to the proof system is reduced by performing a *partial modeling* of the correct and faulty circuit.

Circuit parts which will not propagate the fault to primary outputs can be eliminated in $\mathcal{J}$ and in $\mathcal{J}^{\varepsilon}$. When modeling the circuit by a directed graph, this elimination affects the predecessor nodes of all primary output nodes which are not successors of the faulty node. Furthermore, circuit parts, which are not affected by the fault can be modelled only once for $\mathcal{J}$ and $\mathcal{J}^{\varepsilon}$ (nodes which are not successors of the faulty node). Finally, when dealing with stuck-at faults, all those nodes can be eliminated, which would have been only necessary to compute the value of the faulty node.

These optimizations lead to considerable savings. When dealing e.g. with a stuck-at fault at a primary output, it is not necessary to model the faulty circuit as in that case the uncover condition only denotes, that the correspondent output of the correct circuit must eventually carry the proper logical value (e.g. 0 for a stuck-at-1 fault).

Moreover, only the predecessor nodes of that output must be modelled.

A temporal logic based approach is well suited for incorporating *user guidance*. It is easily possible to add to the circuit description initializing values (e.g. a reset signal) or sequences, the designer knows to put the circuit into a state, suited for a given fault by providing additional temporal formulas to the proving procedure.

6.3 Optimizations of the Proving System

The proving procedure can be optimized by reducing the number of nodes represented by a tableau and by implementing more efficient transition conditions than the tableau rules, originally used [26]. Both approaches can be combined.

Fujita and Fujisawa have shown, that it is possible to represent the transition conditions of the tableau with binary decision diagrams (BDDs) to reduce the representation overhead [16]. However, an explicit enumeration of all reachable nodes in the large space of the power set of all subformulas is still required ([23], [26], [21]). This large space can be reduced when using temporal logic only for representing and analyzing the behavior of digital circuits. In that case, it is possible to represent the states of the digital system with propositional state variables and the nodes of the tableau can be also encoded by a vector of state variables, which can be implemented more efficiently, compared to a characterization of states with an elementary formula labelling. Moreover, the model can be represented symbolically by a transition relation and sets of states with characteristic functions.

Coudert et al. presented a method for sequential circuit verification, which traverses the FSMs by symbolic manipulations of Boolean functions, represented as BDDs, which avoids the state explosion drawback ([35], [27], [36]). This approach has proven successful and has been refined immediately ([19], [28]). Burch et al. have shown, that the basic mechanisms are well suited for implementing model checkers for temporal logic ([17], [29]). Our own implementation is based on these approaches using the BDD-package of Brace et al. [37]. The construction process is stopped after the first satisfying variable sequence is found, so that the whole tableau of a PTL formula has to be constructed only if no solution exists.

7. EXPERIMENTAL RESULTS

The presented approach has been validated on a variety of sequential circuits. In the following, we present the results achieved on the ISCAS '89 s-benchmark set [32]. All runtimes are given in seconds and have been achieved on a SUN 4/65 workstation. Table 2 shows the results of verification runs, according to lemma 1. The compared circuits are known to have identical behavior. "Depth" indicates the maximal length of an input sequence which may be applied to the circuit before a same state is encountered again. The runtimes give a worst case estimation of the time effort needed in case of undetectable faults for circuits with reset state, if the circuit modeling has not been optimized as indicated in section VI. A undetectable fault requires at worst the same exploration of the complete state space. Hence, if the designer succeeds in the verification step he can also be sure that for each stuck-at fault a test sequence can be generated with similar computing time. Aborted faults are avoided this way.

Table 2: Verification results

circuits	depth	time in seconds
s344 $\leftrightarrow$ s349	7	59.2
s382 $\leftrightarrow$ s400	151	213.7
s526 $\leftrightarrow$ s526n	151	127.6
s820 $\leftrightarrow$ s832	11	1.5
s1488 $\leftrightarrow$ s1494	22	3.8

Table 3: Test Generation Results (resetable flipflops)

circuit	#flts	#undet faults	#test vectors	avg. ATPG time/flt	total time in seconds
s27	32	0	16	0.01	0.2
s208	215	65	135	0.4	35.3
s298	308	36	271	1.1	78.6
s344	342	5	98	31.9	837.7
s349	350	7	101	32.4	944.0
s382	399	20	1858	43.4	2405.1
s386	384	70	162	0.5	74.5
s400	424	27	1815	46.0	2868.7
s420	430	226	121	1.0	263.1
s444	474	35	1794	60.3	4103.7
s510	564	0	724	1.3	33.7
s526	555	89	2182	58.0	7745.3
s526n	553	87	2182	58.0	7626.5
s820	850	35	767	1.2	230.4
s832	870	51	785	1.2	269.3
s838	857	555	138	4.7	2725.6
s1196	1242	3	330	2.8	648.2
s1238	1355	72	336	4.5	1331.4
s1488	1486	40	1034	2.7	399.4
s1494	1506	51	970	2.6	378.5

In table 3 and 4, the close relation of verifiability and testability is obvious. Test generation time for circuits with reset state is high for all circuits, which have shown to be hard to verify. With our first implementation we were able to generate generally valid test patterns for those circuits without reset state, which had small verification times. It is apparent that the sequential depth directly influences test pattern length and runtimes especially in the case of circuits with non resetable flipflops. Since the system is based on a breadth-first traversal of the circuits, always minimal length test pattern are generated in case of circuits with reset state. If a fault is undetectable, acceptable runtimes are generally preserved, since in that case a complete exploration of the whole state space has to be performed, which is a strength of the verification oriented approach. By using "cheaper" methods like random-patterns before applying verification based techniques a considerable speed-up may be achieved for test generation [20]. However, since we want to emphasize in this paper the similarities between test and verification we renounced to elaborate these possibilities.

