Design Error Diagnosis in Sequential Circuits

Ayman Wahba and Dominique Borrione

ARTEMIS Laboratory, Joseph Fourier University
BP. 53, 38041 Grenoble Cedex 9, France
email: Ayman.Wahba@imag.fr, Dominique.Borrione@imag.fr

Abstract. We present a new diagnostic algorithm for localising design errors in sequential circuits. The specification and the implementation may have different number of state variables, and different state encoding. The algorithm is based on the new concept of ***possible next states*** describing the possible states of the circuit due to the existence of the error. Results obtained on benchmark circuits show that the error is always found, with an execution time proportional to the product of the circuit size, and the length of the test sequences used.

1 Introduction

Although automated design tools are routinely used for digital circuits synthesis, manual changes are still being done to improve the performance, to obtain more compact structures, or to carry on small specification changes; doing so, the insertion of an unintentional error is very likely to happen. Another source of design errors is the presence of software bugs in the automated design and optimization tools.

Therefore, formal verification tools are needed to verify the equivalence between the specification and the obtained implementation, or between an initial version of the implementation and a new optimized one. However, when the two descriptions are not equivalent, these verification tools can only give a list of counter examples in the form of input patterns that can detect the existence of the error. It is then the designer's task to apply these patterns to the implementation and simulate it, to manually locate and correct the error. The verification/diagnosis cycle is repeated until a correct implementation is obtained. This process may exceed the time spent during the design phase itself. Automated diagnosis tools represent a real designers' request. In this paper, diagnosis means both locating the design error, and proposing a rectification. Exhibiting the existence of an error is assumed to be performed by verification tools.

Research in this area is very limited, and little has been published in this domain. Few automatic diagnosis systems are found in the literature, which are capable of diagnosing faults in combinational circuits only [1, 2, 3, 4, 5, 6, 7, 8, 9], but none of them could process sequential circuits.

In [10] an algorithm is presented to diagnose errors in restricted classes of sequential circuits, namely, sequential circuits without feedback loops, or sequential circuits with feedback loops which repeat their behaviour after a fixed number of clock cycles (like counters, shift registers, serial/parallel converters,

etc). This algorithm can treat other sequential circuits only if both the specification and the implementation have the same number of states and the same state encoding, or if at least a logic relation can be found between their state variables: in this case the sequential diagnosis problem is reduced to a combinational one. The algorithm uses *a greedy technique* which does not always find a solution even if one exists.

In this paper, we present a method for automatically locating and correcting single simple design errors [11] in synchronous sequential circuits. For clarity, we assume that the circuits are completely specified finite state machines. In principle, this does not affect the generality of our method since any uncompletely specified machine can be converted into a completely specified one by adding a dummy state to represent all unspecified states [12]. The implementation and the specification may have different number of states and state encoding. The specification will be regarded as a black box for which only the primary inputs and the primary outputs are observable, and which can be initialized in an initial state corresponding to the initial state of the implementation.

Principles of the method:

We introduce the new concept of *Possible Next States.* They are the set of states reachable from a given initial state, or set of initial states, due to the existence of several possible locations of the error. The implementation of the sequential circuit is represented by its *iterative logic array model.* The circuit is then simulated in each time frame separately, and diagnosed by applying combinational diagnosis rules, where the present-state lines are treated as primary inputs, and the next-state lines as primary outputs. Before proceeding to the analysis in the next time frame, the set of possible next states is computed, and then the analysis is done in the next time frame under the application of each one of these possible next states. This operation is repeated until the error is found.

Our method is the extension of previous works on combinational circuits [8]. Our basic assumption, which covers 72% of fully hand made design errors at logic level [13] is that the error is due to a single gate. Single gate errors are classified into:

- HYP-0: An extra/missing inverter.
- HYP-1: A gate replacement of type 1. (OR ↔ AND, NOR ↔ NAND).
- HYP-2: A gate replacement of type 2. (OR ↔ NAND, NOR ↔ AND).

In the framework of this paper, we consider that registers and flip/flops are basic components of the circuit description; the single gate error hypothesis thus only concerns the combinational part of the circuit (no error on the number of memory elements, nor inside them).

Difficulties in diagnosing sequential circuits:

When we perform diagnosis on a combinational logic circuit, the circuit is simulated under the application of special purpose test patterns [8]. Under the assumption of a single error, there is *only one source for the erroneous signal,* and the majority of combinational diagnostic routines are based on this fact.

When we deal with sequential circuits, instead of applying simple test vectors, we apply a sequence of vectors. We unfold the sequential circuit as a succession of combinational ones over time frames. The error is repeated in each copy of the combinational circuit.

The added complexity is due to the fact that erroneous signals may have as sources not only the error location but also the present state for which the error effect has propagated from previous times. Another major problem is that the whole iterative logic array model can't be stored in the computer memory, when the number of time frames is large.

