Institute of Computer Engineering and Computer Architecture
Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Pfaffenwaldring 47, D-70569 Stuttgart

Bachelor Project Nr. 2332

Evaluation of Backtracing Based Diagnosis Algorithms

by
Maha Badreldein

Examiner :	Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Supervisor :	Dipl.-Inf. Stefan Holst
Start Date :	April 05, 2011
Submission Date :	August 31, 2011
CR Classification :	B.2.3 Reliability, Testing, and Fault-Tolerance—Diagnostics, B.2.2 Performance Analysis and Design Aids—Simulation, and B.6.1 Design Styles—Combinational logic.

Acknowledgment

I would like to begin by expressing my sincere gratitude to the people who helped me throughout my bachelor project.

I am grateful to the German University in Cairo (GUC), the German Academic Exchange Service (DAAD) and the University of Stuttgart for providing me with the opportunity to do this project in Germany.

In addition, I would like to thank Prof. Dr. Hans-Joachim Wunderlich for giving me a warm welcome and encouraging me throughout my stay. I am also indebted to my supervisor, Stefan Holst, for the guidance and support he gave me. His comments and suggestions have been of great use for my work, presentation, and thesis.

I would also like to thank my dear friends:
Eman, Engy, Yasmine, Ghada, Heba and Heidi, my roommates, for being more than sisters to me.
Mohamed, my amazing lab colleague, for being always so kind, helpful and encouraging.
Menna Amer, Hadeer, Ramy and Sarah Mosleh for giving me moral support.

Last but not least, I would like to thank my dear family for giving me everything, believing in me and being always there for me.

Abstract

With the growing size and complexity of modern circuits, more algorithms are being developed nowadays for efficient fault diagnosis. Backtracing based diagnosis algorithms are effect-cause approaches that start from the failing outputs of the circuit and try to diagnose fault locations by backtracing lines toward the circuit inputs. In this thesis, general functionality was extracted between backtracing based diagnosis algorithms and implemented as an extension to an existing diagnosis framework. Furthermore, a simple graphical user interface was developed for the extended framework. The extended framework aims at facilitating the implementation and evaluation of different backtracing based diagnosis algorithms. In order to demonstrate its powerfulness, two modern backtracing based diagnosis algorithms were implemented on top of the extended framework. A number of diagnosis experiments on benchmark circuits was carried out in order to evaluate the two implemented algorithms. The experimental tools used and the results obtained are presented.

Nomenclature

ATPG	Automatic Test Pattern Generation
BFS	Breadth-First Search
BUF	Buffer
CPT	Critical Path Tracing
CUD	Circuit Under Diagnosis
CUT	Circuit Under Test
DERRIC	Diagnosis of logic ERRors in VLSI Integrated Circuits
DSIE	Defect Site Identification and Elimination
GUI	Graphical User Interface
PI	Primary Inputs
PO	Primary Outputs
PPI	Pseudo Primary Inputs
PPO	Pseudo Primary Outputs
SLAT	Single Location At-a-Time
STF	Slow To Fall
STR	Slow To Rise

Contents

Chapter 1

Introduction

1.1 Description of the Diagnosis Problem

In order to describe the diagnosis problem precisely, one must first describe the concept of testing. Testing is the process that verifies the manufactured hardware is defect free. Its role is to detect the existence of defects in the Circuit Under Test (CUT) [5]. The diagnosis process follows the testing process in order to determine exact details about the defects [5].
The diagnosis problem can be described as follows: Given the Circuit Under Diagnosis (CUD), report the list of defects responsible for its failure. For the scope of this work, defects are reported in terms of their locations and possibly their assigned fault models. As modern circuits keep growing rapidly in size, the solution's allowed execution time as a function of the CUD size (number of gates) keeps shrinking.