Table 4: Test Generation Results (non resetable flipflops)

circuit	#flts	#undet faults	#test vectors	avg. ATPG time/flt	total time in seconds
s27	32	0	14	0.1	0.7
s208	215	65	208	0.5	72.2
s298	308	35	356	20.3	1238.2
s386	384	70	185	0.8	85.5
s420	430	226	252	6.2	1692.7
s820	850	35	977	1.8	2057.7
s832	870	51	977	1.8	2063.7
s1488	1486	40	1107	5.1	1849.5
s1494	1506	51	1034	5.5	1586.7

8. CONCLUSIONS AND FUTURE WORK

Using temporal logic it is possible to generate test pattern sequences by performing a constructive proof of the formally stated testing problem. Following this approach, a method has been presented which allows test generation for arbitrary fault models and leads to a novel approach for not resetable flipflops, which avoids many drawbacks of other approaches. Thus we are able to provide one tool, which can be used for both, test generation and hardware verification.

With our prototype implementation of this design tool, we are currently able to process the small and medium sized circuits from the ISCAS '89 benchmark set. This is not a fundamental drawback since we have shown, that it is possible to reduce test generation to a satisfiability problem in formal logic as it has been done previously for hardware verification. Temporal logic model checking algorithms are subject to constant improvements so that the size of manageable circuits will further increase [38]. Moreover, it is possible to extend the approach to hierarchical circuits since hierarchy is one of the key issues of verification and many useful approaches have already been published, which can also be applied to testing ([1], [39]).

ACKNOWLEDGEMENTS

We would like to thank Karl Brace, who provided us with their BDD-package, which considerably eased the implementation of our proof system [37]. The email correspondence with Ken McMillan revealed some useful implementation hints to us. Without the continuous assistance of Oliver Seitz, the whole implementation would not have been possible.

REFERENCES

1 M. J. C. Gordon: Why High-Order Logic is a good Formalism for Specifying and Verifying Hardware; Milne/Subrahmanyam (Eds.), Formal Aspects of VLSI Design, Proc. Edinburgh Workshop on VLSI 1985, North-Holland 1986, pp. 153-178.

2 V. Stavridou, H. Barringer, D.A. Edwards: Formal Specification and Verification of Hardware: A Comparative Case Study; Proc. 25th Design Automation Conference (DAC 88), 1988, pp. 197-204.

3 K. Schneider, R. Kumar, T. Kropf: Structuring Hardware Proofs: First steps towards Automation in a Higher-Order Environment; Proc. International Conference on Very Large Scale Integration, A. Halaas, P.B. Denyer (Eds.), 1991, North-Holland.

4 R. Marlett: An Efficient Test Generation System for Sequential Circuits; Proc. 23rd Design Automation Conference, June 1986, pp. 250-256.

5 M. Schulz, E. Trischler, T. Safert: SOCRATES: A Highly Efficient Automatic Test Pattern Generation System; IEEE Transactions on CAD, Vol. 7, No. 1, January 1988, pp. 126-137.

6 W.T. Cheng: The BACK Algorithm for Sequential Test Generation; Proc. International Conference on Computer Design (ICCD 88), 1988, pp. 66-69.

7 M.H. Schulz, E. Auth: ESSENTIAL: An Efficient Self-Learning Test Pattern Generation Algorithm for Sequential Circuits; Proc. Intl. Test Conference (ITC 89), 1989, pp. 28-37.

8 M.J. Y. Williams, J.B. Angell: Enhancing Testability of Large-Scale Integrated Circuits via Test Points and

Additional Logic; IEEE Transactions on Computers, vol. C-22, pp. 46-60, 1973.

9 E.B. Eichelberger, T.W. Williams: A Logic Design Structure for LSI Testability; Proc. Design Automation Conference 1977, pp. 462-468.

10 K.-T. Cheng, V.D. Agrawal: An Economical Scan Design for Sequential Logic Test Generation; Proc. 19th International Symposium on Fault-Tolerant Computing, pp. 28-35, 1989.

11 Hans-Joachim Wunderlich: The Design of Random-Testable Sequential Circuits; Proc. 19th Int. Symp. Fault-Tolerant Computing, pp. 110-117, 1989.

12 A. Kunzmann, H.-J. Wunderlich: An Analytical Approach to the Partial Scan Problem; Journal of Electronic Testing: Theory and Applications, vol. 1, pp. 163-174, 1990.