The paper is organised as follows. Section 2 introduces the basic terminology and definitions. Section 3 presents the concept of *possible next states*, and a method to compute them. The sequential diagnosis algorithm is then presented in section 4, together with an illustration example. The experimental results on benchmark circuits are given in section 5. Section 6 presents our conclusions and directions for further work.

2 Basic definitions and terminology

Throughout this paper, we consider sequential circuits described at logic level, and synchronized by a single master clock. Such a circuit is modeled as a finite state Mealy machine $M = (I,O,R,S_0,\delta,\lambda)$, where:

- I is the set of primary inputs, $I = (I_1, I_2, ...I_n)$
  A test vector PI is a valuation of the primary inputs, i.e. an element of $(\mathcal{B}^n)$[1]
- O is the set of primary outputs, $O = (O_1, O_2, ...O_p)$
- R is a set of memory elements, $R = (R_1, R_2, ...R_m)$
  A state S is a valuation of R, i.e. an element of $\mathcal{B}^m$.
- $S_0 \in \mathcal{B}^m$ is the initial state.
- δ: $\mathcal{B}^m \times \mathcal{B}^n \to \mathcal{B}^m$ is the state transition function computing the next state, $\delta = (\delta_1, \delta_2, ...\delta_m)$
- λ: $\mathcal{B}^m \times \mathcal{B}^n \to \mathcal{B}^p$ is the output function computing the current value of the primary outputs, $\lambda = (\lambda_1, \lambda_2, ...\lambda_p)$.

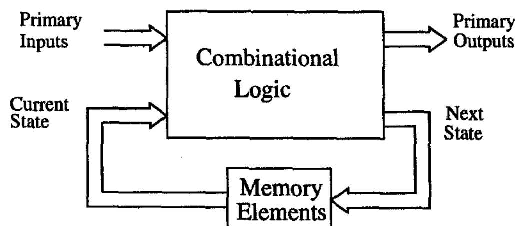

Fig. 1. A finite state machine

[1] $\mathcal{B}$ denotes the set $\{0,1\}$

In the following, to simplify the writing, we shall talk about the gates of a machine M to mean the gates of the circuit modeled by machine M.

The *iterative logic array model* of sequential circuits has been defined for sequential test generation to detect fabrication errors [14], and is shown in Figure 2.

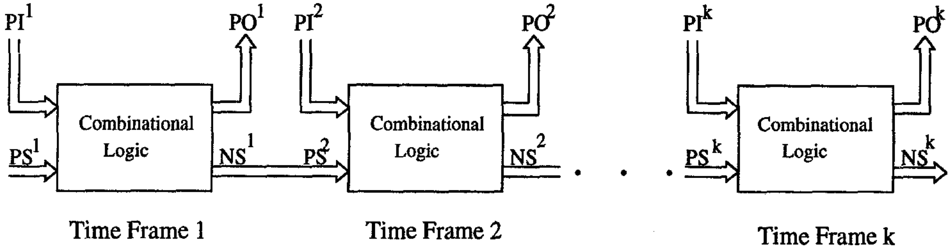

Fig. 2. The iterative logic array model

The combinational logic block of the original machine is repeated in each time frame. The inputs PI^1, PI^2, ... PI^k represent the test vectors applied to the primary inputs at each time frame. PS^i $(1 \leq i \leq k)$ represent the present-state lines, (i.e. the output of the flip/flops). PO^i $(1 \leq i \leq k)$ are the primary output values at each time frame, and NS^i $(1 \leq i \leq k)$ are the next-state lines, (i.e. the inputs to flip/flops). The present-state lines of the first time frame are set to the value of the initial state. Any other PS^i is connected to NS^{i-1}.

The test generation procedure can be stated as the generation of a sequence PI^1, PI^2, ... PI^k such that the error is detected at PO^k, if this sequence is applied to the machine starting from its initial state.

Let S be the current state of the machine M, and PI a test vector. In the following, all the definitions are relative to the current error hypothesis HYP (HYP is either HYP-0, HYP-1 or HYP-2).

Definition 1. *Search Space:*

The search space is a subset of all the circuit gates, and it contains the erroneous gate. ◇

Initially, the search space contains all the gates. It is reduced at each iteration of the diagnosis algorithm on the successive time frames.

Definition 2. *A Suspected Gate: SG(S,PI)*

A suspected gate for S is a gate of the search space which gives a correct circuit output, under the application of pattern *PI*, if it is replaced according to HYP. ◇

Definition 3. *A Correct Gate: CG(S,PI)*

A correct gate for S is a gate of the search space which gives an erroneous circuit output, under the application of pattern *PI*, if it is replaced according to HYP. ◇

Definition 4. *Modified State Transition Function:*

Let G be a gate of the search space. Function δ_G is the state transition function of M modified by changing G according to HYP. ◇

Definition 5. *Possible Next States: PNS(S,PI)*

The set of Possible Next States of S under test vector PI is the union of all the next states in which machine M can be put, by changing zero or one of the gates of the search space, according to HYP. ◇

$$PNS(S,PI) = \delta(S,PI) \cup \bigcup_{G \in \text{search space}} \delta_G(S,PI)$$

Proposition 1.

