TALLINN TECHNICAL UNIVERSITY

DESIGN ERROR DIAGNOSIS IN DIGITAL CIRCUITS

BY

ARTUR JUTMAN

A Master Thesis
Submitted to the Chair of Computer Engineering and Diagnostics
of the Department of Computer Engineering

In fulfillment of the requirements for the
Degree of Master of Science
Computer Engineering

TALLINN
June 1999

TABLE OF CONTENTS

DIGITAALSKEEMIDE DISAINIVIGADE DIAGNOSTIKA

Koostaja: Artur Jutman

ANNOTATSIOON

VLSI skeemide projekteerimine muutub iga aastaga üha keerukamaks seoses projektide mahukuse ja keerukuse kasvuga. Mida keerulisem on projekt, seda suurem tõenäolisus vigade tekkimiseks eri projekteerimise staadiumites. Algstaadiumides mitte avastatud vead võivad olla põhjuseks projekti maksumuse mitmekordistamises, projekteerimise aja märgatavas pikenemises ja valmistoodangu realiseerimise viivitamises. Antud asjaolud sunnivad teadlasi otsima ja välja töötama meetodeid projekteerimise vigade kiireks leidmiseks ja nende parandamiseks.

Antud magistritöös on võetud vaatluse alla projekteerimise vigade diagnostika digitaalskeemides, on tehtud lühike analüüs olemasolevatest meetoditest ja on pakutud uus lähenemismeetod antud probleemi lahendamiseks.

Juhendaja: Prof. Raimund Ubar

1. INTRODUCTION

Diagnostics finds place in many various aspects of our life. In medicine, for example, it is a human body condition determination. In industry it is a certain device condition determination. Such a device is called a *unit under test*. *Diagnosis* is a process of the unit under test analysis. The result of the diagnosis is a conclusion about the condition of the device. The conclusion can be, for example, one of the following: the device is working properly, the device is not working properly, and there is a defect in the device. Diagnosis is done in the field of digital electronics in order to discover defects in a digital system which could be caused either during manufacture or because of wear-out in the field. Testing for manufacturing defects is done at various stages in the production of a system: the dies are tested during fabrication, the packaged chips before placing in the boards, the boards after assembly, and the entire system is tested when complete. Testing for serviceability of a device is done during the whole lifetime of a device.

The choice of test patterns for each of these tests is determined by factors such as the time available for test, the degree of access to internal circuitry, and the percentage of failures which are required to be detected. The theory and techniques of testing for manufacturing defects or operation faults are thoroughly studied in literature. The basics, for example, are given in [35, 36].

However, before the device is sent for manufacturing, it goes through many stages of design process. Each stage may be performed numerous times before an acceptable solution is obtained. The whole process is also iterative and may require circuit modifications in order to perform trade-offs. These modifications can be inserted either by means of automated designing tools or manually. For example netlists generated by the synthesis tools very often have to be modified by designers to improve timing performance, to obtain more compact structures, or to carry out small specification changes. The whole VLSI design process is getting more and more complicated due to continuously growing sizes and complexity of the projects. Due to this fact the error emergence probability through different design stages is big enough. An error not detected at the earlier stages may cause a cost explosion of the project and significantly increase the time-to-market. This fact forces the research in the direction of development fast and efficient methods for design error detection and rectification.

The concept of *design error* slightly differs from one paper to another. In [5], for example, a design error is referred to an error in a specification but most authors treat the design error as an error in an implementation in relation to the specification. To avoid confusion, the latter concept is chosen in this work. An example of a design error could be a gate substitution error, for example, AND gate is replaced by OR gate in the implementation. It could be also a bad or missing connection between gates.

Design error detection is usually performed at a validation phase using symbolic or simulation based design verification methods. Conventional verification tools can discover the functional difference between the specification and the implementation and provide counter examples in the form of input patterns that witness a difference between the implementation and the specification behaviors. Using these counter examples as input patterns, design engineers simulate the implementation to manually correct the

design. This process is called *design error rectification*. Here is where automatic methods may replace a lengthy manual correction saving a lot of debugging time. No matter what way of correction is applied the result must be verified again and the same diagnosis-correction-verification cycle is repeated until a correct design is obtained.

There are several other applications of design error diagnosis techniques in fields that are closely related to the problem. For example, verification of a new version of design that is functionally equivalent to the previous one but has another timing or structure.

Another one problem closely related to automatic error correction is *engineering change* [12]. It is called also *incremental synthesis* in [22, 23]. An old specification, an old implementation, and a new specification are given. . The goal is to create a new implementation fulfilling the new specification while reusing as much of the old implementation as possible. Suppose a specification of the design is slightly changed and a lot of engineering effort have already been invested (e.g., the layout of a chip may have been obtained). In this case it is desirable that such changes will not lead to a very different design allowing the reuse of the existing investment on the implementation. The quality of engineering change is measured by the amount of logic in the old implementation reused in the final new implementation.

A couple of ideas how else the design error diagnosis can be used are given in [13]. The first one is a trick for logic minimization. The authors propose to inject intentionally two errors into an implementation and see if it can be corrected at just one location. The second application is debugging CAD tools that cause incorrect implementations. Software bugs existing in gate-level, timing, layout optimization, or technology mapping tools could be discovered diagnosing the incorrect implementation.

The application domain of formal verification is no longer restricted to providing correctness of a specific design but it can also be used as a debugging tool and therefore helps speeding up the whole design cycle [2].

The next chapter of the thesis gives a general information about design error diagnosis issues and its location in a design process. A brief overview of existing approaches and requirements that make a design error diagnosis method good are presented in chapter 3. Chapter 4 describes a new approach for design error localization, based on stuck-at fault model. The program description and analysis of experiments based on this new method are presented in chapter 5. Finally, chapter 6 concludes the work.

2. PRELIMINARIES

Design of a digital system, as it was mentioned above, is a highly complicated process containing loops of repeating phases. Let us consider it in more detail and see where the design error diagnosis is located in the design process. We will also refine the problem of design error diagnosis in this section.

2.1 DESIGN AND VERIFICATION

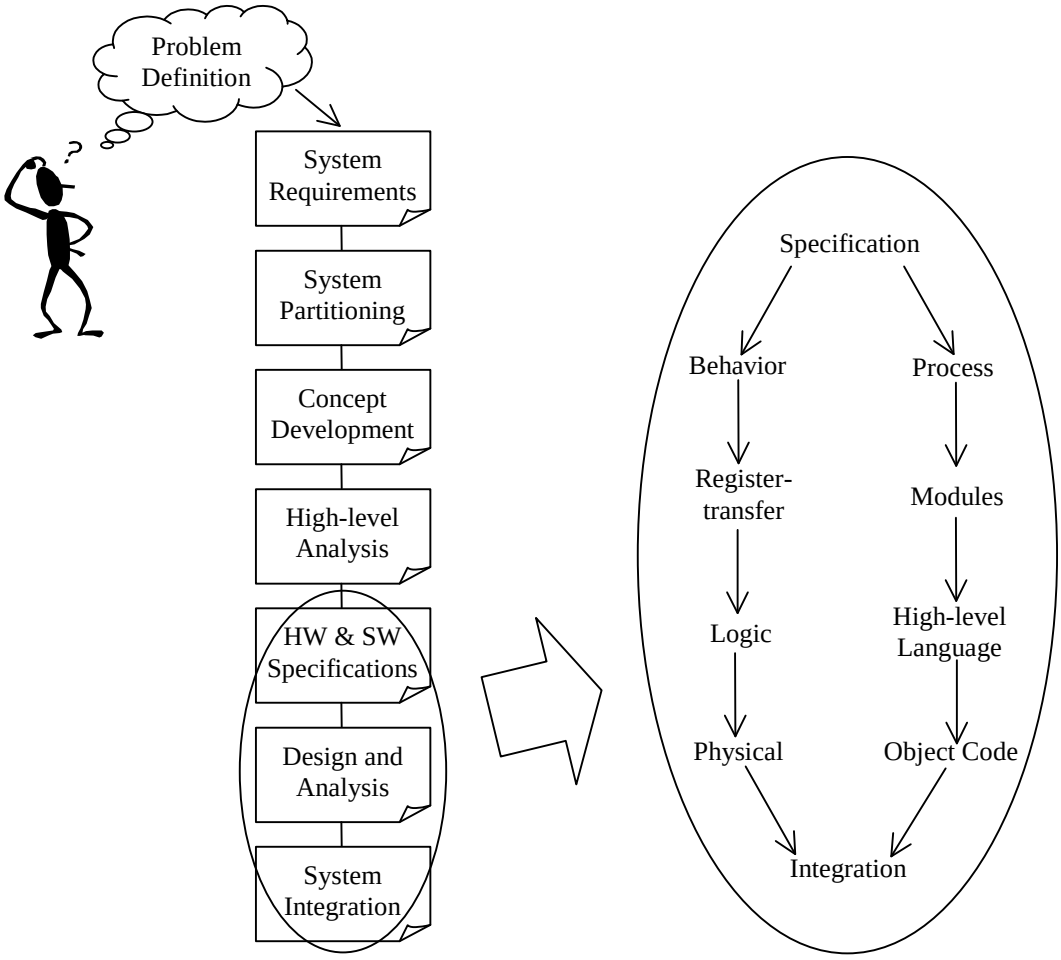


Fig. 2.1 ***A top-down design process***

The main stages of design process are shown in Fig. 2.1 according to [37] and [34]. The first step in any design process is to develop a *description of a problem*. For example, the problem may be to build a control system for an aircraft or a plant. The second step is to extract a set of *requirements* from problem description. They include typically cost, weight, size, power consumption, reliability, and so on. Then goes the *system*

partitioning step when the whole design is divided into set of smaller subsystems that can each be handled easily by individual teams of design engineers. During the *concept development* step several design approaches are considered and then analyzed in detail during *high-level analysis* step to find out advantages and disadvantages of one approach compared to another. Before the designers start the development of the final implementation, hardware (HW) and software (SW) *specifications* must be drawn up. The specifications must achieve a delicate balance necessary to meet system requirements and, at the same time, produce a practical, implementable, and manageable design. Then the *design and analysis* phase begins, when the HW and SW parts are concurrently developed. The analysis must be performed continuously during this phase to ensure that the system requirements are not compromised by design decisions that are made, and that hardware decisions do not negatively impact the software, and vice versa. The final step is the *system integration*, when all the subsystems, HW and SW parts are combined together and tested. If it is found that some element is not working properly in the system, then the design process for this element or several ones is repeated partly or completely.

In order to achieve a reliable operation and a correspondence of device behavior to initial requirements with minimum costs, the design verification must be performed continuously in parallel to the design process. First of all, according to [37], the *requirements design review* must be applied to ensure that all necessary requirements have been identified, that they are reasonable and will be verifiable during the analysis and testing process. Then the *conceptual design review* occurs immediately after high-level analysis step, to assure that the basic concept of a concrete candidate design is correct and meets the requirements that have been developed. During the *specifications design review* it is important to make sure that the specifications are both correct and understood correctly. Also, it is extremely important to guarantee that the designs will be testable. The *detailed design review* must encompass both the detailed design and the detailed analysis. Its purpose is to ensure that the specific designs meet the specifications and are capable of fulfilling the system requirements. The last checkpoint in the design process is the *final review*, the basic purpose of which is to examine the performance of the prototype and the results of the final analysis, to ensure that specifications have been adhered to and the system requirements have been met. If the design has been performed correctly and the design reviews have been successful throughout the design process, the final review should not uncover any major problems.

On the right part of Fig. 2.1 the design and analysis stage of the design process is presented. The right part inside the oval belongs to a software design and the left one is a hardware design part. After a gate-level logic is synthesized from the register-transfer level (RTL) description it must be compared to the specification so that to detect any inconsistency. This is the exact point where the design error diagnosis is applied in order to find the erroneous area in the logic implementation. This procedure belongs to the most important and difficult part of the review process, the detailed design review.

2.2 THE PROBLEM OF DESIGN ERROR DIAGNOSIS

Generally, the problem is defined as follows. Given a correct specification and an implementation. The goal is to find whether the implementation is correct or have errors inside. In the latter case, the task is also to localize the possibly minimum areas that contain the errors. In practice, gate level logic is treated as the implementation. The representation form of the specification varies from one approach to another. It could be

a description on either behavioral or RTL level or a graph representation of a circuit. Some structure-based approaches [5, 7, 9, 12] require also the information about structure of the “golden device” in the specification.