13 H.-J. Wunderlich, S. Hellebrand: The Pseudo-Exhaustive Test of Sequential Circuits; Proc. International Test Conference, 1989.

14 S. Devadas, H.-K. T. Ma, A. R. Newton, A. Sangiovanni-Vincentelli: Irredundant Sequential Machines Via Optimal Logic Synthesis; IEEE Trans. on Computer-Aided Design, vol. CAD-9, pp. 8-18, 1990.

15 Paolo Camurati, Paolo Prinetto: Formal Verification of Hardware Correctness: Introduction and Survey of Current Research; Computer, , July 1988, pp. 8-19.

16 M. Fujita, H. Fujisawa: Specification, Verification and Synthesis of Control Circuits with Propositional Temporal Logic; Computer Hardware Description Languages and their Applications (CHDL 89), J.A. Darringer and F.J. Rammig (Eds.), Elsevier Science Publishers, North-Holland, 1989, pp. 265-279.

17 J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill: Sequential Circuit Verification Using Symbolic Model Checking; Proc. 27th Design Automation Conference (DAC 90), 1990, pp. 46-51.

18 B. Krishnamurthy: Hierarchical Test Generation: Can AI Help?; Proc. International Test Conference (ITC 87), 1987, pp. 694-700.

19 H. Cho, G. Hachtel, S.-W. Jeong, B. Plessier, E. Schwarz, F. Somenzi: ATPG Aspects of FSM Verification; Proc. International Comference on CAD (ICCAD 90), 1990, pp. 134-137.

20 F. Somenzi, H. Cho, G.D.Hachtel: Fast Sequential ATPG Based on Implicit State Enumeration; Proc. International Test Conference (ITC 91), 1991.

21 G.L.J.M. Janssen: Hardware Verification using Temporal Logic: A Practical View; Proc. Workshop Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, 1989, pp. 291-300.

22 E. M. Clarke, E. A. Emerson, A. P. Sistla: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications; ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, pp. 244-263.

23 Z. Manna, A. Pnueli: Verification of Concurrent Programs: The Temporal Framework; in "The Correctness Problem in Computer Science", R.S. Boyer and J.S. Moore (eds.), Academic Press, 1981, pp. 215-273.

24 Gregor V. Bochmann: Hardware Specification with Temporal Logic: An Example; IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982, pp. 223-231.

25 S. Bapat, G. Venkatesh: Reasoning About Digital Systems Using Temporal Logic; Proc. 23rd Design Automation Conference (DAC 86), 1986, pp. 215-219.

26 P. Wolper: Temporal Logic Can Be More Expressive; Proc. 22nd Annual Symposium on Foundation of Computer Science, 1981, pp. 340-348.

27 O. Coudert, C. Berthet, J.C.Madre: Verification of Sequential Machines Using Boolean Functional Vectors; Proc. Workshop Applied Formal Methods for Correct VLSI Design, Leuven, Belgium, 1989, pp.111-128.

28 H.J. Touati, H. Savoj, B. Lin, R.S. Brayton, A. Sangiovanni-Vincentelli: Implicit State Enumeration of Finite State Machines using BDD's; Proc. International Conference on CAD (ICCAD 90), 1990, pp. 130-133.

29 J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang: Symbolic Model Checking: 10^20 States and Beyond; Proc. 5th Annual Symposium on Logic in Computer Science, 1990.

30 K. Cho, R.E. Bryant: Test Pattern Generation for Sequential MOS Circuits by Symbolic Fault Simulation; Proc. 26th Design Automation Conference (DAC 89), 1989, pp. 418-423.

31 Z. Kohavi: Switching and Finite Automata Theory; McGraw-Hill Computer Science Series, 1970.

32 F. Brglez, D. Bryan, K. Kozminski: Combinational Profiles of Sequential Benchmark Circuits; Proc.

International Symposium on Circuits and Systems (ISCAS 89), Portland, Oregon, May 9-11, 1989, pp. 1929-1934.

33 A. Miczo: The Sequential ATPG: A Theoretical Limit; Proc. International Test Conference, 1983, pp. 143-147.

34 GenRad Inc.: System HILO, System Reference Manual; Doc. No. 2523-0101, United Kingdom, 1988.

35 O. Coudert, C. Berthet, J.C. Madre: Verification of Synchronous Sequential Machines Based on Symbolic Execution; Proc. Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, June 1989.

36 Randal E. Bryant: Graph-Based Algorithms for Boolean Function Manipulation; IEEE Transactions on Computers, Vol. C-35, No. 8, August 1986, pp.677-691.

37 K.S. Brace, R.L. Rudell, R.E. Bryant: Efficient Implementation of a BDD Package; Proc. 27th Design Automation Conference (DAC 90), 1990, pp. 40-45.

38 J.R. Burch, E.M. Clarke, D.E. Long: Representing Circuits More Efficiently in Symbolic Model Checking; Proc. 28th Design Automation Conference (DAC 91), 1991, pp. 403-407.

39 T.E. Melham: Abstraction Mechanisms for Hardware Verification; VLSI Specification, Verification and Synthesis, G. Birtwistle, P.A. Subrahmanyam (eds.), Kluwer Academic Press, 1988, pp. 267-291.