If M has a search space of g gates, then, in the worst case, the maximum number of distinct possible next states is:

$$\min\{g+1, 2^m\}$$

Proof.

The total number of machine states is 2^m. In the worst case, the replacement of each gate in the search space by another one will make a transition to a different next state. ◇

This limit is a theoretical one. In practice several gate replacements may lead to the same possible next state, depending on the circuit topology. For example, in the case of tree structured circuits, this limit is greatly reduced.

Proposition 2.

For a finite state machine M, which has m state variables, and a search space containing g gates, if the state transition function is tree structured, the upper limit on the number of possible next states is:

$$\min\{g+1, m+1\}$$

Proof.

In tree structured circuits all the gates in the cone of influence of one next state line affect only this next state line. Thus the possible next states are the state corresponding to no gate replacement, and other m states corresponding to the value change of only one of the m next-state lines. ◇

Definition 6. *Affecting Gates of a next-state line NS_i: $AG(S,PI,NS_i)$*

For a next state line NS_i ($1 \leq i \leq m$), an Affecting Gate is a gate of the search space such that, if replaced according to HYP, the value of NS_i is complemented with respect to its value when no gate is changed. We shall note $AG(S,PI,NS_i)$ the set of affecting gates for NS_i, for state S and test vector PI.

$$AG(S,PI,NS_i) = \{G \in \text{search space} \mid \delta_i(S,PI) \neq \delta_{G_i}(S,PI)\}$$

Definition 7. *Sensitive next-state lines*

A next-state line NS_i $(1 \leq i \leq m)$ is sensitive to a gate G if the value of NS_i is complemented when the gate is replaced according to HYP. We shall denote *SNSL(S,PI,G)* the set of next state lines sensitive to G, for state S and test vector PI.

$$
\begin{aligned}
SNSL(S,PI,G) &= \{NS_i \mid G \in AG(S,PI,NS_i) \wedge (1 \leq i \leq m)\} \\
&= \{NS_i \mid \delta_i(S,PI) \neq \delta_{G_i}(S,PI) \wedge (1 \leq i \leq m)\}
\end{aligned}
$$

3 Possible Next States Computation

3.1 Possible next states reachable from a single state

Let HYP be the current error hypothesis. We now present a method to get the set of possible next states of a sequential circuit for a present state S under the application of an input vector *PI*.

A direct method is to change successively each suspected gate according to HYP, and then simulate the circuit after each change to get the next state. The set of distinct next states reached is the required *PNS(S,PI)*. This method is very time consuming: if the circuit has *NG* gates, and g of them are in the search space, the processing time is proportional to $NG \times g$.

A more efficient method exploits the diagnostic algorithms for combinational circuits presented in [8]. The circuit is treated as a combinational one as shown in Figure 2. This operation is done in three steps:

Step 1: Computation of $AG(S,PI,NS_i)$, $1 \leq i \leq m$

The circuit is put in state S and simulated only once, under the application of test vector *PI*. Each next-state line NS_i is then treated as an erroneous output, and the diagnosis rules are applied to get a set of gates which can complement its value. The time required to get $AG(S,PI,NS_i)$ for all NS_i is in same order of magnitude as the time required to simulate the circuit once. Thus, the computation of the AG sets takes a time proportional to $2NG$.

Step 2: Computation of sensitive next state lines

The purpose of this second step is to compute the different sets of next-state lines sensitive to the elements of the affecting gate sets computed in step 1, using the first formula of definition 7.

Step 3: Computation of the possible next states

The first possible next state is $\delta(S,PI)$. The other possible next states are obtained by complementing in $\delta(S,PI)$ the values of the next state lines which are elements of the *SNSL* sets computed in step 2.

Example 1. Possible Next States computation:

The circuit shown in Figure 3, taken from the ISCAS'89 benchmarks [15], has four inputs $(G0, G1, G2, G3)$, one primary output $G17$, three next-state lines $(G10, G11, G13)$, and three present-state lines $(G5, G6, G7)$.

Assume the current state is $(G5, G6, G7) = (0, 0, 0)$, the input vector is $(G0, G1, G2, G3) = (1, 0, 1, 1)$, and the current error hypothesis is HYP-1 (AND $\leftrightarrow$ OR, or NAND $\leftrightarrow$ NOR replacement). In the following computations, the search space contains all the circuit gates.

Fig. 3. The circuit s27 of the ISCAS'89 benchmarks

The combinational view of the circuit is shown in Figure 4.

Fig. 4. The circuit s27 treated as a combinational circuit

Result of step 1:

$$AG((0,0,0),(1,0,1,1),G10) = \{G10, G15, G16\}$$
$$AG((0,0,0),(1,0,1,1),G11) = \{G15, G16\}$$
$$AG((0,0,0),(1,0,1,1),G13) = \phi$$

Remark:

The gates $G8$, $G9$, $G11$, $G12$, and $G13$ affect no next state lines, and are therefore not considered in the successive steps.