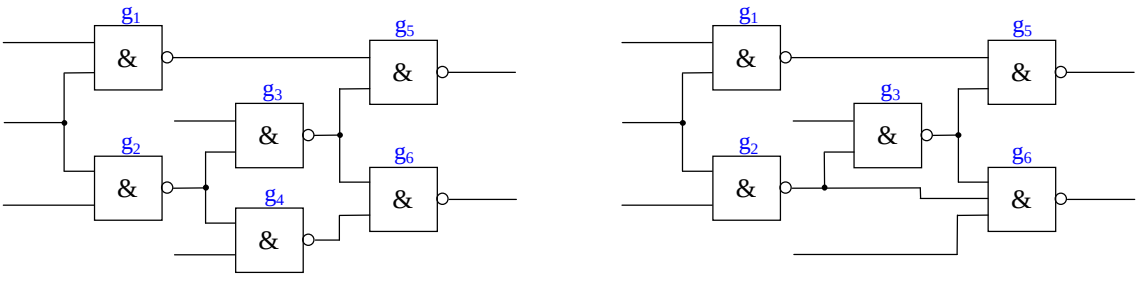
We can try to check the physical level circuit for design errors by design error diagnosis technique too. For this purpose, we would turn everything upside-down. Suppose, we have the logic-level circuit checked for errors and found it to be correct. Then, we take the physical circuit as the specification and the logic as the implementation and apply a design error diagnosis method. If the implementation is proven to be incorrect, for example, AND gate should be somewhere instead of NOR gate, we can suppose that the NOR gate from the logic level has been improperly implemented in the physical level, where it is appears as the AND gate. If we know the exact place of the physical-level AND gate, the erroneous area could be then identified.

2.2.1 Design Error Model

Most of design error diagnosis methods rely on an error model. That means that in case of a certain behavior of a circuit, a certain type of error is suspected to be presented in it. In [20] a widely adopted *simple design error model* is proposed by Abadir et al. that includes following basic categories:

- functional errors of gate elements
  - gate substitution
  - extra gate
  - missing gate
  - extra inverter
  - missing inverter
- connection errors of signal lines
  - extra connection
  - missing connection
  - wrong connection

Gate substitution error means that a gate *g* is implemented by a wrong type of gate, for example, an AND gate is used instead a NOR. *Simple extra gate* means that two wires are accidentally gated together before feeding to another gate. *Simple missing gate* refers to the situation where a gate *g* is omitted and its inputs go straight to the place where its outputs should normally go (Fig. 2.2).



a) Correct circuit b) Missing gate g_4

Fig. 2.2 ***Missing gate example***

Extra/missing inverter means that an inverter is accidentally inserted/omitted on some line. *Extra connection* means that a wire is accidentally added from the output of a gate g_i to the input of another gate g_j. *Missing connection* refers to the reversed situation of extra connection. *Wrong connection* means incorrectly placed gate input. An input to some gate g_k should have come from gate g_i, but is mistakenly drawn from another gate g_j. Gates in the model are restricted to primitive gates, i.e. AND, OR, NAND, NOR, XOR or XNOR, for simplicity.

The disadvantages of using any error model are manifest when a real design error existing in implementation is not describable by the chosen model. However, in practice, the range of commonly encountered error types is limited. An experimental study described in [24, 25] has shown that the simple design error model covers about 98% of all the errors made by students. By the way, most frequently encountered error types were extra/missing inverter errors (Fig. 2.3). It is sensible then to start the diagnosis from inverter error verification continuing with all the others error types from the chosen error model in case the former has failed.

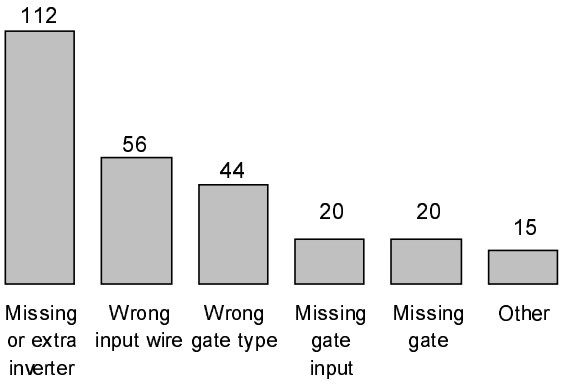


Fig. 2.3 ***Distribution of design error classes [25]***

2.2.2 Error Hypothesis

Error hypothesis is an assumption about an error(s) existing in an incorrect implementation. Error hypotheses are usually based on a concrete error model. In the case of the model described above, the error hypothesis may be any of the eight error types listed above. It could be also any combination of the errors, for example, a gate substitution and a missing inverter are supposed to be in the implementation. If the correction applied to the suspected erroneous area according to an error hypothesis fails, another hypothesis is usually chosen.

2.2.3 Ambiguity of Error Location

Since there is more than one way to synthesize a given function, it is possible that there is more than one way to model the error in an incorrect implementation, i.e., the correction can be made at different locations. In Fig. 2.4(a), for example, a correct circuit is placed that occasionally was incorrectly implemented (Fig. 2.4(b)). We can correct the implementation either by changing back one OR gate to NAND, or another way, as shown in Fig. 2.4(c). The former correction based on a single gate substitution hypothesis, while the latter suppose both a gate substitution and a missing gate errors.

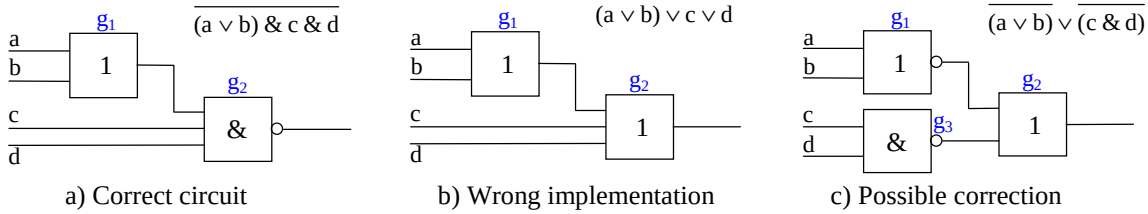


Fig. 2.4 ***Multiple rectification possibilities***

Another one example is given in [10]. In Fig. 2.5(a) a correct circuit is given. During logic synthesis process an extra OR gate was occasionally inserted (Fig. 2.5(b)). This error can be rectified using gate substitution hypothesis (Fig. 2.5(c)). Such examples for extra gate errors could be composed for any type of simple gates. The latter proofs that gate substitution hypothesis can also model simple extra gate errors, reducing the number of error types to check. More examples on error location ambiguity can be found in [13].

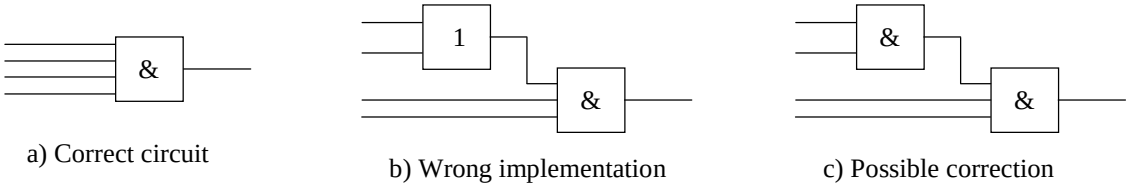


Fig. 2.5 ***Gate substitution versus extra gate***

All error hypotheses capable to rectify the circuit are considered equivalent. Error diagnosis methods are usually capable to find and correct an error within a functional equivalence class, which means that if a designer makes a simple design error, the error diagnosed is equivalent to that is made. This problem studied in more detail in [15]

2.2.4 Single and Multiple Design Errors

The simplest scenario of design error localization appears in case when an implementation has only one design error from a chosen model. As there is only one error, the area of influence of the error is smaller than in case of multiple errors. Multiple errors, especially dispersed on a wide area of a circuit, may have influence upon a large part of the circuit and also upon each other. This makes it much harder to identify the exact locations of the errors and their types. However, as it was mentioned above, in some cases, occasionally inserted multiple errors could be rectified only at one place (Fig. 2.3(a)). And backwards, a single design error may be corrected by changing functions of several other gates (Fig. 2.3(c)).

2.2.5 Combinational and Sequential Circuits

The diagnosis of combinational circuits is much easier than of sequential ones. As combinational circuits do not have memory and, therefore, do not retain information about previously applied test patterns, a choice of a test vector sequence does not affect the result of the test procedure; the diagnosis is based on an unordered set of diagnostic test patterns. On the contrary, when we deal with sequential circuits, instead of applying simple test vectors, we need to apply a specific sequence of vectors.

A sequential circuit can be seen as a succession of combinational ones over time frames. An error is repeated then 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. Thus, the error influence in a sequential circuit is growing from state to state that makes it harder to diagnose the circuit.

However every sequential circuit is divided into a combinational part and a set of flip-flops. This fact may give rise to the idea of testing a combinational part and flip-flops separately. The technique of testing a single flip-flop is trivial. It is needed to check the flip-flop truth table, that is all. The combinational part could be tested then as a conventional combinational circuit. However this strategy usually fails because the specification and the implementation may have different state encoding or even different number of states. This leads to the situation when one-to-one flip-flop correspondence between the implementation and the specification may not exist, and thus, the combinational approaches cannot be applied. A sequential error diagnosis approach is needed for this kind of situation.

3. OVERVIEW OF DESIGN ERROR DIAGNOSIS METHODS

In this chapter we will consider the previous work that is done in the field of design error diagnosis and describe several requirements for a "good" method.

3.1 WHAT MAKES A DESIGN ERROR DIAGNOSIS METHOD "GOOD"

The problem of design error diagnosis is thoroughly studied in a past few years. Many different approaches and techniques have been developed this time. However this problem still remains a hot topic for researchers because of a strong demand from industry and drawbacks every known method have. Some of them are restricted to a certain type of errors they capable to diagnose, others are restricted to a circuit size or maybe require a lot of CPU time to solve the problem. In [15] several requirements are presented satisfaction of which allow a logic diagnosis technique to put into practical use:

- applicable to multiple design errors
- applicable to both gate errors and connection errors
- indicate error location(s) exactly: what gate(s) or what line(s) are erroneous
- provide detailed solutions for error rectification

As the most frequent number of errors occasionally appeared during design process is 2 [24], a system not allowing the possibility of multiple error diagnosis has very restricted practical use. Several approaches also restrict an error model to only gate and inverter errors, however in [25] it is noted that the second of the most frequent appearing design error classes is wrong input wire error (Fig. 2.3). So, connection error diagnosis should be an essential part of a design error diagnosis system. The third requirement is not very clear because of discussed above ambiguity of error location. The sense of it should be understood as follows: the exact location of at least one of possible *corrections* must be provided. The best case is when the minimal possible change for the error correction is determined. The fourth requirement means that a diagnosis procedure should give not only information about what needs a correction but also how to correct it.

We can add the speed requirement to this list. Large circuits may require a lot of time to diagnose an error. The faster the error is found the faster re-synthesis is performed the shorter the time-to-market. If a design process has many design cycles, any delay is multiplied with each cycle. Thus the method must be scalable to a circuit size.

In other words, a perfect design error diagnosis method is a method providing the exact location of minimally possible change capable to correct the circuit. It does not rely on any error model to be able providing the correction for any possible design error. It is capable to find any combination of multiple design errors and does everything as fast as possible.

3.2 A BRIEF CLASSIFICATION

We can classify available design error diagnosis and automatic rectification methods in many ways: combinational and sequential, simulation-based and symbolic, model-based or not, structure-based/not based, error-matching and re-synthesis, single and multiple error diagnosis approaches, and so on. Several years ago, most existing approaches relied on comparatively simple model-based single-error-matching techniques. On the contrary, most recent approaches are capable to find arbitrary errors due to their model-free, multiple-error-oriented nature.

3.2.1 SIMULATION-BASED VERSUS SYMBOLIC APPROACHES

Most error diagnosis approaches are either simulation-based or symbolic. The *simulation-based* approaches first derive a number of input vectors that can differentiate the implementation and the specification. These binary or 3-valued input vectors are called *erroneous vectors* or *error detecting patterns*. By simulating each erroneous vector, the potential error region can then be trimmed down gradually. The conditions for eliminating those signals that cannot be an error source vary from one approach to another.

The *symbolic* approaches do not enumerate the erroneous vectors. They primarily rely on Ordered Binary Decision Diagram (OBDD) [21] to characterize the necessary and sufficient condition of a potential error source as a boolean formula. Based on this formulation, every potential error source can be precisely identified. In comparison, the symbolic approaches are accurate and extendible to multiple errors. However, constructing the required BDD representations may cause memory explosion when applied to large circuits. On the other hand, the simulation-based approaches, although scalable with the circuit size, are usually not accurate enough.

- simulation-based approaches [1, 3, 4, 6, 8, 10, 11, 14, 15, 16, 17, 18]
- symbolic approaches [2, 5, 12, 13, 19]
- mixed approaches [7, 9]

3.2.2 RE-SYNTHESIS VERSUS ERROR-MATCHING RECTIFICATION APPROACHES

Rectification and diagnosis are slightly different notions. Diagnosis usually finds an error and rectification corrects it. Rectification methods mostly can be divided into three categories:

- re-synthesis based approaches [2, 3, 4, 5, 7, 9, 12, 19]
- error-matching based approaches
  - single error approaches [6, 8, 10, 11, 13, 14, 16]
  - multiple error approaches [1, 15, 18]

Error-matching based approaches use an error model consisting of most commonly occurred types of errors. After error diagnosis, the implementation is rectified by matching the error with an error type in the model. This method is relatively restricted because, as it was mentioned above, it may fail when a real error in implementation can not be matched to any error in the model. Also, it is hard to be generalized for a circuit with multiple errors.