1.2 Motivation

Testing is a vital step in achieving better quality and economy from the manufacturing process. In order to achieve good quality, it is important that faulty devices are detected before reaching the user. The prices of the good devices, however, must cover the manufacturing costs of both the good and the faulty devices. With the increasing size and complexity of modern circuits, fault diagnosis has become an important tool to generate information for repairing and learning from the device under test. For instance, fault diagnosis information can help locate and replace faulty replaceable units or improve the manufacturing process [5].
Defects in modern circuits may no longer be adequately modeled by the single-fault model. Moreover, with the advanced manufacturing technologies and the aggressive clocking strategies applied in modern designs, timing defects have become more common [9]. Therefore, backtracing based diagnosis approaches are particularly considered more promising in diagnosing failures with dynamic behavior [11]. On the other hand, using fault dictionaries and matching techniques could have misleading results as assuming that the results of fault simulations can match delay defects behavior is unrealistic [9].

1.3 Overview of Modern Algorithms

In this section, an overview of modern diagnosis algorithms, their capabilities and their limitations is presented. The development of modern diagnosis algorithms is driven by the nature of defects present in modern circuits. While older diagnosis algorithms made use of the single fault assumption and the classical stuck-at fault model to build their solutions, the complex nature of defects in modern circuits sometimes renders these solutions inadequate.

1.3.1 Single and Multiple Fault Diagnosis

The single fault assumption assumes a frequent testing strategy. In this strategy, the frequency of testing the CUD is high enough to ensure the probability of having more than one fault between two successive tests is sufficiently small [1][2]. This strategy, however, is considered impractical. Sometimes, a single physical defect can only be represented by a multiple fault model. Moreover, unless every test applied is successful in identifying the single fault present in its turn, there will be cases where the CUD develops a new single fault in addition to the undetected fault it has, creating a multiple fault [2].
The construction of fault dictionaries is done by simulating a number of single faults and is hence based on the single fault assumption. Furthermore, some backtracing based approaches are limited to single fault diagnosis because they use intersection procedures based on the single fault assumption. Examples of these approaches are [11] and [7].
Another assumption, that is less restricting than the single fault assumption, is the availability of Single Location At-a-Time (SLAT) patterns [13]. A failing test pattern is said to be a SLAT pattern if its failure can be explained by a single fault [4]. The algorithm presented in [13] tries to tackle the problem of multiple fault diagnosis without making assumptions about the failing patterns.

1.3.2 Fault Models

As defects in modern circuits became more complex, new fault models were developed to represent them. Consequently, modern diagnosis algorithms try to consider a wider variety of fault models in their diagnosis. For instance, bridging fault diagnosis is discussed in [12] and delay fault diagnosis is discussed in [7].

1.4 Aim of the Project

The main aim of this project is to evaluate different backtracing based diagnosing approaches by extending the existing diagnosis framework to facilitate the implementation and evaluation of such algorithms.
The initial project plan specified the following list of tasks:

1. Literature study on the different logic diagnosis approaches with emphasis on backtracing based approaches.
2. In-depth study of at least two backtracing based diagnosis approaches to extract common properties, tools and functions.
3. Study of the existing logic diagnosis framework: ADAMA.
4. Implementing the extracted general functionality in the framework.
5. Implementing at least one backtracing based logic diagnosis approach completely on top of the extended framework.
6. Demonstration of the diagnosis implementation by a number of experiments on benchmark circuits.

Although not one of the initial tasks, a very simple Graphical User Interface (GUI) was implemented to help visualize the internal representation of the CUD and to display results. The purpose of such an addition is to increase the usability of the extended framework in implementing and evaluating algorithms.

1.5 Overview of the Thesis

This document is organized as follows: Chapter 2 introduces the basic definitions, terminology, data structures and algorithms necessary to understand backtracing based diagnosis algorithms. It also presents a summary of the two backtracing based diagnosis algorithms selected for study and implementation in this work as well as a summary of the existing framework ADAMA. In chapter 3, the common properties, tools and functions extracted between the two backtracing based diagnosis algorithms are presented along with their implementation details, constructing the extended framework. In chapter 4, the implementation of the two selected algorithms on top of the extended framework is presented. In chapter 5, the graphical user interface of the extended framework is presented. Chapter 6 presents the tools used in the experimental evaluation of the two implemented algorithms and their results. Finally, chapter 7 concludes the thesis and proposes some future work possibilities.