Result of step 2:

In this step we extract the *SNSL* sets from the *AG* sets computed in the previous step. The following results are obtained:

$$SNSL((0,0,0),(1,0,1,1),G10) = \{G10\}$$
$$SNSL((0,0,0),(1,0,1,1),G15) = \{G10,G11\}$$
$$SNSL((0,0,0),(1,0,1,1),G16) = \{G10,G11\}$$

There are only two distinct *SNSL* sets, which are: $\{G10\}$, $\{G10, G11\}$

Result of step 3: Possible next states.

Here we write the possible next states in the order $G10, G11, G13$.

$$\delta((0,0,0),(1,0,1,1)) = (0,1,0).$$

Complementing $G10$ we obtain:

$$\delta_{G10}((0,0,0),(1,0,1,1)) = (1,1,0).$$

Complementing $G10$ and $G11$ we obtain:

$$\delta_{G15}((0,0,0),(1,0,1,1)) = \delta_{G16}((0,0,0),(1,0,1,1)) = (1,0,0).$$

$$PNS((0,0,0),(1,0,1,1)) = \{(0,1,0),(1,1,0),(1,0,0)\}$$

3.2 Fixed gates and candidate gates

For the purpose of the diagnosis, it is important to know the conditions under which a possible next state is reached. These conditions are expressed in the form of the gates that must be fixed, and the gates that are candidate to be changed to go to this state.

Definition 8. *Candidate and Fixed gates for a possible next state:*

Starting from state S_1, and under the application of a test vector *PI*, the sets $candidate(S_1, PI, S_2)$ resp. $fixed(S_1, PI, S_2)$ represent the subset of the search space such that, if one and only one of its gates is replaced according to HYP, the next state of machine M is S_2, resp. is different from S_2.

$$candidate(S_1, PI, S_2) = \{G \in \text{search space} \mid \delta_G(S, PI) = S2\}$$
$$fixed(S_1, PI, S_2) = \{G \in \text{search space} \mid \delta_G(S, PI) \neq S2\}$$

Proposition 3.

$$candidate(S_1, PI, \delta(S_1, PI)) = search\ space - fixed(S_1, PI, \delta(S_1, PI)).$$

$$fixed(S_1, PI, \delta(S_1, PI)) = \bigcup_{S_2 \in PNS(S_1, PI) \wedge S_2 \neq \delta(S_1, PI)} candidate(S_1, PI, S_2).$$

Example 2. Fixed gates and Candidate gates computation

In the previous example, the computation of the candidate and fixed gates gives the following result:

Candidate gates:

$candidate((0,0,0),(1,0,1,1),(0,1,0)) = \{G8, G9, G11, G12, G13\}$
$candidate((0,0,0),(1,0,1,1),(1,1,0)) = \{G10\}$
$candidate((0,0,0),(1,0,1,1),(1,0,0)) = \{G15, G16\}$

Fixed gates:

$fixed((0,0,0),(1,0,1,1),(0,1,0)) = \{G10, G15, G16\}$
$fixed((0,0,0),(1,0,1,1),(1,1,0)) = \{G15, G16\}$
$fixed((0,0,0),(1,0,1,1),(1,0,0)) = \{G10\}$

The computation of these fixed and candidate gate sets corresponding to each possible next state is of great importance to shrink rapidly the search space of the circuit, while performing the diagnosis. This fact will be discussed in more details when we present the diagnosis algorithm.

3.3 Possible next states reachable from a set of present states

In the previous section we showed how to compute the set of possible next states reached from a given present state. In the diagnosis algorithm, we need to get the set of possible next states that can be reached from a *set of present states*, not only from a single one, except at the very first time frame.

At time frame i, the set PNS^i is the union of the individual possible next states reachable from each individual state in PNS^{i-1}, but care must be taken while computing the fixed and candidate gate sets.

When computing PNS^i we must take into consideration that only one state of PNS^{i-1} is the correct one. Since the diagnosis rules that we apply are based on the assumption of only one error, then uncorrect diagnostic decisions may be taken if the applied state of PNS^{i-1} is not the correct one. That is because in this case, multiple sources of error exist at the same time: the erroneous gate itself, and the erroneous values at the present state lines. Thus, when computing the *candidate gate set* corresponding to one of the possible next states, the actual faulty gate might not be included; even worse, it might be considered a fixed gate. In both cases, the error location would disappear from the search space.

Definition 9.