On the other hand, *re-synthesis* based approach is more general. These approaches rely on the symbolic error diagnosis techniques to find an internal signal in the implementation that satisfy the *single fix condition*, i.e., the condition of fixing entire implementation by changing the function of an internal signal. Once such a signal is found, a new function is realized to replace the old function of this signal to fix the error. In [12], a primary output partitioning algorithm was proposed to further enhance the capability of this approach. This enhancement makes it more suitable for solving engineering change problem. Theoretically this process will always succeed. But in the worst case, it may completely re-synthesize every primary output function. Another drawback of this approach is that it cannot handle larger designs because of using OBDD.

3.2.3 Structure-Based Approaches

Structural approaches [5, 7, 9, 12] mostly rely on finding structural similarities between a specification and an implementation. This has been shown to be effective in reducing the verification complexity of large combinational circuits [9,23]. These approaches require a structural description for the specification or reference circuit. First, equivalent signal pairs in two circuits are identified. The more the similarity degree between the two circuits, the smaller the part of implementation remained for further diagnosis. The technique applied then, to search error(s) in this remained area, may be various. In [24], for example, a heuristic called *back-substitution* is employed in hopes of fixing the error incrementally. In [5] a symbolic re-synthesis approach is used. The major drawback of structural approaches is that the success of the whole procedure is highly depended on existence of structural similarity between two circuits. However, these approaches could be suitable for large circuits.

3.2.4 Distinct Features of Sequential Circuit Oriented Approaches

Due to special features of sequential circuits, combinational approaches could not be directly applied to them. However, with certain modifications done, some simulation-based combinational circuit oriented approaches can be used for sequential circuits too [11, 3]. In [11] a modified combinational circuit approach is presented that is based on the method presented in [10]. The authors introduce a concept of Possible Next States that 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. A drawback of this approach is that they consider registers and flip/flops as basic components of the circuit description and the diagnosis is only concerns the combinational part of the circuit. This method also relies on a restricted error model and could not rectify multiple errors.

Some sequential approaches also use symbolic techniques [7]. In these approaches, the circuits are regarded as finite state machines (FSM) and characterized by a transition relation and a set of output functions using BDD's. A product machine is constructed and its state space is traversed. Most of them assume a reset state, and employ a

breadth-first traversal algorithm to compute the set of reachable states. The equivalence of these two machines can be proved by checking the tautology of every primary output of the product machine. Due to the memory explosion problem, these approaches can easily fail for large designs.

In [7] a hybrid approach is presented that combines symbolic BDD techniques and exploiting structural similarity between two circuits. The key idea in these algorithms is that, instead of directly examining the functional equivalence of the primary outputs, equivalent flip-flop pairs and equivalent internal signal pairs are first identified. This process proceeds forward from the primary inputs towards the primary outputs. Once an internal equivalent signal pair is identified, it is merged to speed up the subsequent verification process. This approach can show good results if the two circuits are structurally similar. This approach can be also less vulnerable to memory explosion problem.

4. A NEW STUCK-AT FAULT MODEL ORIENTED METHOD

4.1 INTRODUCTION

In [26] and [27] general ideas and basic theoretical concepts for a new hierarchical design error diagnosis method are presented. The method is based on the stuck-at fault model, where all the analysis and reasoning is carried out in terms of stuck-at faults and only in the end, the result of diagnosis will be mapped into the design error area. Such a treatment allows exploiting traditional ATPGs to serve the problem of design error diagnosis.

Another distinct feature of the method is that it uses a new model of structurally synthesized BDDs (SSBDD) [32]. In contrast to BDD or OBDD circuit representations the complexity of SSBDD representation model generation does not grow exponentially with the circuit size but only linearly [32]. In addition SSBDD representation preserves structural information about circuit allowing developing fault diagnosis procedures that are more efficient for increasing the speed in error detection and localization that gate-level ones. On SSBDDs, a primary set of suspected faulty signal paths are calculated. Based on these paths, a list of suspected erroneous gates is generated, which subsequently will be reduced to the minimum by using the information obtained from the test experiment.

Our approach combines both verification and error localization techniques together. The information gathered on the verification stage is used then during localization reducing the time needed for the whole process.

Due to the facts enumerated above, our method provides one of the fastest erroneous area localization processes among other known methods.

Another advantage of the method is that it is not structure based. In other words, the specification can be represented on any level of abstraction; it can be given as a truth table, as a BDD, as another gate-level circuit, or in form of Boolean formula. We use SSBDDs only to represent the erroneous implementation. They can be generated directly from gate-level netlists. Thus, the success of our method does not depend on any structural similarity between the specification and the implementation.

Although, our method is based on a restricted to gate-substitution and inverter errors simple error model of Abadir et al. [20], it was shown above that simple extra gate errors could be rectified using only gate substitution error model. So, missing gate error and connection types of errors are remained out of the scope of this work. They, as well as multiple errors, are the subject of our future work.

The work described in this chapter represents the implementation of the method. Refined fault calculation procedure is developed and described in detail. The presented material has been also accepted for publication [28, 29].

4.2 DEFINITIONS AND TERMINOLOGY

Consider a circuit specification, and its implementation. The way of representation for the specification is not significant. Only relationship between input patterns and output responses in specification is important. However, without loss of generality, let the specification and the implementation are given at the Boolean level. The specification output is given by a set of variables $W = \{w_1 , w_2 , ... , w_m\}$, and the implementation output is given by a set of variables $Y = \{y_1 , y_2 , ... , y_m\}$, where m is the number of outputs. Let $X = \{x_1, x_2, ... , x_n\}$ be a set of input variables. The implementation is a gate network and Z is a set of internal variables used for connection of gates. Let S be the set of variables in the implementation $S = Y \cup Z \cup X$. The gates implement simple Boolean functions AND, NAND, OR, NOR, XOR, XNOR and NOT. An additional gate type FAN is added (one input, two or more outputs) to model fanout points (Fig. 4.1). It is not used in the diagnostic procedure but needed in several definitions below.

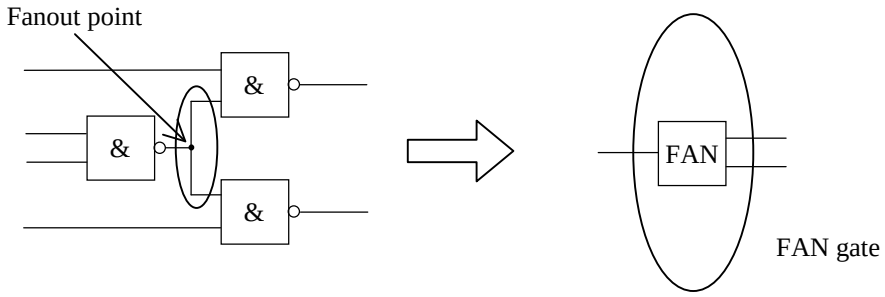


Fig. 4.1 ***A FAN gate***

We use two different levels for representing the implementation: gate and macro-level representations.

Let X^F and Z^F be the subsets of inputs and internal variables that fanout (they are input to a FAN gate). Let Z^{FG} be the subset of internal variables that are output of a FAN gate. At the gate level, the network is described by a set $NG = \{g_k\}$ of gate functions $s_k = g_k (s_k^1, s_k^2, ... ,s_k^h)$ where $s_k \in Y \cup Z$, and $s_k^j \in (Z - Z^F) \cup (X - X^F)$. At the macro-level, the network is given by a set $NF=\{f_k\}$ of macro functions $s_k = f_k(s_k^1, s_k^2, ..., s_k^p)$ in an equivalent parenthesis form (EPF) [31], where $s_k \in Y \cup Z^F$, and $S_k = \{s_k^1, s_k^2, ..., s_k^p\} \subseteq Z^{FG} \cup (X - X^F)$ be its set of inputs. In other words, a macro is a tree-structured subcircuit with no fanout points inside. It has several inputs and only one output (Fig. 4.2).

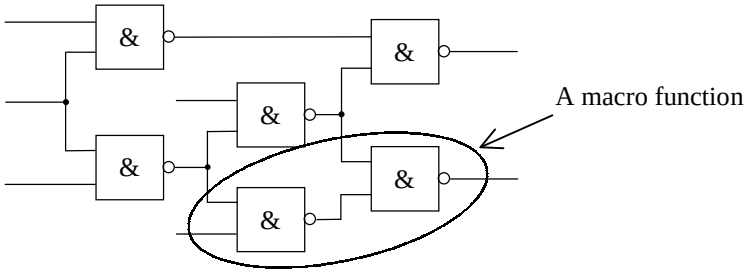


Fig. 4.2 ***An example of a macro function***

The following design error types are considered throughout the paper in relation to gates $g_k \in NG$.

Definition 1. *Gate replacement error*. It denotes a design error which can be corrected by replacing the gate g_i in NG with another gate g_j , by $g_i \rightarrow g_j$. The "$\rightarrow$" sign refers to "is replaced by".

Definition 2. *Extra/missing inverter error*. It denotes a design error which can be corrected by removing/inserting an inverter at some input $s \in X$, or at some fanout branch $s \in Z^{FG}$: $s \rightarrow NOT(s)$.

Definition 3. *Single error hypothesis*. Our design error diagnosis methodology is based on a single error hypothesis where it is assumed that in the circuit a single error from the following error types can exist: 1) an extra/missing inverter, 2) a random gate replacement between AND, OR, NAND, NOR, XOR and XNOR gates.

From these definitions it should be clear that only inverters occasionally inserted/omitted at a gate input or at a primary input of a circuit are treated as extra/missing inverters. The inverters inserted/omitted at a gate output are covered by gate replacement error type (Fig. 4.3).

Fig. 4.3 ***Extra inverter and gate replacement errors***

Definition 4. *Test patterns*. For a circuit with n inputs, a test pattern T'_i is a n-bit vector which may be binary B^n or ternary T^n, where B = {0,1} - the Boolean domain, T = {0,1,U} - the ternary domain, where U - is a don't care. Denote the set of all test patterns applied to a circuit during one test experiment as $T = \{T_1, T_2, ... T_t\}$, where t is a number of test patterns.

Definition 5. *Stuck-at fault set*. Let F be the set of stuck-at faults s/1 and s/0, where $s \in Z \cup X$.

Definition 6. *Detectable stuck-at faults*. A test pattern T_i detects a stuck-at-e fault s/e, $e \in \{0,1\}$ at the output y_j, if when applying the test pattern T_i to the implementation and the specification, the result $y_j(T_i) \neq w_j(T_i)$ is observed. Let us call s/e a detectable by a test pattern T_i at the y_j output stuck-at fault. Denote the detection information as $\varepsilon(T_i, y_j)$, where

$$\varepsilon(T_i, y_j) = \begin{cases} 0, \text{ if the test } T_i \text{ passed at } y_j \\ 1, \text{ if the test } T_i \text{ detected an error at } y_j \end{cases}$$

and as $\delta(T_i, y_j)$, where

$$\delta(T_i, y_j) = \begin{cases} 0, \text{ stuck - at 0 (s/0) is } \textit{detectable} \text{ by } T_i \text{ at } y_j \\ 1, \text{ stuck - at 1 (s/1) is } \textit{detectable} \text{ by } T_i \text{ at } y_j \\ X, \text{ no fault is } \textit{detectable} \text{ by } T_i \text{ at } y_j \end{cases}$$

Denote a set of stuck-at faults detectable by a test pattern T_i at an output y_j as $F(T_i, y_j)$, then let

$$F(T_i) = \bigcup_{y_j \in Y} F(T_i, y_j)$$

be a set of faults detectable by a test pattern T_i, and

$$F(T) = \bigcup_{T_i \in T} F(T_i)$$

be a set of faults detectable by all the test patterns $T_i \in T$. A test T is complete iff F(T)=F. All the following is based on assumption of complete test.

Definition 7. *Set of failing test patterns*. Let $E = \{ T_i \mid \exists j: T_i \rightarrow y_j(T_i) \neq w_j(T_i) \} \subseteq T$ be a set of failing test patterns.

Definition 8. *Suspected faults*. During the process of localization the proposed algorithm produces a set of suspected faults firstly at the macro level and then at the gate level. At the macro level it is a subset $F_M^S \subseteq F(T)$ of stuck-at faults that are supposed to be presented in the circuit on account of the set E. At the gate level it is a subset F_G^S of gates that can be erroneous on account of the set of suspected at the macro level stuck-at faults. We denote a suspected fault in a single node or gate output as σ, where

$$\sigma = \begin{cases} X, \text{ if no fault is suspected} \\ 0, \text{ if stuck - at 0 (s/0) is suspected} \\ 1, \text{ if stuck - at 1 (s/1) is suspected} \\ D, \text{ if both types (s/0, s/1) are suspected} \end{cases}$$

4.3 MAPPING STUCK-AT FAULTS INTO DESIGN ERROR MODEL

The stuck-at fault model does not have in this paper a physical meaning. It only used to produce, as in the case of traditional testing, a diagnosis in terms of stuck-at faults, which is then mapped into design error domain. The method of mapping follows from the proof of the following theorem given in [26] and [27].