Chapter 2

Background

2.1 Definitions and Terminology

This section lists the basic definitions and terminology used throughout the document.

A combinational circuit is a circuit consisting of logic gates whose outputs can be determined at any time from the current inputs with no need to refer to any previous inputs [10].

A sequential circuit is a circuit consisting of both logic gates and memory elements. Their outputs can be determined at any time from the current inputs and the memory elements state. Since the memory elements state depends on previous inputs, the outputs depend on both current and previous inputs [10].

A defect is a difference between the design of the circuit and its actual hardware implementation. It can occur during the manufacturing or during the use of the circuit [5].

An error is a wrong signal observed at the output of the circuit [5]. An error can be caused, for example, by design errors or fabrication defects [2].

A fault is an abstract way of representing a defect. It can be detected if it causes some observable error [2][5].

Fanout the number of branches a signal line feeds.

Fanout-free circuit is a circuit in which all lines have a fanout of one.

Sensitive gate input A gate input is considered sensitive if complementing its value results in a different gate output [3].

Forward implication is determining the unique value of a gate output given the gate inputs values [5].

Backward implication is determining the unique values of all inputs to a gate given the gate output value and possibly some of the gate inputs values [5].

Stuck-at fault is a classical fault model where a line is permanently set to the same logic value 0 or 1 [2].

Bridging fault is a fault model where a short occurs between two or more lines, causing them to have the same value. In many cases when two shorted lines originally have different values, one value (named the strong value) overrides the other and is observed on both lines. A bridging fault is called an AND bridging fault if it has a strong value of 0 and an OR bridging fault if it has a strong value of 1 [2].

2.2 The ATPG Problem

Automatic Test Pattern Generation (ATPG) is the process of automatically generating input test patterns to be applied to a circuit during the testing and diagnosis processes [5]. The purpose is to generate patterns capable of:

1. Detecting faults.
2. Locating and distinguishing between faults.

The difficulty of testing and diagnosing a circuit increases with its size. A circuit with x primary inputs gives an ATPG 2^x different input combinations (patterns) to be considered. In fact, the ATPG problem was found to be NP-complete, which means that no polynomial time solution for it has been found so far and it is assumed to be of exponential complexity [5]. Hence, ATPG algorithms try different heuristics in order to achieve high fault coverage by generating an affordable number of patterns in polynomial time.

2.3 Cause-effect Approach versus Effect-cause Approach

Diagnosis algorithms can be classified into cause-effect approaches and effect-cause approaches.
Cause-effect approaches simulate a list of faults in advance before the diagnosis process actually starts. A fault dictionary is constructed in order to store each fault and its corresponding simulation response of the circuit. The diagnosis process then simulates the CUD and tries to match the simulation response with one of the previously obtained responses stored in the fault dictionary. Successful matches are assumed to indicate the presence of their corresponding faults in the CUD [2]. The advantage of this approach is that it is relatively simple. Nevertheless, it limits the universe of detectable faults to only those in the fault dictionary [4].
Effect-cause approaches, on the other hand, analyze the erroneous response of the CUD (the effect) in order to determine the faults (the cause) responsible for this response [2]. One of the main advantages of these approaches is that they do not require the simulation of every suspect fault in the CUD. Thus, the cost of propagating many suspect faults that are not actually present in the CUD is saved [3].

2.4 Two Modern Backtracing Based Algorithms

In order to achieve the goal of this project, two modern backtracing based diagnosis algorithms were selected to be studied in depth, namely DERRIC [11] and DSIE [13]. The study was used to construct a list of common properties and functions which the framework should support. The two algorithms having differences between them as well, were useful in demonstrating the flexibility and powerfulness of the extended framework when implemented on top of it. The goal of this section is to present the two algorithms in detail.

2.4.1 DERRIC

DERRIC (Diagnosis of logic ERRors in VLSI Integrated Circuits) is a diagnostic tool presented in [11]. It aims at diagnosing locations of potential defect lines as well as assigning them one or more fault models.