Let $PNS^{i-1} = \{S_1, S_2, \ldots S_q\}$ be the set of possible next states at time frame $i-1$, and PI be the test vector at time i. The set of possible next states at time frame i is given by:

$$PNS^i = \bigcup_{j=1}^{q} PNS(S_j, PI)$$

We shall say that a state $S_j \in PNS^{i-1}$ is a *possible predecessor* of a state $D \in PNS^i$ if $D \in PNS(S_j, PI)$. The candidate and fixed gate sets are extended to set of states as follows:

$$candidate(PNS^{i-1}, PI, D) = \bigcup_{j=1}^{q} candidate(S_j, PI, D)$$

$$fixed(PNS^{i-1}, PI, D) = \bigcap_{j=1}^{q} fixed(S_j, PI, D)$$

Justification:

If D is the required correct next state, then one of its possible predecessors say S_j must be the correct present state. The $fixed(S_j, PI, D)$ set is thus correctly computed since there is only one source of the error, and $fixed(S_j, PI, D)$ contains only correct gates. The intersection of $fixed(S_j, PI, D)$ with other fixed gate sets thus only contains correct gates, and there is no risk to consider the erroneous gate as a fixed one.

Likewise, the $candidate(S_j, PI, D)$ set is correctly computed since there is only one source of the error, and $candidate(S_j, PI, D)$ certainly contains the erroneous gate. The union of $candidate(S_j, PI, D)$ with other candidate gate sets will thus contain the error location.

If D is not the correct state, then the candidate and the fixed gate sets associated to it are not critical, and may contain anything $\diamond$.

Proposition 4.

In any time frame i, the correct next state, in which the machine must be found to behave correctly, is a member of the possible next state set PNS^i provided that HYP is the correct error hypothesis $\diamond$.

Proof by induction on time frames:

PNS^1 is computed under the application of the initial state which is sure to be correct. PNS^1 contains the states that can be reached by the replacement of *at most one gate* according to HYP. There are two possibilities:

1. The error does not affect the next-state lines, in which case the state obtained with no gate replacement is the correct one.
2. The error affects the next-state lines and thus one of the gate replacements leads to a possible next state which is the correct one.

Assume the correct state exists in PNS^{i-1}, the same reasoning concludes that PNS^i contains the correct state $\diamond$.

Based on this proposition and the definitions of the candidate and the fixed gate sets, the erroneous gate never exists in fixed gate sets, and always exists in candidate gate sets, provided that HYP is the correct error hypothesis.

4 The Sequential Diagnosis Algorithm

The diagnosis starts by applying a sequence which detects the existence of the error. This sequence may be a counter example provided by a verifier, or by any other means.

In the first time frame, the first test vector of the test sequence is applied to the primary inputs, and the initial state is applied to the present-state lines. The primary output values obtained from the implementation are then compared with those of the specification, and combinational diagnosis rules [8] are applied to get sets of *suspected* or *correct* gates depending on whether the values of primary outputs are correct or wrong.

In a general time frame i and under the application of the test vector PI^i, the same operation is repeated with each one of the possible next states $S_j \in PNS^{i-1}$, and in each case a set of suspected gates $SG(S_j, PI^i)$ or correct gates $CG(S_j, PI^i)$ is computed. The role of the fixed and candidate gate sets, computed for possible next states, is now discussed.

The error is detected with S_j and PI^i:
The state $S_j \in PNS^{i-1}$ can only be reached by the change of any one of the gates of $candidate(PNS^{i-2}, PI^{i-1}, S_j)$. Therefore, all the gates $G1 \notin candidate(PNS^{i-2}, PI^{i-1}, S_j)$ are removed from the suspected set $SG(S_j, PI^i)$ generated by the diagnoser.
Moreover, the state S_j cannot be reached if any gate $G2 \in fixed(PNS^{i-2}, PI^{i-1}, S_j)$ is changed. Thus all $G2 \in fixed(PNS^{i-2}, PI^{i-1}, S_j)$ are removed from the suspected set $SG(S_j, PI^i)$. This will cause $SG(S_j, PI^i)$ to diminish rapidly.

The error is not detected with S_j and PI^i:
The state $S_j \in PNS^{i-1}$ can only be reached if all the gates in the set $fixed(PNS^{i-2}, PI^{i-1}, S_j)$ are kept unchanged. Thus $fixed(PNS^{i-2}, PI^{i-1}, S_j)$ must be added to the correct gate set $CG(S_j, PI^i)$ obtained by the diagnoser. This will increase the size of the correct gate set, and consequently will cause the search space to diminish rapidly.

The same reasoning used in the previous section, for computing the fixed and the candidate gate sets for a possible next state, is also used here. The suspected gate set obtained from the circuit analysis in time frame i, is given by:

$$SG(PNS^{i-1}, PI^i) = \bigcup_{S_j \in PNS^{i-1}} SG(S_j, PI^i)$$

and the correct gate set is given by:

$$CG(PNS^{i-1}, PI^i) = \bigcap_{S_j \in PNS^{i-1}} CG(S_j, PI^i)$$

It is to be noted that the size of the search space decreases with each new time frame. Initially, the search space contains all the circuit gates. At time frame i, the search space is updated as follows:

$$Search_space^i = Search_space^{i-1} \wedge SG(PNS^{i-1}, PI^i) - CG(PNS^{i-1}, PI^i)$$

If there remains more than one gate in the search space after diagnosing the circuit with the first given sequence, extra sequences are needed. For each one of the remaining gates G, a test sequence capable of detecting the replacement of G by another one according to HYP is generated. The circuit specification is then simulated under the application of this sequence, and its output values are compared with those of the implementation. If the error is not detected, then G is not the erroneous gate, otherwise the error shoud have been detected. This gate can, therefore, be removed from the search space. If the error is detected nothing can be said about gate G, and then diagnosis is performed normally as was done with the first detecting sequence.