Theorem 1. To detect a design error in the implementation at an arbitrary gate g_k where $s_k = g_k(s_1, s_2, ..., s_h)$, it is sufficient to apply a pair of test patterns which detect the stuck-at faults $s_i/1$ and $s_i/0$ at one of the gate inputs s_i, i = 1,2, ... h.

From the proof the following set of corollaries was driven which describes the mapping from a stuck-at fault set to the design error model domain:

- localizing both the s/1 and s/0 faults on two or more gate inputs refers to the missing/extra inverter at the gate output, i.e. to the replacement errors: AND ↔ NAND and OR ↔ NOR;

- localizing s/1 faults at one or more gate inputs refers to the replacement errors: AND $\rightarrow$ OR, OR $\rightarrow$ NAND, NAND $\rightarrow$ NOR, and NOR $\rightarrow$ AND;
- localizing s/0 faults at one or more gate inputs refers to the replacement errors: AND $\rightarrow$ NOR, OR $\rightarrow$ AND, NAND $\rightarrow$ OR, and NOR $\rightarrow$ NAND;
- localizing both the s/1 and s/0 faults at one of the gate inputs s_i refers to the error $s_i \rightarrow NOT(s_i)$ at this input;
- localizing both the s-1 and s-0 faults at more than one branch of a primary input $s_i \in X^F$ refers to the error $s_i \rightarrow NOT(s_i)$ at this input.

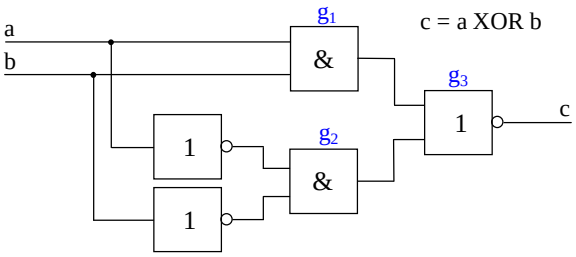
The mapping is summarized in Table 4.1. In the first column suspected erroneous gates in implementation are given. Next several columns refer to stuck-at faults detected at the gate inputs s_1, s_2, …, s_k. The last column represents the correction needed to rectify the circuit.

Gate	Stuck-at faults							Correction
	s_1		s_2		…	s_h		
AND	0	1	0	1	…	0	1	NAND
		1		1	…		1	OR
	0		0		…	0		NOR
	0	1			…			NOT(x_1)
			0	1	…			NOT(x_2)
					…	0	1	NOT(x_h)
OR	0	1	0	1	…	0	1	NOR
	0		0		…	0		AND
		1		1	…		1	NAND
	0	1			…			NOT(x_1)
			0	1	…			NOT(x_2)
					…	0	1	NOT(x_h)
NAND	0	1	0	1	…	0	1	AND
	0		0		…	0		OR
		1		1	…		1	NOR
	0	1			…			NOT(x_1)
			0	1	…			NOT(x_2)
					…	0	1	NOT(x_h)
NOR	0	1	0	1	…	0	1	OR
		1		1	…		1	AND
	0		0		…	0		NAND
	0	1			…			NOT(x_1)
			0	1	…			NOT(x_2)
					…	0	1	NOT(x_h)

TABLE 4.1 ***Mapping stuck-at faults into design error domain***

However it should be noted that gate replacement errors for XOR and XNOR gates are not presented here. This is due to the fact that a network of simpler gates can represent them. In [10], an example is given how a XOR gate replacement error can be rectified

using enumerated above gate replacement types (Fig 4.4). Thus, our model also covers the XOR/XNOR gate replacement by other gates.



Replace g_3 by OR	to get c = a XNOR b
Replace g_1 by NOR	to get c = a OR b
Replace g_1 by NAND	to get c = a AND b
Replace g_2 by NOR	to get c = a NAND b
Replace g_2 by NAND	to get c = a NOR b

Fig. 4.4 ***Representation of a XOR gate [10]***

4.4 MACROS, PATHS, AND STUCK-AT FAULT MODELING

The following fault modeling method was developed for macro-level test generation based on using structurally synthesized BDDs (SSBDD) as the model for tree-like subcircuits or macros [30, 31, 32]. Every macro is represented by its own SSBDD.

Definition 9. *Signal paths*. We denote $L(s_k^j)$ the set of variables on a path from the input of the macro $s_k^j \in S_k$ to its output s_k.

As macros are trees, there exists a one-to-one correspondence between inputs $s_k^j \in S_k$ and the gate-level signal paths $L(s_k^j)$ in the macro. The literal s_k^j in the EPF is an inverted (not inverted) variable if the number of inverters on the path from s_k^j to s_k is odd (even).

Definition 10. *SSBDD*. A SSBDD is a directed noncyclic graph G_k with a single root node and a set of nonterminal nodes M_k labeled by (inverted/not inverted) Boolean variables (arguments of the function or inputs of the tree-like subcircuit). Every node has exactly two successor-nodes, whereas terminal nodes are labeled by constants 0 or 1. The set M_k represents a macro f_k so that one-to-one correspondence exists between the nodes $m \in M_k$ and signal paths L(s) where $s \in S_k$. Let s(m) denote the literal at the node m in the graph G_k.

We also use another definition of signal paths throughout the paper: $L(m_k) = \{g_k^1, g_k^2, g_k^3, \ldots, g_k^r\}$ is a corresponding to the node m_k path where g_k^j is a gate on the path and there is a relation of order $R(g_k^a, g_k^b)$, $a,b \in [1,r]$ where from a<b results that g_k^b stands closer to the output of the macro than g_k^a.

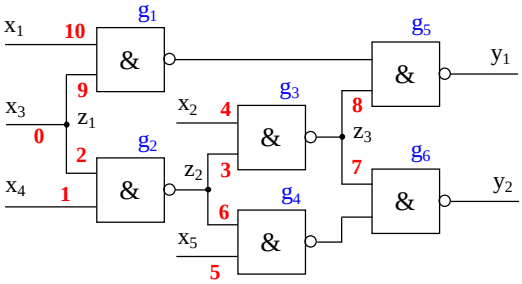


Fig. 4.5 ***Combinational circuit***

Example 1.Consider the combinational circuit from Fig. 4.5. It is a small circuit c17 of the ISCAS'85 benchmarks [38]. Then, by previous definitions, X={x_1, x_2, x_3, x_4, x_5} is the set of input variables, Y={y_1, y_2} is the set of outputs, and NG={g_1, g_2, g_3, g_4, g_5, g_6} is the set of gates. The procedure of formal synthesis of SSBDDs from gate-level networks based on a graph superposition procedure is considered in [30,32]. In Fig. 4.6 five macros needed for this circuit representation are shown. The nodes of these SSBDDs are denoted in Fig. 4.5 by numbers from 0 to 10 for nodes from m_0 to m_{10} correspondingly and internal variables are denoted as z_1, z_2, and z_3.

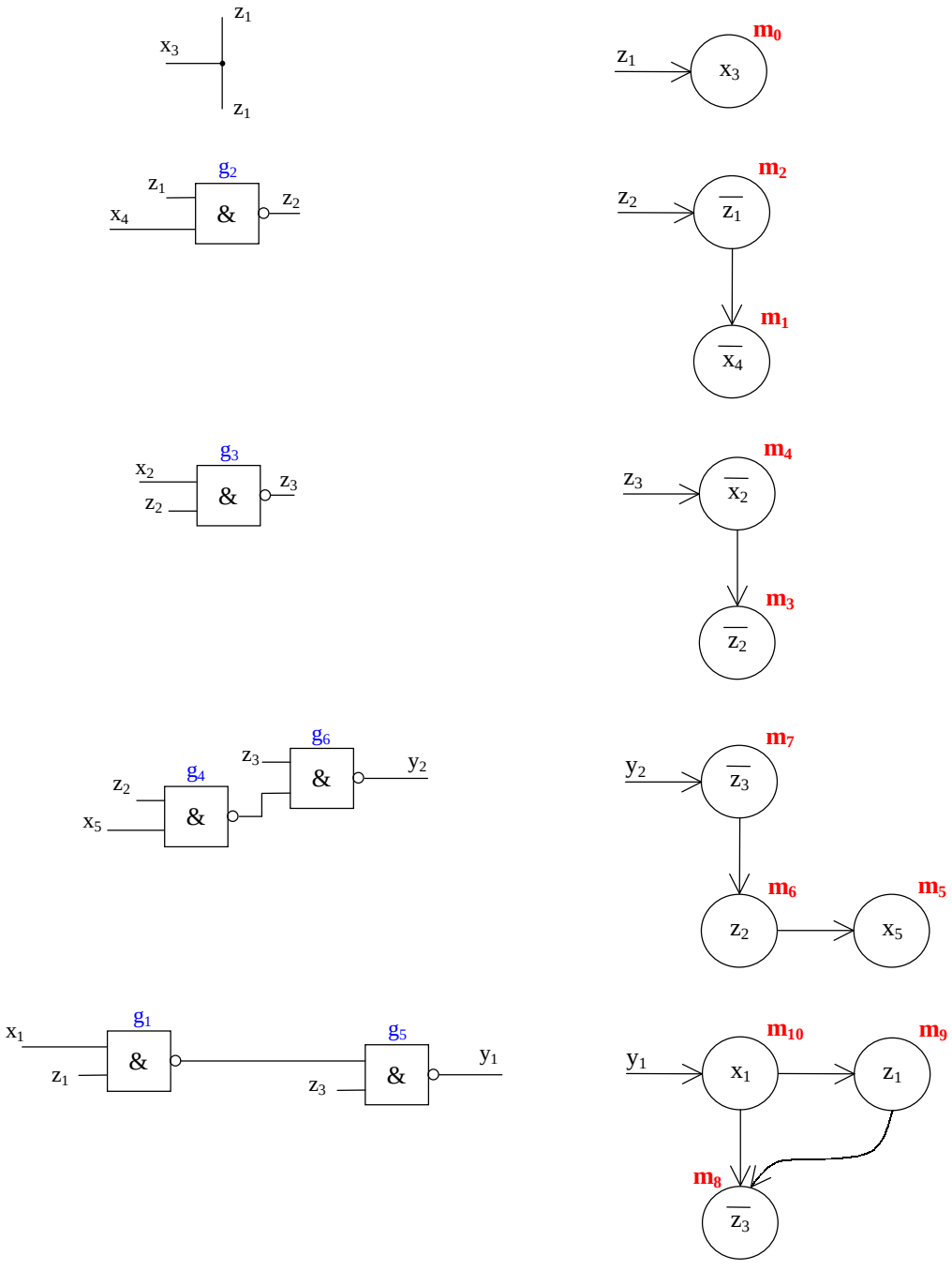


Fig. 4.6 ***Tree-like subcircuits and their SSBDDs***

The following table shows how the nodes m_k in SSBDD macros are related to gate-level paths $L(m_k)$ in the circuit.

m_k	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}
$L(m_k)$	$\varnothing$	g_2	g_2	g_3	g_3	g_4,g_6	g_4,g_6	g_6	g_5	g_1,g_5	g_1,g_5

TABLE 4.2 ***SSBDD nodes and corresponding gate paths***

Definition 11. *Faults on signal paths, fault class*. Let $F(s_k^j/e)$ where $e \in \{0,1\}$ be a set of all faults (a fault class) in the gate-level signal path from the input s_k^j of the macro S_k to its output s_k.

A fault s_k^j/e can be regarded as the representative of a fault class $F(s_k^j/e)$, since to test all the faults $F(s_k^j/e)$ it is enough to test only the fault s_k^j/e.

If a fault $\delta(s(m_k))$ is detected or a fault $\sigma(s(m_k))$ is suspected in a node m_k in SSBDD then similarly fault classes $F(\delta(s(m_k)))$ and $F(\sigma(s(m_k)))$ results at the gate level.

4.5 FAULT TABLES AND VECTOR REPRESENTATION OF FAULTS

Let us represent the set of detectable at the macro level faults F(T) as a *fault table* (Table 3) where row i corresponds to a test T_i and column k corresponds to a variable $s(m_k)$ in the node m_k and $\delta_{ik} \in \{0,1,X\}$ shows what kind of fault is detectable at m_k by a test pattern T_i.

T_i	$s(m_k)$					E
	$s(m_1)$	$s(m_2)$	$s(m_3)$	...	$s(m_q)$	
T_1	δ_{11}	δ_{12}	δ_{13}	...	δ_{1q}	ε_1
T_2	δ_{21}	δ_{22}	δ_{23}	...	δ_{2q}	ε_2
T_3	δ_{31}	δ_{32}	δ_{33}	...	δ_{3q}	ε_3
T_4	δ_{41}	δ_{42}	δ_{43}	...	δ_{4q}	ε_4
⋮	⋮	⋮	⋮		⋮	⋮
T_t	δ_{t1}	δ_{t2}	δ_{t3}	...	δ_{tq}	ε_t

TABLE 4.3 ***A fault table***

Each row of this table $(\delta_{i1}, \delta_{i2}, \delta_{i3}, \ldots, \delta_{iq})$ represents $F(T_i)$ – the set of faults detectable by a test pattern T_i. The last column (E) shows the results of a test experiment, whether a test passed or detected an error.

Since we need also to know sets of faults detectable by a particular test pattern at each primary output y_j we have to construct such fault tables for each output separately.