Restrictions and assumptions

The scope of this tool is limited to diagnosis in the combinational part of the CUD. Moreover, the algorithm used in the tool uses the single fault assumption and thus is not valid for multiple defect diagnosis.

Description of the algorithm

The algorithm targets both static and dynamic behaviors in the CUD by considering two test vectors instead of one. A new test pattern is constructed from every two consecutive test vectors V$_i$ and V$_{i-1}$. Table 2.1 shows the six-valued algebra used as an efficient representation with no need for any timing analysis [11].

C0	static 0	00
C1	static 1	11
F0	falling transition	10
R1	rising transition	01
P0	static 0-hazard	010
P1	static 1-hazard	101

Table 2.1: List of six-valued algebra signals.

The algorithm falls in the category of effect-cause approaches. First, DERRIC reads the gate level description of the CUD, the test sequence and the corresponding response matrix. Next, the diagnosis process goes as follows:

Fault-free simulation For each failing test vector V_i, fault-free simulation of the CUD is performed. First, the six-valued input signals are calculated from V_i and V_{i-1}. These input values are then propagated toward the outputs using six-valued algebra propagation tables, see [11].

Critical Path Tracing (CPT) The goal of CPT is to determine all potential defect lines. For each failing test vector V_i, CPT begins from every failing output and traces back through sensitive lines until reaching the inputs. For each failing output, CPT finally provides a list L of critical pairs of the form (LC_i, S_i) where LC_i is the name[1] of the critical line and S_i is the six-valued algebra symbol associated with that line [11].

Intersection procedure Using the single fault assumption, the actual defect line is asserted to be present in all lists formerly obtained via CPT. Hence, an intersection procedure is performed between these lists in order to produce a final list of possible defect lines. The intersection procedure used is defined as follows:

- The operator $\Cap_s$ denotes the intersection between six-valued symbols of two lines as shown in table 2.2. The symbol '-' denotes an undefined intersection meaning that the error observed on its line is caused by an upstream error propagation not by a defect on that line. Therefore, that line can be eliminated from the list of suspect lines. The D symbol denotes a signal with possible delay both on falling and rising transitions.

$\Cap_s$	C0	C1	F0	R1	P0	P1	D
C0	C0	-	C0	-	C0	-	-
C1	-	C1	-	C1	-	C1	-
F0	C0	-	F0	D	F0	D	D
R1	-	C1	D	R1	D	R1	D
P0	C0	-	F0	D	P0	D	D
P1	-	C1	D	R1	D	P1	D
D	-	-	D	D	D	D	D

Table 2.2: Intersection table [11].

- The intersection of two lists L_1 and L_2, $L_s = L_1 \Cap L_2$, is the result of the intersections between each pair (LC_i, S_i)$_1$ of the list L_1 and each pair (LC_j, S_j)$_2$ of the list L_2.
- The intersection of two pairs (LC_i, S_i)$_1$ and (LC_j, S_j)$_2$ is:
  - If $LC_i \neq LC_j$ or if $S_i \Cap_s S_j$ is undefined, then $(LC_i, S_i)_1 \Cap (LC_j, S_j)_2 = \Phi$.
  - Else if $LC = LC_i = LC_j$ then $(LC_i, S_i)_1 \Cap (LC_j, S_j)_2 = (LC, S_i \Cap_s S_j)$.

Fault model allocation After the intersection procedure, a final list of suspect lines is available along with their associated symbols. Each of these lines can now be assigned one or more fault models according to table 2.3 [11].

One advantage of this method is that fault models don't have to be considered one by one during the diagnosis of potential defect lines.

[1]Every line in the CUD has a unique name.