This operation is repeated for all remaining gates in the search space until its size becomes one, or until sequences are generated for all remaining gates.

If, at any time, the updating of the search space results in the empty set ϕ, the selected error hypothesis is not correct, and the diagnosis must be restarted with another one.

We present here the complete diagnosis algorithm:

```
program diagnose;
begin
    get_first_sequence(FS);
    for an error hypothesis HYP do
        Search_space := all circuit gates;
        States := Initial_state;
        sequential_diagnose(FS);
        if Search_space contains more than one gate then
            for every gate G ∈ Search_space do
                generate an error detecting sequence EDS(G);
                if Impl_out(EDS) := Spec_out(EDS) then
                    Search_space := Search_space - G;
                else
                    sequential_diagnose(EDS)
                endif
            endfor
        endif
    endfor      % At this point Search_space contains one or more gates among
end.            % which the erroneous one.
```

```
procedure sequential_diagnose(Sequence);
begin
   for every vector PI in Sequence do
      SG := SG(States, PI);
      CG := CG(States, PI);
      Search_space := (Search_space ∧ SG) − CG;
      States := PNS(States, PI);
      if Search_space := φ then
         change error hypothesis and restart;
      endif
   endfor
end
```

Example 3.

In the circuit of example 1, assume that $G10$ is erroneous and should be a NAND gate. A test sequence that detects the error is $(G0, G1, G2, G3) = ((1, 0, 1, 1), (1, 0, 0, 1))$. Starting from the state $(G10, G11, G13) =$ (0,0,0), and under the application of this test sequence, the error is discovered in the second time frame.

Here we show the steps of the diagnosis algorithm, under error hypothesis HYP-1.

Initially, the search space contains all the circuit gates, except inverters

$$Search_space = \{G8, G9, G10, G11, G12, G13, G15, G16\}$$

First time frame:

After the application of the input (1,0,1,1) the error is not detected at the output $G17$, because both the correct and the wrong implementation generate the same value '0'. The diagnosis results in a correct gate set $CG = \{G15, G16\}$. Thus the new search space is reduced to:

$$Search_space = \{G8, G9, G10, G11, G12, G13\}$$

The possible next states set PNS^1 is computed as shown in example 1, but here we must take into consideration that the *Search_space* is restricted to a subset of all the circuit gates. The possible next states and their associated fixed and candidate gate sets are shown in the following table:

$(G10, G11, G13)$	*fixed*	*candidate*
(0,1,0)	$\{G10\}$	$\{G8, G9, G11, G12, G13\}$
(1,1,0)	$\{G8, G9, G11, G12, G13\}$	$\{G10\}$

Second time frame:

In this time frame the input vector (1,0,0,1) is applied, while the present state may be any element of the set PNS^1.

- When the present state is (0,1,0):
  . the error is detected.
  . $SG((0,1,0),(1,0,0,1)) = \phi$.
  . $CG((0,1,0),(1,0,0,1)) =$ fixed set $= \{G10\}$.
- When the present state is (1,1,0):
  . the error is not detected.
  . $SG((1,1,0),(1,0,0,1)) =$ candidate set $= \{G10\}$.
  . $CG((1,1,0),(1,0,0,1)) = \{G8, G9, G11, G12, G13\}$.

If we find the intersection of all CG sets, and the union of all SG sets, we get:

$$SG(\{(0,1,0),(1,1,0)\},(1,0,0,1)) = \{G10\}.$$
$$CG(\{(0,1,0),(1,1,0)\},(1,0,0,1)) = \phi.$$

Now the *Search_space* contains only the gate $G10$ which is really the erroneous gate.

If the diagnosis is performed under HYP-0 or HYP-2, the search space at the end of the sequence contains the gates $G10$ and $G13$. An extra test sequence, $\{(0,0,0,1),(1,0,0,1)\}$, is then generated in both cases, and the diagnosis under its application yields an empty search space, thus allowing to discard these hypotheses.

5 Experimental Results

The proposed algorithm has been implemented in a fully home-made package, in PROLOG. The package consists of three basic modules: a sequential test pattern generator, a sequential simulator, and a diagnoser. Experiments were made on the ISCAS'89 benchmark circuits [15]. In every experiment an error of random type was inserted at random location in the circuit, and then the diagnosis algorithm was applied. The diagnosis starts by applying an error detecting pattern, which is assumed to be supplied by a verifier, but in our experiments the detecting pattern is generated in a preprocessing phase.