In this case (Fig. 4.7), each row $(\delta(T_i, y_j, s(m_1)), \delta(T_i, y_j, s(m_2)), \delta(T_i, y_j, s(m_3)), \ldots \delta(T_i, y_j, s(m_q)))$ of a table j represents $F(T_i, y_j)$ – the set of faults detectable by a test pattern T_i at a primary output y_j.

The last column in each table shows whether a test pattern detected an error at a particular output, or not.

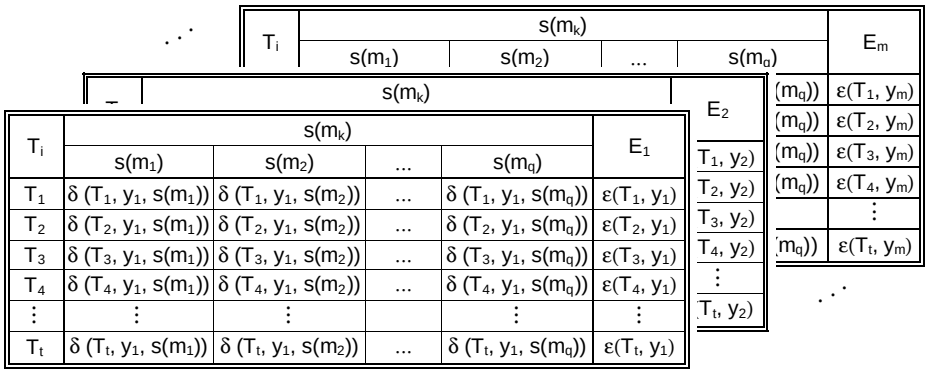


Fig.4.7 ***Tables of faults detectable at each output***

It is not hard to notice that each row of the Table 3 as well as rows of tables in Fig. 4.7 can be represented in a vector form.

Definition 12. *Vector of detectable faults*. Let us denote a vector of faults detectable at the macro level by a test pattern T_i as $V_M^d(T_i) = (\delta_{i1}, \delta_{i2}, \delta_{i3}, \ldots, \delta_{iq})$ and let $V_M^d(T_i, y_j) = (\delta(T_i, y_j, s(m_1)), \delta(T_i, y_j, s(m_2)), \delta(T_i, y_j, s(m_3)), \ldots \delta(T_i, y_j, s(m_q)))$ be a vector of faults detectable at the macro level at a particular output y_j by a test pattern T_i.

Definition 13. *Vector of suspected faults*. Denote a vector of faults suspected on account of a set of failing test patterns E at the macro level as $V_M^s = (\sigma(s(m_1)), \sigma(s(m_2)), \sigma(s(m_3)), \ldots, \sigma(s(m_q)))$.

Let us define some operations with these vectors. We will need four types of operations: intersection ($\cap$), union ($\cup$), subtraction ($-$) and inversion ($\overline{\sigma}$). All these operations are performed element-wise when applied to a pair of vectors.

$\sigma_1 \cap \sigma_2$

	X	0	1	D
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
D	X	0	1	D

$\sigma_1 \cup \sigma_2$

	X	0	1	D
X	X	0	1	D
0	0	0	D	D
1	1	D	1	D
D	D	D	D	D

$\sigma_1 - \sigma_2$

σ_1 \ σ_2	X	0	1	D
X	X	X	X	X
0	0	X	0	X
1	1	1	X	X
D	D	1	0	X

$\overline{\sigma}$

σ	$\overline{\sigma}$
X	X
0	1
1	0
D	D

Since the domain of δ is a subset of the domain of σ all these operations can be performed also when operands are delta, or sigma and delta in all possible combinations. Note that the definition of the inversion is maybe a little outlandish, but it becomes clear if we say that we use it propagating stuck-at fault errors along the signal

path. If a gate on the path is inverting (NOT, NOR, NAND) then the suspected stuck-at fault on the input of the gate is inverted at the output. If there is no suspected fault (X) on the input, there is nothing to invert and there is also no suspected fault at the output. Similarly, if there are both types of faults (D) are suspected at the input, the inversion of each fault separately gives us again both types of faults (D) suspected at the output.

4.6 DESIGN ERROR DIAGNOSIS ALGORITHM

The proposed design error diagnosis algorithm is implemented and works at two different levels of abstraction: macro (SSBDD) and gate level. The algorithm itself is very simple. This is due to underlying SSBDD circuit representation and techniques based on it. Very simple stuck-at fault model is used during almost the whole diagnosis process instead of design error model helping to avoid sophisticated techniques for error localization.

Firstly, during the stage of verification through fault simulation, the fault tables are constructed (Fig. 4.7) for each output. Using these tables and the following theorem we produce macro-level diagnosis.

Theorem 1. From a set E of failing test patterns the following vector of suspected at the macro level faults results:

$$V_M^s = \bigcup_{T_i \in E}\left[\bigcap_{y_j:\varepsilon(T_i,y_j)=1} V_M^d(T_i, y_j)\right] - \bigcup_{T_i \in E}\left[\bigcup_{y_j:\varepsilon(T_i,y_j)=0} V_M^d(T_i, y_j)\right] - \bigcup_{T_i \in F-E} V_M^d(T_i)$$

Proof. The proof results from the single error hypothesis. Suppose an error has been detected at more than one output $y_j \subseteq Y$ by a single test pattern T_i but only a single fault can be the cause of that. Therefore, only the intersection of vectors of detectable faults $V_M^d(T_i, y_j)$ at erroneous outputs $y_j:\varepsilon(T_i, y_j)=1$ can contain the existing fault. From all the failing test patterns $T_i \in E$, a vector

$$V' = \bigcup_{T_i \in E}\left[\bigcap_{y_j:\varepsilon(T_i,y_j)=1} V_M^d(T_i, y_j)\right]$$

of suspected faults results. On the other hand, if some of these suspected faults from this union of intersections have a direct impact to the outputs where no error has been detected, they cannot be anymore suspected and the union

$$V'' = \bigcup_{T_i \in E}\left[\bigcup_{y_j:\varepsilon(T_i,y_j)=0} V_M^d(T_i, y_j)\right]$$

should be subtracted from V'. Similarly all the faults

$$V''' = \bigcup_{T_i \in T-E} V_M^d(T_i)$$

not detected by test patterns $T_i \in$ T-E at all, cannot be suspected and we also subtract them from V'. ■

The explanation of the formula is given in the proof. Using this formula, we compute the set of suspected SSBDDs' nodes in the vector form V_M^s where an element $\sigma(s(m_k))$ of the vector shows what kind of fault is suspected in the node m_k.

Algoritm 1. *Macro-level diagnosis*.

1. Calculate V' as a vector of suspected faults.

2. Calculate V'' as a first vector of fault free nodes.

3. Calculate V''' as another vector of fault free nodes.

4. Calculate V_M^s = V' - V'' - V''' as the updated vector of suspected faults.

At this point macro-level diagnosis stops and localization of erroneous gate(s) begins. Since every node at the macro level correspond to a certain path (a set of gates) $L(m_k)=\{g_k^1, g_k^2, g_k^3, \dots, g_k^r\}$ on the gate level, we have to propagate the suspected in the node fault through all the gates in the path inverting it each time an inverting gate (NOT, NOR, or NAND) is occurred.

Algorithm 2.

Let $s(m_k)$ is a variable in a node m_k, $L(m_k) = \{g_k^1, g_k^2, g_k^3, \dots, g_k^r\}$ is a path corresponding to the node m_k, $\sigma(s(m_k))$ is a fault suspected in the SSBDD node m_k, and $\sigma(g_k^j)$ is a fault suspected at the output of a gate g_k^j, then we propagate a fault through the path as follows:

1. $$\sigma(g_k^1) = \begin{cases} \sigma(s(m_k)), \text{ if the gate } g_k^1 \text{ is either OR or AND} \\ \overline{\sigma}(s(m_k)), \text{ if the gate } g_k^1 \text{ is either NOT, NOR or NAND} \end{cases}$$

2. $$\sigma(g_k^j) = \begin{cases} \sigma(g_k^{j-1}), \text{ if the gate } g_k^j \text{ is either OR or AND} \\ \overline{\sigma}(g_k^{j-1}), \text{ if the gate } g_k^j \text{ is either NOT, NOR or NAND} \end{cases}$$

3. Repeat step 2. while $2 \le j \le r$.

For improving the resolution of diagnosis, the following theorem can be used.

Theorem 2. If two test patterns T_1 and T_2, which detect the both stuck-at faults s(m)/1 and s(m)/0 at the node m in G_k, will not fail then all the gates along the path L(m) in the gate-implementation are free from design errors.

The proof of the theorem is given in [26].

The next theorem is used for the suspected set of erroneous gates calculation.

Theorem 3. From a vector V_M^s of suspected at the macro level faults the following set of suspected erroneous gates results

$$F_G^s = \bigcup_{\sigma(s(m_k))\neq X} L(m_k) - \bigcup_{\sigma(s(m_k))=X} L(m_k)$$

Proof. Since $F(\sigma(s(m_k)))$ is a fault class corresponding to the suspected in a node m_k fault $\sigma(s(m_k))$, then from the suspicion of a fault $\sigma(s(m_k))$ at the macro level all the faults belonging to the path $L(s(m_k))$ at the gate level become suspected. When we take a union of all such paths we are assured that the fault is contained in it. On the other hand, if there is no fault suspected in the node m_k and from Theorem 1 we know that in this case no fault was detected at this node and from Theorem 2 the whole path $L(m_k)$ cannot be any more suspected and the union of such paths, where $\sigma(s(m_k))=X$, must be subtracted from the union of suspected paths, where $\sigma(s(m_k))\neq X$. ■

From this theorem only the set of suspected erroneous gates results. To find the set of faults, or in other words to find types of faults at outputs of suspected erroneous gates we use Algorithm 2.

From Algorithm 2 and Theorem 3 the following algorithm results:

Algoritm 3. *Gate-level diagnosis.*

1. Calculate $F' = \bigcup_{\sigma(s(m_k))\neq X} L(m_k)$ as a set of suspected erroneous gates.

2. Calculate $F'' = \bigcup_{\sigma(s(m_k))=X} L(m_k)$ as a set of correct gates.

3. Calculate F_G^s = F' - F" as the updated set of suspected gates.

4. Use Algorithm 2 for fault type determination.

Example 2.

Consider a test with 5 patterns that is applied to the inputs of the circuit in Fig. 4.5. Suppose now, the test patterns T_1 and T_5 fail at both outputs which results in E={T_1,T_5} and $\varepsilon(T_1, y_1)= \varepsilon(T_1, y_2)= \varepsilon(T_5, y_1)= \varepsilon(T_5, y_2)=1$ while other $\varepsilon(T_i, y_j)=0$. The vectors of detectable faults $V_M^d(T_i, y_j)$ and error detection information for such case are presented in the following fault tables for outputs y_1 and y_2 correspondingly.

T_i	$s(m_k)$											E_1
	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	
T_1	0	0	0	1	X	X	X	X	0	X	1	1
T_2	1	X	X	X	1	X	X	X	0	1	X	0
T_3	0	X	X	X	X	X	X	X	X	0	0	0
T_4	1	X	1	0	0	X	X	X	1	X	X	0
T_5	X	X	X	0	0	X	X	X	1	X	X	1

TABLE 4.4 ***Fault table for y_1***

T_i	$s(m_k)$											E_2
	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	
T_1	0	0	0	1	X	X	1	0	X	X	X	1
T_2	X	X	X	X	1	1	X	0	X	X	X	0
T_3	X	1	X	X	X	0	0	X	X	X	X	0
T_4	1	X	1	X	X	X	X	X	X	X	X	0
T_5	X	X	X	0	0	X	X	1	X	X	X	1

TABLE 4.5 ***Fault table for y_2***

From that, according to Algorithm 1, we create a vector of suspected representative faults at the macro level (macro-level diagnosis). First, compute suspected fault intersections by outputs for each faulty vector:

$$\bigcap_{y_j:\varepsilon(T_1,y_j)=1} V_M^d(T_1, y_j) = (0,0,0,1,X,X,X,X,X,X)$$

$$\bigcap_{y_j:\varepsilon(T_5,y_j)=1} V_M^d(T_5, y_j) = (X,X,X,0,0,X,X,X,X,X,X)$$

Then (step 1) find the vector of suspected faults as a union of the two vectors

$$V' = (0,0,0,D,0,X,X,X,X,X,X)$$

The first vector of fault free nodes V" (step 2) can not be calculated because all the tests $T_i \in E$ fail at all outputs. However, the second one V"' (step 3) can be successfully calculated as a union of all vectors corresponding to test patterns that has not detected an error.

$$V''' = (D,1,1,0,D,D,0,0,D,D,0)$$

After that the final step of macro-level diagnosis follows:

$$V_M^s = V' - V''' = (X,0,0,1,X,X,X,X,X,X,X)$$

At this step, we have got the vector of suspected faults $V_M^s = (X,0,0,1,X,X,X,X,X,X,X)$, which shows that stuck-at 0 is suspected at SSBDD nodes m_1 and m_2 and stuck-at 1 is suspected at the node m_3.