	C0	C1	F0,P0	R1,P1	D
Stuck at 0		x		x	
Stuck at 1	x		x		
Tn Stuck open	x		x		
Tn Stuck on		x		x	
Tp Stuck open		x		x	
Tp Stuck on	x		x		
Open 0		x		x	
Open 1	x		x		
Resistive open			x	x	x
Short Or (with any line at 1)	x		x		
Short And (with any line at 0)		x		x	
Resistive Short (with any line at 1)			x		
Resistive Short (with any line at 0)				x	
Delay StF			x		
Delay StR				x	
Delay StR & StF			x	x	x

Table 2.3: Fault Models for suspect lines symbols [11].

2.4.2 DSIE

DSIE (Defect Site Identification and Elimination) is a backtracing based diagnosis algorithm presented in [13]. The paper originally presents two algorithms: DSIE and path-based defect site elimination. It targets diagnosing circuits with multiple defects without adding restrictions on the characteristics of the failing patterns. For the scope of this work, only DSIE was studied and implemented.

Description of the algorithm

Algorithm 1 shows the description of DSIE [13].

Algorithm 1 DSIE algorithm.

```
for each failing pattern $t_k$ do
  Path-tracing to find initial candidate set for $t_k$.
end for
while first iteration or site eliminated in last iteration do
  for each failing pattern $t_k$ do
    1. Perform fault-free simulation
    2. Inject the unknown value X at each potential defect site and propagate all X
    values to outputs
    3. Assign fault-free values to all passing outputs
    4. Perform conservative implication and site elimination from passing outputs
  end for
end while
```

In order to construct a failing pattern's initial candidate set, path-tracing starts from a failing observable point (a primary output or a scanned memory element) and traces the circuit lines back according to the following rules [13]:

1. On reaching a gate output, trace back only the gate's controlling (sensitive) inputs. If all the inputs are non-controlling, trace back all inputs.
2. On reaching a fanout branch, continue backtracing from the stem.
3. Mark the lines encountered as possible defect sites and keep track of the errors associated with them. The notation used to represent a potential defect site is f/v, where f is the suspect site ID and v is the suspect faulty signal value.

The rest of the algorithm attempts to eliminate the defect-free sites added by path-tracing to the candidate sets in order to improve the algorithm complexity. The following idea is employed: A potential defect site is implied to be defect-free (eliminated) if it is found to have contributed to the defect-free downstream logic having correct values [13].
The unknown X value is injected at each potential defect site in order to implicitly consider all defect combinations and represent the simultaneous interactions between them [13].
Conservative implication is performed by applying the two following rules [13]:

Implication rule for a signal line f assigned a value $v \in \{0, 1\}$ in a failing pattern t: If f is not a potential defect site for any failing pattern, perform forward and backward implication from f. Otherwise, perform forward implication only.

Assignment rule for a signal line f implied to a value $v \in \{0, 1\}$ in a failing pattern t: If f has been implied to have the value v by forward implication, assign v to f only if it is not a potential defect site for any failing pattern. If f has been implied to have the value v by backward implication, assign v to f always.

Due to the fact that backward implication always starts from a signal line that is not a potential defect site (as defined by the implication rule), whenever a line f is assigned to a known value v by backward implication, the potential defect site f/v' can be safely eliminated.
Figure 2.1 illustrates these ideas by showing the steps performed in DSIE using only one test pattern on a very small circuit. The output line of a gate is assigned the gate label as its ID. It should be pointed out, however, that the figure does not show the complete steps of the algorithm. Since a potential defect site was eliminated, another iteration of the while loop in algorithm 1 was performed. The steps of this second iteration led to no more eliminations and thus were not shown.