Table 1 shows the characteristics of each one of the tested circuits. The column headed *loops per flip/flop* indicates the average number of loops in which each flip/flop takes part. This value gives an idea about how strongly different loops are tied together. It was shown in [16] that when this value increases, the test pattern generation for detecting a given fault becomes more difficult. Column six gives the number of primary inputs which affect the state variables. The smaller this number, the longer the generated test sequences.

Table 2 shows the obtained diagnosis results. The second column indicates the number of experiments made on each individual circuit: each experiment is the insertion of a different gate error. The third column shows the average number of extra test sequences generated in addition to the first one supplied by the preprocessing phase. Each one of these sequences consists of one or more test vectors. The average number of vectors in all sequences is shown in the fourth column. This value represents the number of time frames in which the

Circuit name	Flip/Flops	Gates	Loops per flip/flop	Primary inputs		Primary outputs
				Number	State controlling	
s27	3	10	1.3	4	4	1
s208	8	96	1.8	11	2	1
s298	14	119	1.8	3	3	6
s344	15	160	3.5	9	9	11
s349	15	150	3.1	9	9	11
s382	21	158	2.1	3	3	6
s386	6	159	8.0	7	6	7
s400	21	148	2.2	3	3	6
s420	16	196	1.8	19	2	1
s510	6	211	11.8	19	19	7
s641	19	379	2.9	35	15	24
s820	5	289	9.2	18	18	19
s838	32	390	1.8	35	2	1
s953	29	395	1.8	16	16	23
s1196	18	529	0.0	14	14	14
s1423	74	657	3.3	17	17	5
s5378	179	2779	2.9	35	35	49

Table 1. Characteristics of the benchmark circuits

circuit was analysed before a diagnostic decision is made. The average CPU time, expressed in seconds, on a SUN SPARC station 10, is shown in the fifth and sixth columns. The *diagnosis time* is the time required to simulate the circuit, to calculate possible next states and to apply the diagnostic rules. The *total time* takes also into consideration the time required to generate the extra test sequences. The last column of Table 2 shows the average number of suggested gate replacements proposed by the diagnoser. The real error was always found within this set of suggestions.

It is to be noted that the diagnosis time is proportional to the product of the *total number of vectors* and the *number of circuit gates*. This product reflects the number of gates traversed and processed during the diagnosis process. The total number of vectors depends on the length of the test sequences which are, in turn, related to the number of *loops per flip/flop* and the number of *state controlling inputs*. If we compare, for example, the circuits *s420*, and *s641*, we find that *s641* has a larger number of gates (379 against 196), and a larger number of flip/flops (19 against 16), however the diagnosis time with *s641* is less (128 seconds against 775 seconds). This is due to the fact that, in average, *s641* is processed over 4 time frames, while *s420* is processed over 79 time frames.

The CPU times mentioned here are based on our experimental implementation of the algorithms. We didn't devote a large effort to program optimization, as our main interest was to show the applicability of our approach. We believe that these time performances can be significantly improved by rewriting the software using faster languages like the C language.

Circuit name	Number of experiments	Average				
		# Sequences	Total # vectors	Diagnosis time	Total time	Suggestions
s27	14	1.79	2.43	0.22	0.61	1.64
s208	114	6.53	132.79	184.40	251.05	3.44
s298	126	2.35	17.62	51.53	59.89	2.09
s344	174	1.79	6.55	51.13	64.88	1.16
s349	177	1.77	6.53	55.15	69.55	1.26
s382	104	2.74	125.21	1071.43	1298.99	2.37
s386	222	6.66	18.10	12.45	49.18	3.26
s400	90	2.23	69.59	802.97	1009.89	2.22
s420	95	4.84	79.18	566.73	775.13	4.02
s510	209	2.94	18.07	47.12	127.75	2.32
s641	174	1.39	4.07	128.71	306.79	1.06
s820	111	3.23	8.41	25.48	199.52	2.31
s838	10	3.70	129.70	8084.73	8427.23	3.60
s953	61	4.72	20.61	1151.84	1360.41	3.95
s1196	378	1.78	2.69	93.01	167.37	1.91
s1423	10	4.80	211.00	10323.87	14245.10	3.60
s5378	10	7.10	20.30	12936.69	14445.42	6.30

Table 2. Diagnosis results for the benchmark circuits

The above experiments were aimed at providing performance data. For each of the ISCAS'85 benchmarks, we only have an implementation, but no abstract specification. The initial benchmark is used as specification, while the implementation is obtained from it by inserting random errors. Thus, both the good and the erroneous circuits have the same state encoding.

To show that our method also applies when the specification and the implementation have different state encodings, an additional benchmark is now discussed: the controller of an elevator, initially proposed by Hans Eveking [18]. The specification is given by a behavioral VHDL description, and the implementation, synthesized manually, is given by a netlist. The specification has 9 state variables. The implementation has 125 gates and 6 flip/flops. Both have the same 7 inputs and 3 outputs. We purposely used a different state encoding in the specification and in the implementation. We first proved the equivalence using the LOVERT FSM equivalence checker, then we performed 85 tests, inserting each time one random error in the implementation. Diagnosis was performed against the behavioral specification, and the error was always found.