We will turn to Algorithm 3 now to produce the gate-level diagnosis. The best case is when the result of the diagnosis is exactly located erroneous gate and such combination of suspected stuck-at faults at the gate inputs that clearly defines the type of detected design error (Table 4.1). However, sometimes the final result represents just a set of suspected erroneous gates with some suspected stuck-at fault combinations at their inputs. The explanation of possible reasons for that is given in the next chapter.

At the gate-level diagnosis stage we first use Table 4.2 to find a set of gates (a path) corresponding to each suspected SSBDD node (m_1, m_2, and m_3), and then (step 1) compute

$$F' = \{g_2\} \cup \{g_2\} \cup \{g_3\} = \{g_2, g_3\}$$

as a set of suspected erroneous gates. After that, (step 2) from each fault-free SSBDD node the following correct gate set results:

$$F'' = \{\varnothing\} \cup \{g_3\} \cup \{g_4, g_6\} \cup \{g_4, g_6\} \cup \{g_6\} \cup \{g_5\} \cup \{g_1, g_5\} \cup \{g_1, g_5\} = \{g_1, g_3, g_4, g_5, g_6\}$$

Subtracting F" from F' we have the final set of suspected erroneous gates:

$$F_G^s = \{g_2, g_3\} - \{g_1, g_3, g_4, g_5, g_6\} = \{g_2\}$$

We are lucky that inputs of g_2 are, at the same time, inputs of a macro and there is no need to use Algorithm 2 for error propagation purposes.

As the stuck-at 0 faults are suspected at the nodes m_1 and m_2 that are inputs of the gate g_2, then according to Table 4.1 (the case of NAND gate) it means the design error NAND $\rightarrow$ OR. To correct the design, the NAND gate g_2 should be replaced by an OR gate.

5. EXPERIMENTS

In this chapter we are going to describe the program implementing the diagnosis techniques given in previous part. Experimental data and analysis are also presented.

5.1 PROGRAM DESCRIPTION

A set of tools was written to carry out diagnostic experiments. They are based on and closely interact with Turbo Tester CAD system [33] tool set. For example, SSBDD generation from a netlist and input test patterns creation are implemented within Turbo Tester package. However netlist parser that creates SSBDDs (Appendix C) was slightly rewritten in order to preserve also SSBDD-*node*-gate-level-*path* relationship (Table 4.2, Appendix C).

There are two main parts of the design error diagnosis tool set:

- error insertion tool
- error analysis tool

Error insertion part is implemented as a number of functions that take a correct SSBDD model and insert a specified type of error to a specified place. The error could be either a certain gate replacement or inverter insertion/removal. Error analysis tool (Appendix A) is based on the error diagnosis techniques described in previous chapter. It has three main parts:

- faulty outputs for each test pattern detection
- suspected faults for SSBDD nodes calculation
- suspected gate set determination and faults propagation

Note that we do not produce the final diagnosis in terms of design errors. The result of diagnosis is a set of suspected erroneous gates with some suspected stuck-at fault combinations at their inputs. This is due to the high probability that the exact error location has not been found. However, in case of exact error localization, the diagnostic information, that is given, can be used to refer to the mapping table (Table 4.1) and identify the design error.

Another tool has been created to carry out a number of experiments and quantify their results. An experiment refers here to the process of an arbitrary error insertion, erroneous circuit simulation, and error detection and localization. This tool allows creating a set of experiments and displaying information with several parameters. Firstly, it provides the following ways of error insertion:

- insert each possible type of error for each gate
- insert a random type of error for each gate
- insert a predefined type of error for each possible gate
- insert each possible type of error for a specified gate

- insert a random type of error for a specified gate
- insert a predefined type of error for a specified gate

Note that only one error can be inserted during one experiment. The results of the experiments can be also presented in several ways:

- to show/not to show fault tables for each primary output
- to show/not to show which vector has detected the error, which has not
- to show/not to show a set of stuck-at faults suspected in SSBDD nodes
- to show/not to show a set of suspected erroneous gates
- to show/not to show a statistics for the set of experiments

Statistics includes the following information:

- the total number of experiments
- spectrum of suspected gates or/and nodes
- minimal number of suspected gates or/and nodes
- maximal number of suspected gates or/and nodes
- average number of suspected gates or/and nodes

Spectrum of suspected gates/nodes shows a distribution of experiments along suspected area axis.

The following information is also provided for each experiment separately:

- time, used by process
- a set of suspected gates or/and nodes and suspected stuck-at faults
- number of suspected gates or/and nodes
- names of suspected gates

An example of diagnostic results for c17 ISCAS'85 [38] benchmark is given in Appendix B.

The following files are needed to carry out an experiment (Appendix C):

- design.agm – macro-level SSBDD model
- design.gate.agm – gate-level SSBDD model
- design.tst – input test patterns
- design.pat – SSBDD node to gate-level path relationship
- design.gat – names and types of gates

The word "design" in filenames stands for the name of a design. For example, for c17 circuit it is c17.agm or c17.pat and so on. A design.gate.agm file represents a gate-level SSBDD model for the same circuit. The difference between macro and gate-level SSBDD representations is that the latter is partitioned by gates, not by macros. In other words, an SSBDD is constructed for each gate separately. The both SSBDD representations are equal functionally but different structurally. This gate-level SSBDD model is needed for error insertion purposes where the whole SSBDD of a particular

gate can be replaced with another one. A design.tst file contains input test patterns generated by the tool described in [32]. Gate names and types needed, for example, when replacing a certain gate are preserved in design.gat file. Examples of all these files for c17 circuit can be found in Appendix C.

```
pitsa:/export/home2/tester/artur/C/derr>deserr
Single Gate Design Error Diagnosis
Usage: deserr [-ovngs] [options] -gate <type> <design>

        o       show info for outputs
        v       show which vector is passed, which not
        n       show suspected nodes
        g       show suspected gates
        s       show statistic
options:
   -name <gatename> make experiment only for specified gate
   -all                    make experiments for all gates

   -gate <type>            generate gate (error) of <type>
         <type>: INV,AND,OR,NAND,NOR,RANDOM,ALL
```

Fig. 5.1 ***Usage options of the tool for design error diagnosis experiments***

The only version of the design error diagnosis tool set is written on C/C++ language and works under UNIX operating system. The screen-shot with usage options of the program is given in Fig. 5.1. When program is run without any option defined, it uses default options, so

```
deserr <design>
```

is equivalent to

```
deserr –g –all –gate RANDOM <design>
```

that means that for every gate one RANDOM error will be inserted and detection information will be displayed only for gate-level (suspected SSBDD nodes will not be displayed).

The program described here will also be included into Turbo Tester CAD system and used by students for educational purposes.

5.2 Results and Analysis of Experiments

The goals of the experiments described here were twofold:

- to compare the efficiency (the speed of fault localization) of the new diagnostic approach in comparison with previous results

- to evaluate the design error *diagnostic* properties of test patterns generated by traditional gate-level ATPGs for only stuck-at fault *detecting* purposes

Diagnosis experiments were carried out on internationally recognized ISCAS'85 benchmarks. Columns 2,3,4 in Table 5.1 give information about size of benchmarks in terms of input, output and gate quantities. Note that the number of gates given in the table may be different from the real gate count because of XOR and XNOR gates representation by a number of simpler gates (Fig. 4.4).

Experiments were carried out in the following way. We took a netlist of an ISCAS'85 circuit and treated it as a wrong implementation. Then we generated SSBDD model from it and created test patterns for detecting stuck-at faults. After that, using error insertion tool, we created a "correct" specification and simulated it with the same test patterns to find the difference between output responses of the specification and the implementation. Using this information, error analysis tool produced a diagnosis. Then, we took the initial "wrong" SSBDD again and created new "correct" specification with error insertion tool. And the whole process was repeated. In other words, we had one "wrong" implementation and many "correct" specifications.

The fault coverage (column 6) of the test patterns created by the test generator described in [32] and the test generation time in seconds (column 11) are presented in Table 5.1. Experiments were carried out on the computer platform Sun SparcServer 20 (2 x Super Sparc II microprocessors, 75MHz) with Solaris 2.5.1 operating system.

Circuit Name	Number of				Fault Cove-rage, %	Suspected Erroneous Gates				Time, s			Time for [10], s
						Number			Av. % of Total	Test Gene-ration	Fault Analysis (average)	Total	
	Inputs	Outputs	Gates	Experi-ments		Min	Max	Av.					
1	2	3	4	5	6	7	8	9	10	11	12	13	14
c432	36	7	232	671	91,07	1*	107	8,8	3,78	0,79	0,1	0,9	17,57
c499	41	32	618	1622	99,33	1	307	76,5	12,38	1,01	1,4	2,4	111,64
c880	60	26	357	1144	100	1	33	6,2	1,73	0,19	0,5	0,7	126,79
c1355	41	32	514	1830	99,51	1	248	58,4	11,37	1,35	1,5	2,9	241,79
c1908	33	25	718	1922	99,31	1	76	11,1	1,55	0,93	1,6	2,5	341,92
c2670	233	140	997	997*	94,97	1*	161	25,3	2,53	3,55	14,1	17,7	661,91
c3540	50	22	1446	1446*	95,27	1*	86	9,9	0,69	3,08	3,7	6,8	1513,82
c5315	178	123	1994	1994*	98,69	1*	239	11,1	0,56	2,38	29,4	31,8	1814,04
c6288	32	32	2416	2416*	99,34	1*	138	8,4	0,35	2,17	2,7	4,9	1895,90
c7552	207	108	2978	2978*	95,95	1*	269	15,8	0,53	12,06	44,8	56,9	

TABLE 5.1 ***Diagnostic results for ISCAS '85 benchmark circuits***

The number of experiments carried out for each circuit are shown in column 5. For the cases marked by star (*), one random error for each gate per experiment was inserted. For other cases, all possible single gate errors were simulated and analyzed – one error per one experiment.

The efficiency in the speed of diagnosis (columns 11, 12, 13) are compared to the results of [10] (column 14). The total time of diagnosis (column 13) in this work consists of two components: test generation time (column 11) and fault diagnosis (column 12).

However, it should be noted that the diagnostic resolution of this method could not be compared with the one in [10] because the test patterns were originally not generated for diagnostic purposes but just to *detect* an error. The numbers in columns 7, 8, and 9

show, correspondingly, the minimal, maximal, and average diagnostic resolutions (numbers of suspected gates) reached by the tests.

In the cases marked by star (*) in column 7, some design errors were not detected at all. The possible reason of that can be the fact that the tests were not complete (the fault coverage was not 100%).

A very interesting result is that in 30% cases (c499, c1355, and c1908) the tests with lower than 100% stuck-at fault coverage detected all possible single gate design errors. The reason lies in the mapping mechanism explained in Table 4.1 where each design error is mapped into a subset of at least two stuck-at faults.

To reach the same high resolution of [10], additional test patterns should be generated. For this purpose, the same method of [10] can be used. Since the suspected area (column 10) for diagnostic search is reduced from 100% to from 1,55% (in the best case) to 12,38% (in the worse case), the combination of the method proposed in the present work with some other method can reach significant improvements.

Circuit Name	Level of Abstraction	Total	Suspected Erroneous Area			
			Number			Av. % of Total
			Min	Max	Av.	
c499	Nodes	601	2	365	93,0	15,47
	Gates	618	1	307	76,5	12,38
c1908	Nodes	866	1	131	17,5	2,02
	Gates	718	1	76	11,1	1,55

TABLE 5.2 ***Macro and gate-level resolutions***

In Table 5.2, the diagnostic resolutions for two benchmark circuits (with the best and worst diagnostic resolutions) are shown for both macro and gate-level representation levels. Note that the average percent of suspected nodes is a little higher than the average percent of suspected gates. This is due to additional refinement of diagnosis on the gate level.

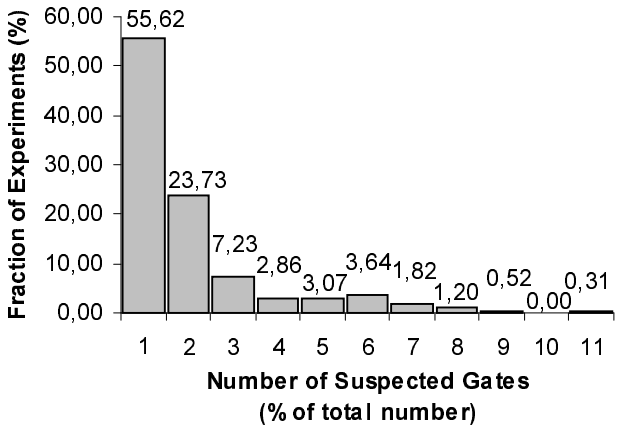


Fig. 5.2 ***Distribution of diagnostic resolutions over all possible error cases for the circuit c1908***

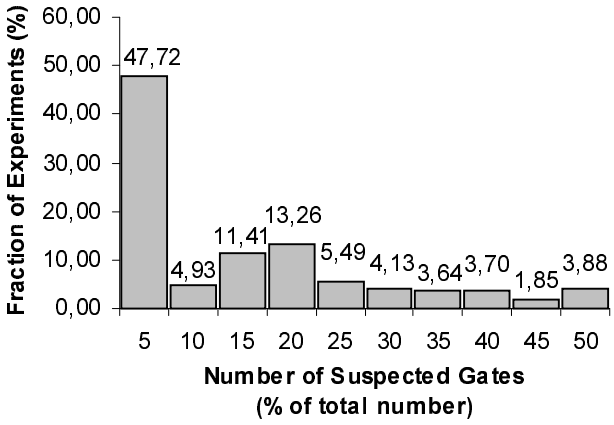


Fig. 5.2 ***Distribution of diagnostic resolutions over all possible error cases for the circuit c499***

The diagrams in Fig. 5.2 and 5.3 show, correspondingly, the distribution of experiments with different diagnostic resolutions (the best case for the circuit c1908, and the worst case for the circuit c499). Note that, in the best case, almost in 90% of experiments, the suspected area was less than 5% of the whole circuit. Even in the worst case, in almost half of experiments, the 5% suspected area was provided.