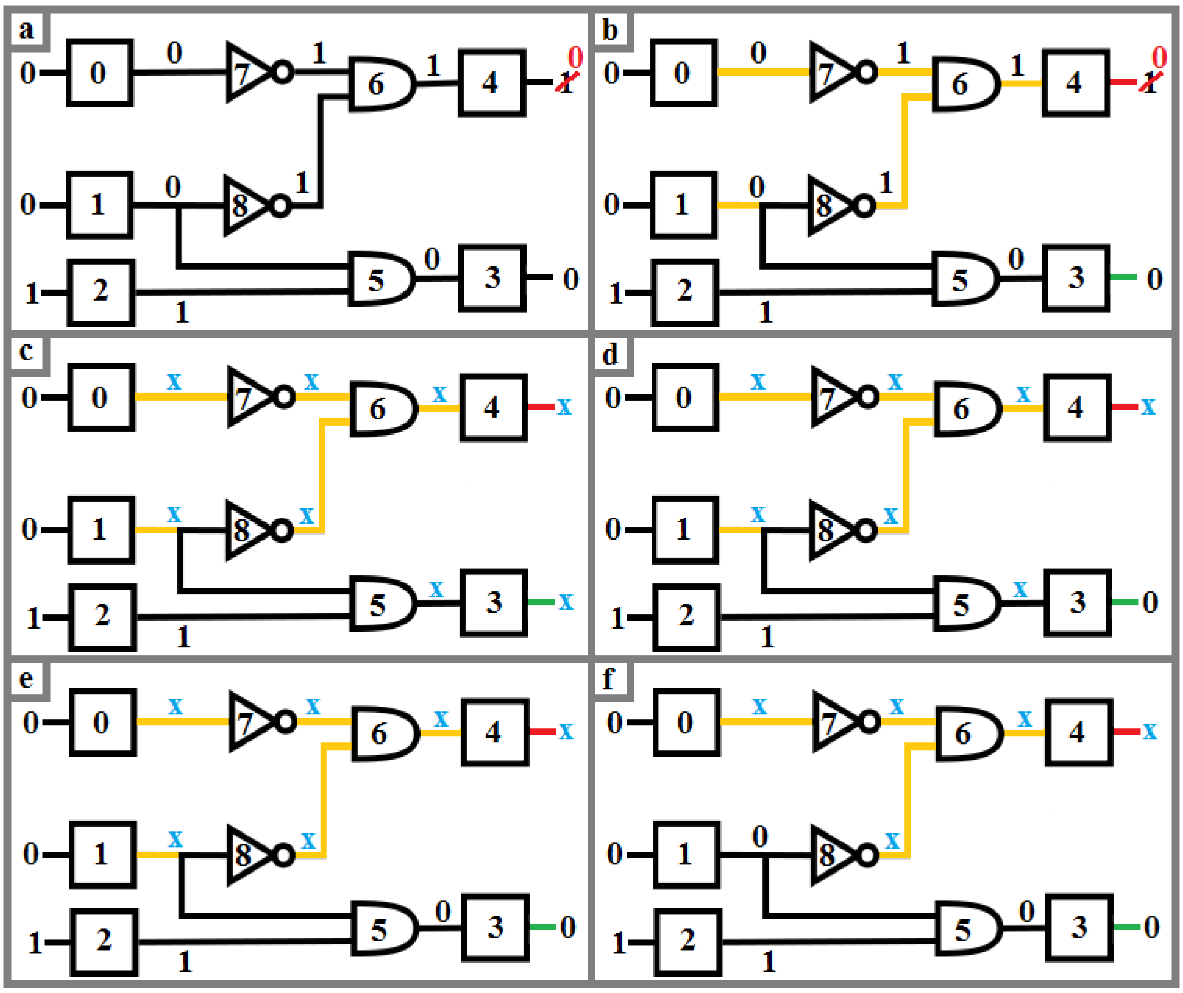


Figure 2.1: DSIE steps on a small circuit using the test pattern 001: (a) Fault-free simulation values are shown on every line, output 4 failed the actual simulation while output 3 passed. (b) Path-tracing from the failing output. Potential defect lines marked in yellow: 6/0, 7/0, 8/0, 0/1 and 1/1. (c) Unknown X values injection and propagation. (d) Fault-free value assigned to the passing output. (e) Backward implication from gate 3. Line 5 is assigned the value 0 via backward implication. (f) Backward implication from gate 5, triggered as it is not a potential defect site. Line 1 is assigned the value 0 via backward implication. Line 1, having the value 0, contributed to the correct value of the passing output 3. Hence, the potential defect line 1/1 is eliminated.