Table 3 shows the average values of the number of sequences, vectors, diagnosis time, total time, and the number of error candidates found.

6 Conclusions

A new approach for single design error location and correction in sequential circuits has been presented. The algorithm is based on a combinational diagnosis

Number of experiments	Average				
	# Sequences	Total # vectors	Diagnosis time	Total time	Suggestions
85	3.05	15.93	22.71	34.38	1.66

Table 3. Diagnosis results for the elevator circuit

algorithm, and the concept of *possible next states*. Experimental results, on a set of benchmark circuits, showed that this algorithm could limit the suspected error locations to a small number of candidates, independent of the circuit size, in a reasonable time. The time performance of the algorithm is proportional to the product of the circuit size and the length of the test sequences used. An efficient tool that can generate short error detecting sequences will significantly improve the performance.

For instance, a BDD-based finite state machine verifier, that performs the symbolic state space traversal to check the equivalence between the specification and the implementation, would produce the shortest possible error detecting sequence(s) [17]. We do not possess such a tool that can process all the circuits of our benchmark set. This is the reason why, in our experimental software, we developed a prototype test pattern generator, which does not always guarantee shortest sequences.

Knowing that the length of an error detecting sequence is minimal can be used to further optimize the diagnosis algorithm: possible next states from which the error is detected on the circuit outputs prior to the last time frame can be discarded. The results of Table 2 were obtained without this optimization.

The single gate error fault model is practically valid for manual changes, if the verification is made frequently during the design change procedure. In this case the probability of inserting multiple errors is not very high. Conversely, this model does not apply to the detection of errors performed by synthesis software.

The problem of locating and correcting multiple errors is still a challenging one. We are currently working on this problem for combinational circuits, and encouraging first results have been obtained, but still on a restrictive model.

References

1. J. C. Madre, O. Coudert, J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM," *Proceedings of ICCAD'89*, pp. 30-33, 1989.
2. K. A. Tamura, "Locating Functional Errors in Logic Circuits," *Proceedings of 26th Design Automation Conference (DAC'89)*, pp. 185-191, 1989.
3. S. Y. Kuo, "Locating Logic Design Errors via Test Generation and Don't Care Propagation," *Proceedings of EURO-DAC'92*, pp. 466-471, 1992.
4. P. Y. Chung, Y. M. Wang, I. N. Hajj, "Diagnosis and Correction of Logic Design Errors in Digital Circuits," *Proceedings of 30th Design Automation Conference (DAC'93)*, 1993.

5. A. M. Wahba, E. J. Aas, "Verification and Diagnosis of Digital Systems by Ternary Reasoning," *Lecture Notes on Computer Science* No. 683, Springer Verlag, pp. 55-67, May 1993.
6. Q. H. Zhang, C. Trullemans, "Logic Verification of Incomplete Functions and Design Error Location," *Lecture Notes on Computer Science* No. 683, Springer Verlag, pp. 68-79, May 1993.
7. M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano, "Rectification of Multiple Logic Design Errors in Multiple Output Circuits," *Proceedings of the 31st Design Automation Conference(DAC'94)*, 1994.
8. A. Wahba, and D. Borrione, "Design Error Diagnosis in Logic Circuits using Diagnosis-Oriented Test Patterns," *Research Report RR-940-I*, ARTEMIS-IMAG, Grenoble, France, June 1994.
9. Q. Zhang, "Logic Verification and Design Error Diagnosis for Combinational Circuits," *Ph.D. Thesis*, Université Catholique de Louvain, Belgium, Feb. 1995.
10. M. Fujita, "Methods for Automatic Design Error Correction in Sequential Circuits," *Proceedings of European Conference on Design Automation with The European Event in ASIC Design*, 1993, pp. 76-80, 1993.
11. M. S. Abadir, J. Ferbuson, and T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 1, pp. 138-148, January 1988.
12. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Book Edition, 1978.
13. E. J. Aas, K. A. Klingshheim, and T. Steen, "Quantifying Design Quality: A Model and Design Experiments," *Proc. EURO-ASIC'92*, IEEE Computer Society, pp. 172-177, 1992.
14. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, New York: Computer Science Press, 1976.
15. F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proceedings of International Symposium of Circuits and Systems (ISCAS'89)*, Portland, OR, May 1989.
16. A. Lioy, P. L. Montessoro, and S. Gai, "A Complexity Analysis of Sequential ATPG," *Proceedings of IEEE International Symposium of Circuits and Systems (ISCAS'89)*, pp. 1946-1949, May 1989.
17. G. Cabodi, P. Camurati, S. Quer, "Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversal," *Proceedings of EURO-DAC'94*, pp. 22-27, Grenoble, France, Sept. 1994.
18. D. Borrione, H. Eveking, "Formal Verification of Hardware Designs," To appear in *Journal of the Brazilian Computer Society, Special Issue on Electronic Design Automation*, Nov. 1995.