5.3 Conclusions and Future Work

Although the method, presented in this work does not own all the described in section 3.1 features of a "perfect" method, it has good chances for improvement due to its high efficiency in the speed and, therefore, the possibility of use in combination with some other rectification technique. This work, then, should be treated as a first step in the direction to a powerful design error diagnosis approach.

The utilized design error model allows diagnosing about 80% of all design error made [26]. In case the connection errors would be also added to the model, the diagnosis capability is increased to about 98%. However the use of error model may be the only major drawback of the approach.

The diagnostic resolution of the method should be improved. Additional diagnostic pattern generation could be used for this purpose. However the diagnostic pattern generation is comparatively time consuming process. Therefore maybe it is worth to investigate first, how additional randomly generated patterns could improve the resolution of the approach.

The method is also could be easily generalized to multiple design error case. Firstly, the two basic formulas the diagnosis relies upon should be slightly changed and, secondly, the test pattern generation techniques should be different.

6. CONCLUSIONS

In this thesis, general issues of automatic design error diagnosis in digital circuits have been considered, a brief analysis of existing methods is given and a new approach to this problem is presented.

The method described in this thesis has the following distinct features:

- The whole procedure takes place hierarchically at two different levels:
  - macro level, where error detection and localization of the suspected erroneous SSBDD nodes is carried out
  - gate level for erroneous gate localization and exact specification of the design error

  Exploiting the hierarchy allows to combine the efficiency of working at the higher level (for error detecting) with the accuracy (needed for error diagnosis) at the lower level.
- Working with the stuck-at fault model allows to base on a single error hypothesis, which actually means working with several error hypothesis from design error model [20] in parallel.

The future research in this field is directed to the resolution of the method improvement. For this purpose additional test pattern generation heuristics must be worked out or several existing approaches can be used. It is also necessary to pay attention to the cases of multiple errors, complex gates, and line errors.

7. REFERENCES

[1] A. Veneris, S. Venkataraman, I. N. Hajj, W. K. Fuchs, "Multiple Design Error Diagnosis and Correction in Digital VLSI Circuits," *Proceedings of the IEEE VLSI Test Symposium*, April 1999, pp. 58-63

[2] D. W. Hoffmann and T. Kropf. "AC/3 V1.00 - A Tool for Automatic Error Correction of Combinatorial Circuits," *Technical Report 5/99, University of Karlsruhe,* 1999. available at http://goethe.ira.uka.de/~hoff

[3] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and J.-Y. Lu "Fault Simulation-Based Design Error Diagnosis for Sequential Circuits," *Proc. of Design Automation Conference*, June 1998

[4] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and D.-I. Cheng, "ErrorTracer: A Fault Simulation Based Approach to Design Error Diagnosis," *Proc. of International Test Conf*, Nov. 1997, pp. 974-981

[5] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng, "Incremental Logic Rectification," *Proc. of VLSI Test Symposium*, April 1997, pp. 134-139

[6] A. Wahba, and D. Borrione, "Connection errors location and correction in combinational circuits," In *European Design and Test Conference ED&TC-97*, Paris, France, March 1997

[7] S.-Y. Huang, K.-T. Cheng and K.-C. Chen, "AQUILA: An Equivalence Verifier for Large Sequential Circuits," *Proc. of Asia-South Pacific Design Automation Conference*, Jan. 1997, pp. 455-460

[8] A. Wahba and D. Borrione, "Automatic Diagnosis may Replace Simulation for Correcting Simple Design Errors," *Proc. European Design Automation Conference EuroDAC/EuroVHDL'96*, Geneva, Switzerland, 16-20 September 1996

[9] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng, "Error Correction Based on Verification Techniques," *Proc. of Design Automation Conference*, June 1996, pp. 258-261

[10] A.M. Wahba and D. Borrione, "A Method for Automatic Design Error Location and Correction in Combinational Logic Circuits," *Journal of Electronic Testing: Theory and Applications 8*, 1996, pp. 113-127

[11] A.M. Wahba and D. Borrione, "Design Error Diagnosis in Sequential Circuits," *Proc. of Correct Hardware Designs and Verification Methods, CHARME'95, Lecture Notes in Computer Science* No. 987, Springer Verlag, Oct. 1995, pp. 171-188

[12] C.-C. Lin, K.-C. Chen, S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng, "Logic Synthesis for Engineering Change," *Proc. of Design Automation Conference*, June 1995, pp. 647-652

[13] P. Y. Chung, Y. M., Wang, and I. N., Hajj, "Logic design error diagnosis and correction," *IEEE Transactions on VLSI Systems*, vol. 2, no. 3, pp. 320-332, Sept. 1994

[14] 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

[15] M. Tomita, T.Yamamoto, F.Sumikawa, K.Hirano, "Rectification of Multiple Logic Design Errors in Multiple Output Circuits," *Proc. 31st Design Automation Conference*, 1994, pp. 212-217

[16] A. Wahba, and D. Déharbe, "Design Error Diagnosis in Logic Circuits using Ternary Test Sets", *Research Report RR-928-I, ARTEMIS-IMAG*, Grenoble, France, 1994

[17] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *Journal of the Association for Computing Machinery*, Vol. 38, No. 2, April 1991, pp. 299-328

[18] K.A. Tamura, "Locating Functional Errors in Logic Circuits," *Proc. 26th Design Automation Conference*, June 1989, pp. 185-191

[19] J.C. Madre, O.Coudert, J.P.Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM," *Proc. ICCAD'89*, 1989, pp. 30-33

[20] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Trans. on Computer*, Jan. 1988, pp. 138-148

[21] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, vol. C-35, Aug. 1986, pp. 667-691

[22] D. Brand, A. Drumm, S. Kundu and P. Narrain, "Incremental Synthesis," *Proceedings of the International Conference on Computer Aided Design*, 1994, pp. 14-18

[23] D. Brand, "Incremental synthesis," *Proceedings of the International Conference on Computer Aided Design*, 1992, pp. 126-129

[24] E. J. Aas, K. Klingsheim, and T. Steen, "Quantifying design quality: A model and design experiments," *Proc. EUROASIC'92, IEEE Computer Society Press* , 1992, pp. 172-177

[25] E.J. Aas. "Towards a Unified Model for Test and Design Quality," *Proc. of the 5th Baltic Electronics Conference*, Tallinn, Estonia, Oct. 1996, pp. 17-20

[26] R.Ubar, D.Borrione, "Localization of Single Gate Design Errors in Combinational Circuits by Diagnostic Information about Stuck-at Faults," *Proc. of the 2nd Int. Workshop on Design and Diagnostics of Electronic Circuits and Systems*, Szczyrk, Poland, Sept. 2-4, 1998, pp. 73-79

[27] R.Ubar, D.Borrione, "Generation of Tests for Localization of Single Gate Design Errors in Combinational Circuits Using the Stuck-at Fault Model," *Proc. of the 11th IEEE Brazilian Symp. on IC Design*, Rio de Janeiro, Brazil, Sept. 30 - Oct. 3, 1998, pp. 51-54

[28] R. Ubar and A. Jutman, "Hierarchical Design Error Diagnosis in Combinational Circuits by Stuck-at Fault Test Patterns," *Proc. of 6th International Conference*

Mixed Design of Integrated Circuits and Systems, Kraków, June 17-19, 1999, (to appear)

[29] A. Jutman and R. Ubar, "Design Error Localization in Digital Circuits by Stuck-At Fault Test Patterns," *Proc. of 22 International Conference on Microelectronics*, 19-22 Sept. 1999 (to appear)

[30] R. Ubar, "Test Synthesis with Alternative Graphs," *IEEE Design and Test of Computers*, Spring, 1996, pp. 48-59

[31] R.Ubar, J.Raik, "Multi-Valued Simulation with Binary Decision Diagrams," *Proc. of the IEEE European Test Workshop*, Cagliari, May 28-30, 1997

[32] J.Raik, R.Ubar, "Feasibility of Structurally Synthesized BDD Models for Test Generation," *Proc. of the IEEE European Test Workshop*, Barcelona, May 27-29, 1998, pp.145-146

[33] G. Jervan, A. Markus, P. Paomets, J. Raik, R. Ubar, "Turbo Tester: A CAD System for Teaching Digital Test," *Microelectronics Education*, Kluver Academic Publishers, 1998, pp. 287-290

[34] W. H. Wolf, "Hardware-Software Co-Design of Embedded Systems", *Proc. of the IEEE, vol. 82, no. 7*, July 1994, pp. 967-989

[35] W. R. Simpson, J. W. Sheppard, *System Test and Diagnosis*, Kluver Academic Publishers, Dordrecht, 1994

[36] D.K. Pradhan et al., *Fault-Tolerant Computing: Theory and Techniques*, Prentice-Hall, Engelwood Cliffs, New Jersey, 1986

[37] D.K. Pradhan et al., *Fault-Tolerant Computer System Design*, Prentice-Hall PTR, Upper Saddle River, New Jersey, 1996

[38] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN," *In Int. Symposium on Circuits and Systems, Special Session on ATPG and Fault Simulation*, 1985

APPENDIX A. DESIGN ERROR LOCALIZATION TOOL

```
// Functions for suspected set of faulty gates localization
// Artur Jutman

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "locgate.h"
#include "vector.h"
#include "messages.h"
#include "diagnosis.h"
#include "readpath.h"
#include "options.h"


void LocateGate(void)
{
 int vector,output,node,i,j;
 char *temp1;                           // Temporary arrays
 char FaultType;
 int CurGate;               //

 if(!(temp1=(char*)malloc(NodCount*sizeof(char))))
  Error("Out of memory",-1);

//         ---------- Filling FaultyOuts table ----------
 for(vector=0; vector<vcount; vector++) //
  for(output=0; output<OutCount; output++)
   if(etalon_outputs[vector][output]-'0'!=
               vects[vector][VarCount-OutCount+output])
    FaultyOuts[vector][output]=1;    // When the test T(vector) showed
   else                              // an error on the output - put 1
    FaultyOuts[vector][output]=0;

 for(i=0; i<vcount; i++)
  FaultyOuts[i][OutCount]=0;         // Initializing
 for(vector=0; vector<vcount; vector++)
  for(output=0; output<OutCount; output++)
   if(FaultyOuts[vector][output])
    {
     FaultyOuts[vector][OutCount]=1;
     break;
    }

//     ---------- Filling FaultyNodes & FfreeNodes tables ----------
 for(node=0; node<NodCount; node++)
  {
   FaultyNodes[node]=X;                 // Initializing FaultyNodes
   FfreeNodes[node]=X;                  // and FfreeNodes
  }
 for(vector=0; vector<vcount; vector++) // Filling design_ftable table
  if(FaultyOuts[vector][OutCount])      // This vector has shown an error
   {
    for(node=0; node<NodCount; node++)
     temp1[node]=D;                     // Initializing temp1
    for(output=0; output<OutCount; output++)
     if(FaultyOuts[vector][output])     // This output has shown an error
      for(node=0; node<NodCount; node++)
       temp1[node]=temp1[node] & out_ftable[output][vector][node];
     else                               // This output hasn't shown an error
      for(node=0; node<NodCount; node++)
```

```
       FfreeNodes[node]=FfreeNodes[node] | out_ftable[output][vector][node];
    for(node=0; node<NodCount; node++)
     FaultyNodes[node]=FaultyNodes[node] | temp1[node];
   }
  else                          // This vector hasn't shown an error
   {
    for(output=0; output<OutCount; output++)
     for(node=0; node<NodCount; node++)
      FfreeNodes[node]=FfreeNodes[node] | out_ftable[output][vector][node];
   }
 for(node=0; node<NodCount; node++)
  FaultyNodes[node]=FaultyNodes[node] & ~FfreeNodes[node];  // Subtraction

//          ---------- Filling FaultyGates table ----------
 for(i=0; i<GateCount; i++)
  FaultyGates[i]=X;                  // Initializing
 for(node=0; node<NodCount; node++)
  if(FaultyNodes[node]!=X)           // If there is a fault in this node
   {
    FaultType=FaultyNodes[node];
    for(i=0; i<Node_Gate[node].length; i++)
     {
      CurGate=Node_Gate[node].num[i];
      if(GateType[CurGate].type<3)   // If there is an invertor
       if(FaultType!=D)
        FaultType=FaultType^003;     // Inverting the last two bits
      FaultyGates[CurGate]=FaultyGates[CurGate] | FaultType;
     }
   }
 for(node=0; node<NodCount; node++)
  if(FfreeNodes[node]==D)           // If there is no fault in this node
   for(i=0; i<Node_Gate[node].length; i++)
    FaultyGates[Node_Gate[node].num[i]]=X;  // The gate is fault-free


 return;
}


// This function initializes the required memory structures
void LocateGate_init(void)
{
 int i;

 if(!(FaultyOuts=(char**)malloc(vcount*sizeof(char*))))
  Error("Out of memory",-1);
 for(i=0; i<vcount; i++)
  if(!(FaultyOuts[i]=(char*)malloc((OutCount+1)*sizeof(char))))
   Error("Out of memory",-1);

 if(!(FfreeNodes=(char*)malloc(NodCount*sizeof(char))))
  Error("Out of memory",-1);

 if(!(FaultyNodes=(char*)malloc(NodCount*sizeof(char))))
  Error("Out of memory",-1);

 if(!(FaultyGates=(char*)malloc(GateCount*sizeof(char))))
  Error("Out of memory",-1);

}
```