2.5 Data Structures and Algorithms

The purpose of this section is to provide a description of the data structures and algorithms used to represent and operate on the CUD.

2.5.1 Graph Representation

A graph is defined as a pair $G = (V, E)$ where V is a set of vertices or nodes and E is a collection of edges. CUDs are commonly represented using graphs where every vertex corresponds to a gate and every edge corresponds to a line between two gates.
Graphs are typically stored using either:

1. The adjacency matrix representation where every vertex is assumed to have a unique number i, $j \in \{1, 2, \cdots, \mid V \mid\}$ and the adjacency matrix is a $\mid V \mid \times \mid V \mid$ matrix $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

   This representation is considered suitable for dense graphs ($\mid E \mid$ is close to $\mid V \mid^2$) [6].

2. The adjacency list representation where every vertex u has a list that contains all vertices v such that the edge $(u, v) \in E$. Typically, the vertices in such adjacency lists are stored in an arbitrary order. This representation is considered suitable for sparse graphs ($\mid E \mid \ll \mid V \mid^2$) [6].

2.5.2 Breadth-first Search

Breadth-first search (BFS) is a simple, but important graph search algorithm. The algorithm takes as input a graph and a source vertex. Initially, all vertices are marked as unvisited. From the source vertex, the algorithm starts by visiting all vertices that are one edge away from the source vertex, followed by those that are two edges away, and so on. The first time a vertex v is reached, it gets marked as visited and the path followed in the BFS between the source node and v is the shortest path in terms of the number of edges. Hence, all vertices at distance d from the source vertex are visited before any vertices at distance $d + 1$ [6]. Algorithm 2 shows how BFS can be implemented using a queue.

Algorithm 2 BFS(Graph g, Vertex s)

```
Queue q
q.enqueue(s)
while q is not empty do
  Vertex current = q.dequeue()
  if current is not visited then
    mark current as visited
    for all Vertex v in current's neighbors do
      q.enqueue(v)
    end for
  end if
end while
```

2.6 Existing Framework: ADAMA

The purpose of this section is to provide an overview of the existing framework (ADAMA) and to specifically highlight the ADAMA functions used in the scope of this work. ADAMA is one of the projects within the ITI department. Its core function is logic simulation and its implementation is Java pure. The tools provided by ADAMA are organized in a task system. Each tool, represented by a task, can be run using ADAMA's command line interface [8].

2.6.1 Leveled Graph Node

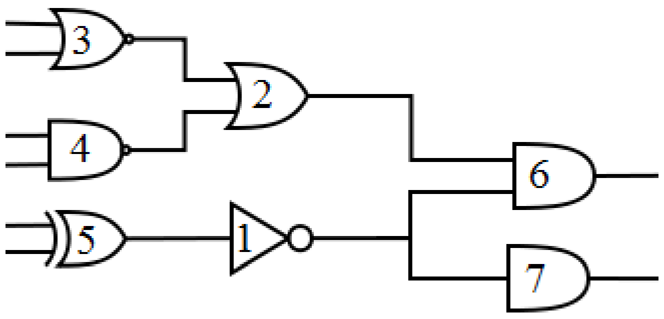
The class LeveledGraphNode in ADAMA models a circuit gate as a leveled graph[2] node. ADAMA supports twelve primitive gates types with at most two inputs, namely:

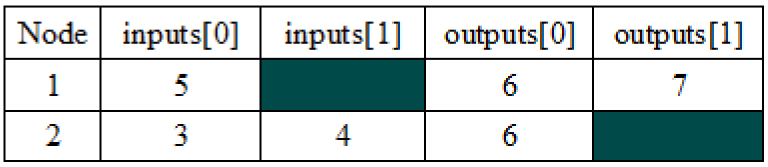
- Buffer (BUF) gate, one input.
- NOT gate, one input.
- AND gate, two inputs.
- NAND gate, two inputs.
- OR gate, two inputs.
- NOR gate, two inputs.
- XOR gate, two inputs.
- XNOR gate, two inputs.
- Input gate, no inputs.
- Output gate, one input.
- Slow to fall gate (STF), one input.