APPENDIX B. DIAGNOSTIC RESULTS FOR C17 ISCAS'85 BENCHMARK

```
Single Gate Design Error Diagnosis

Reading paths... OK

Reading gate names and types... OK


Generating NOR instead of inst_0>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXX011
Suspected nodes (total): 3
Suspected gates:
0XXX1X
Their names:
inst_0>o [0];  inst_4>o [1];
Suspected gates (total): 2


Generating OR instead of inst_0>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXXX00
Suspected nodes (total): 2
Suspected gates:
1XXXXX
Their names:
inst_0>o [1];
Suspected gates (total): 1


Generating AND instead of inst_0>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXX0DD
Suspected nodes (total): 3
Suspected gates:
DXXXDX
Their names:
inst_0>o [D];  inst_4>o [D];
Suspected gates (total): 2


Inserting inverter to one of the inputs of inst_0>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXXD0
Suspected nodes (total): 2
Suspected gates:
DXXXXX
```

```
Their names:
inst_0>o [D];
Suspected gates (total): 1


Generating NOR instead of inst_1>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X11XX00XXXX
Suspected nodes (total): 4
Suspected gates:
X0X1XX
Their names:
inst_1>o [0];  inst_3>o [1];
Suspected gates (total): 2


Generating OR instead of inst_1>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X001XXXXXXX
Suspected nodes (total): 3
Suspected gates:
X1XXXX
Their names:
inst_1>o [1];
Suspected gates (total): 1


Generating AND instead of inst_1>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XDDD000XXXX
Suspected nodes (total): 6
Suspected gates:
XDD1X0
Their names:
inst_1>o [D];  inst_2>o [D];  inst_3>o [1];  inst_5>o [0];
Suspected gates (total): 4


Inserting inverter to one of the inputs of inst_1>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XD01X00XXXX
Suspected nodes (total): 5
Suspected gates:
XDX1XX
Their names:
inst_1>o [D];  inst_3>o [1];
Suspected gates (total): 2


Generating NOR instead of inst_2>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X0011XXXXXX
Suspected nodes (total): 4
```

```
Suspected gates:
X10XXX
Their names:
inst_1>o [1];  inst_2>o [0];
Suspected gates (total): 2


Generating OR instead of inst_2>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXX00XXX1XX
Suspected nodes (total): 3
Suspected gates:
XX1XXX
Their names:
inst_2>o [1];
Suspected gates (total): 1


Generating AND instead of inst_2>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X00DDXXX1XX
Suspected nodes (total): 5
Suspected gates:
X1DX0X
Their names:
inst_1>o [1];  inst_2>o [D];  inst_4>o [0];
Suspected gates (total): 3


Inserting inverter to one of the inputs of inst_2>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X00D0XXX1XX
Suspected nodes (total): 5
Suspected gates:
X1DXXX
Their names:
inst_1>o [1];  inst_2>o [D];
Suspected gates (total): 2


Generating NOR instead of inst_3>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXX110XXX
Suspected nodes (total): 3
Suspected gates:
XXX0X1
Their names:
inst_3>o [0];  inst_5>o [1];
Suspected gates (total): 2


Generating OR instead of inst_3>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
```

```
X1XXX00XXXX
Suspected nodes (total): 3
Suspected gates:
XXX1XX
Their names:
inst_3>o [1];
Suspected gates (total): 1


Generating AND instead of inst_3>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X1XXXDD0XXX
Suspected nodes (total): 4
Suspected gates:
XXXDXD
Their names:
inst_3>o [D];  inst_5>o [D];
Suspected gates (total): 2


Inserting inverter to one of the inputs of inst_3>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X1XXXD0XXXX
Suspected nodes (total): 3
Suspected gates:
XXXDXX
Their names:
inst_3>o [D];
Suspected gates (total): 1


Generating NOR instead of inst_4>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXX100
Suspected nodes (total): 3
Suspected gates:
1XXX0X
Their names:
inst_0>o [1];  inst_4>o [0];
Suspected gates (total): 2


Generating OR instead of inst_4>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXX011
Suspected nodes (total): 3
Suspected gates:
0XXX1X
Their names:
inst_0>o [0];  inst_4>o [1];
Suspected gates (total): 2


Generating AND instead of inst_4>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000
```

```
Suspected nodes
XXXXXXXXDDD
Suspected nodes (total): 3
Suspected gates:
DXXXDX
Their names:
inst_0>o [D];  inst_4>o [D];
Suspected gates (total): 2


Inserting inverter to one of the inputs of inst_4>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
XXXXXXXXD11
Suspected nodes (total): 3
Suspected gates:
0XXXDX
Their names:
inst_0>o [0];  inst_4>o [D];
Suspected gates (total): 2


Generating NOR instead of inst_5>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X1XXX001XXX
Suspected nodes (total): 4
Suspected gates:
XXX1X0
Their names:
inst_3>o [1];  inst_5>o [0];
Suspected gates (total): 2


Generating OR instead of inst_5>o - NAND gate
Analyzing... OK
Time, used by process: 0.010000

Suspected nodes
XXXXX110XXX
Suspected nodes (total): 3
Suspected gates:
XXX0X1
Their names:
inst_3>o [0];  inst_5>o [1];
Suspected gates (total): 2


Generating AND instead of inst_5>o - NAND gate
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X1XXXDDDXXX
Suspected nodes (total): 4
Suspected gates:
XXXDXD
Their names:
inst_3>o [D];  inst_5>o [D];
Suspected gates (total): 2


Inserting inverter to one of the inputs of inst_5>o - NAND gate
```

```
Analyzing... OK
Time, used by process: 0.000000

Suspected nodes
X1XXXDD0XXX
Suspected nodes (total): 4
Suspected gates:
XXXDXD
Their names:
inst_3>o [D];  inst_5>o [D];
Suspected gates (total): 2

Spectrum of suspected nodes
2: 2   3: 12   4: 6   5: 3   6: 1
The total number of experiments is 24
Minimal number of suspected nodes is 2
Maximal number of suspected nodes is 6
Average number of suspected nodes is 3.541667

Spectrum of suspected gates
1: 6   2: 16   3: 1   4: 1
The total number of experiments is 24
Minimal number of suspected gates is 1
Maximal number of suspected gates is 4
Average number of suspected gates is 1.875000
```

APPENDIX C. EXAMPLES OF FILES NEEDED TO RUN THE DIAGNOSTIC PROGRAM

MACRO-LEVEL SSBDD MODEL FILE: C17.AGM

```
STAT#    11 Nods,  10 Vars,  5 Grps,  5 Inps,  0 Cons,  2 Outs

MODE#   STRUCTURAL


VAR#   0:  (i_______)  "i_5"
VAR#   1:  (i_______)  "i_4"
VAR#   2:  (i_______)  "i_3"
VAR#   3:  (i_______)  "i_2"
VAR#   4:  (i_______)  "i_1"

VAR#   5:  (________)  "i_3"
GRP#    0: BEG =   0, LEN =   1 -----
   0   0:  (____)   (      0      0)  V = 2     "i_3"

VAR#   6:  (________)  "inst_1>o"
GRP#    1: BEG =   1, LEN =   2 -----
   1   0:  (I___)   (      1      0)  V = 1     "i_4"
   2   1:  (I___)   (      0      0)  V = 5     "inst_1>i_2"

VAR#   7:  (________)  "inst_2>o"
GRP#    2: BEG =   3, LEN =   2 -----
   3   0:  (I___)   (      1      0)  V = 6     "inst_2>i_1"
   4   1:  (I___)   (      0      0)  V = 3     "i_2"

VAR#   8:  (_o______)  "o_2"
GRP#    3: BEG =   5, LEN =   3 -----
   5   0:  (____)   (      2      1)  V = 0     "i_5"
   6   1:  (____)   (      2      0)  V = 6     "inst_3>i_2"
   7   2:  (I___)   (      0      0)  V = 7     "inst_5>i_2"

VAR#   9:  (_o______)  "o_1"
GRP#    4: BEG =   8, LEN =   3 -----
   8   0:  (I___)   (      1      0)  V = 7     "inst_4>i_1"
   9   1:  (____)   (      0      2)  V = 5     "inst_0>i_1"
   10   2:  (____)  (      0      0)  V = 4     "i_1"
```

GATE-LEVEL SSBDD MODEL FILE: C17.GATE.AGM

```
STAT#    15 Nods,  14 Vars,  9 Grps,  5 Inps,  0 Cons,  2 Outs

MODE#   STRUCTURAL


VAR#   0:  (i_______)  "i_5"
VAR#   1:  (i_______)  "i_4"
VAR#   2:  (i_______)  "i_3"
VAR#   3:  (i_______)  "i_2"
VAR#   4:  (i_______)  "i_1"

VAR#   5:  (________)  "i_3"
```

```
GRP#    0: BEG =   0, LEN =   1 -----
   0   0:  (____)   (      0      0)  V = 2     "i_3"

VAR#   6:  (________)  "inst_0>o"
GRP#    1: BEG =   1, LEN =   2 -----
   1   0:  (I___)   (      1      0)  V = 5     "inst_0>i_1"
   2   1:  (I___)   (      0      0)  V = 4     "i_1"

VAR#   7:  (________)  "inst_1>o"
GRP#    2: BEG =   3, LEN =   2 -----
   3   0:  (I___)   (      1      0)  V = 1     "i_4"
   4   1:  (I___)   (      0      0)  V = 5     "inst_1>i_2"

VAR#   8:  (________)  "inst_2>o"
GRP#    3: BEG =   5, LEN =   2 -----
   5   0:  (I___)   (      1      0)  V = 7     "inst_2>i_1"
   6   1:  (I___)   (      0      0)  V = 3     "i_2"

VAR#   9:  (________)  "inst_3>o"
GRP#    4: BEG =   7, LEN =   2 -----
   7   0:  (I___)   (      1      0)  V = 0     "i_5"
   8   1:  (I___)   (      0      0)  V = 7     "inst_3>i_2"

VAR#   10:  (________)  "inst_4>o"
GRP#    5: BEG =   9, LEN =   2 -----
   9   0:  (I___)   (      1      0)  V = 8     "inst_4>i_1"
   10   1:  (I___)  (      0      0)  V = 6     "inst_0>o"

VAR#   11:  (________)  "inst_5>o"
GRP#    6: BEG =   11, LEN =   2 -----
   11   0:  (I___)  (      1      0)  V = 9     "inst_3>o"
   12   1:  (I___)  (      0      0)  V = 8     "inst_5>i_2"

VAR#   12:  (_o______)  "o_2"
GRP#    7: BEG =   13, LEN =   1 -----
   13   0:  (____)  (      0      0)  V = 11     "inst_5>o"

VAR#   13:  (_o______)  "o_1"
GRP#    8: BEG =   14, LEN =   1 -----
   14   0:  (____)  (      0      0)  V = 10     "inst_4>o"
```

INPUT TEST PATTERNS FILE: C17.TST

```
.VECTORS 5


.PATTERNS

11110hlhLL
00001lhhLL
10101hhhHH
11010lhlHH
00011lhlHH

.FAULTS

&&&&&&&&&&

.COVERAGE

22 / 22 = 100.000000 %
```

SSBDD-NODE-TO-GATE-LEVEL-PATH RELATIONSHIP FILE: C17.PAT

```
Gates: 6     Nodes: 11

   node 0:
   node 1:   1
   node 2:   1
   node 3:   2
   node 4:   2
   node 5:   3->5
   node 6:   3->5
   node 7:   5
   node 8:   4
   node 9:   0->4
   node 10:  0->4
```

FILE CONTAINING NAMES AND TYPES OF GATES: C17.GAT

```
Gates: 6

              Name          Type
gate 0 :      inst_0>o      NAND
gate 1 :      inst_1>o      NAND
gate 2 :      inst_2>o      NAND
gate 3 :      inst_3>o      NAND
gate 4 :      inst_4>o      NAND
gate 5 :      inst_5>o      NAND
```