[2]Leveled Graphs are described in subsection 2.6.2.

- Slow to rise gate (STR), one input.

Every node has an array of input nodes and an array of output nodes. A connection between two gates is modeled as a link (edge) between the output of one node and the input of the other.
In order to link two nodes, simply the source node is inserted to the end node's inputs array and the end node is inserted to the source node's outputs array. This representation is similar to the adjacency list graph representation described in subsection 2.5.1.
While the length of a node's inputs array is determined by its gate type, the length of its outputs array represents the number of its fanout branches. Thus, representing fanout branches requires no extra modeling in ADAMA's leveled graph node.
Figure 2.2 shows an example of how lines between gates might be represented with leveled graph nodes. It is important to point out, however, that this is merely one of the possible representations, but not guaranteed to be the actual representation. Importing a circuit into ADAMA does not necessarily keep the order of inputs to a gate. Hence, another possible representation could list node(4) as node(2)'s inputs[0] and node(3) as node(2)'s inputs[1].





Node	inputs[0]	inputs[1]	outputs[0]	outputs[1]
1	5	■	6	7
2	3	4	6	■

Figure 2.2: Example of inputs and outputs arrays for two leveled graph nodes.

2.6.2 Leveled Graphs

The class LeveledGraph in ADAMA represents a combinational circuit consisting only of primitive gates with a maximum of two inputs. It models gates using the class LeveledGraphNode. The circuit is stored as a leveled graph using a two dimensional jagged array of nodes. Each row in the array represents a level and the depth of a graph is the index of its last level[3]. Nodes in a leveled graph must satisfy the following property:
$\forall$ level$_x$, $x \in \{0, 1, \cdots, depth\}$, $\forall$ node n stored in level$_x$:

1. $\forall$ node i in n's inputs array, i is stored in level$_y$: $y \in \{0, 1, \cdots, x - 1\}$.

2. $\forall$ node o in n's outputs array, o is stored in level$_z$: $z \in \{x + 1, x + 2, \cdots, depth\}$.

Input nodes are stored in the very first level while output nodes are stored in the last one. Within these levels, ports are ordered: PI (Primary Inputs), PPI (Pseudo Primary Inputs), PPO (Pseudo Primary Outputs) and PO (Primary Outputs).
For instance, figures 2.3 and 2.4 show a simple circuit and its corresponding leveled graph in ADAMA.

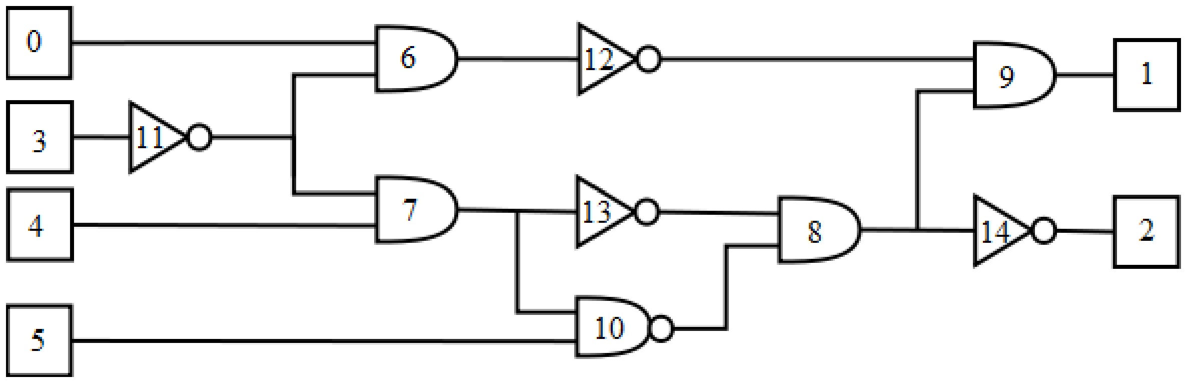


Figure 2.3: Example Circuit [11].

[3]Zero-based indexing.

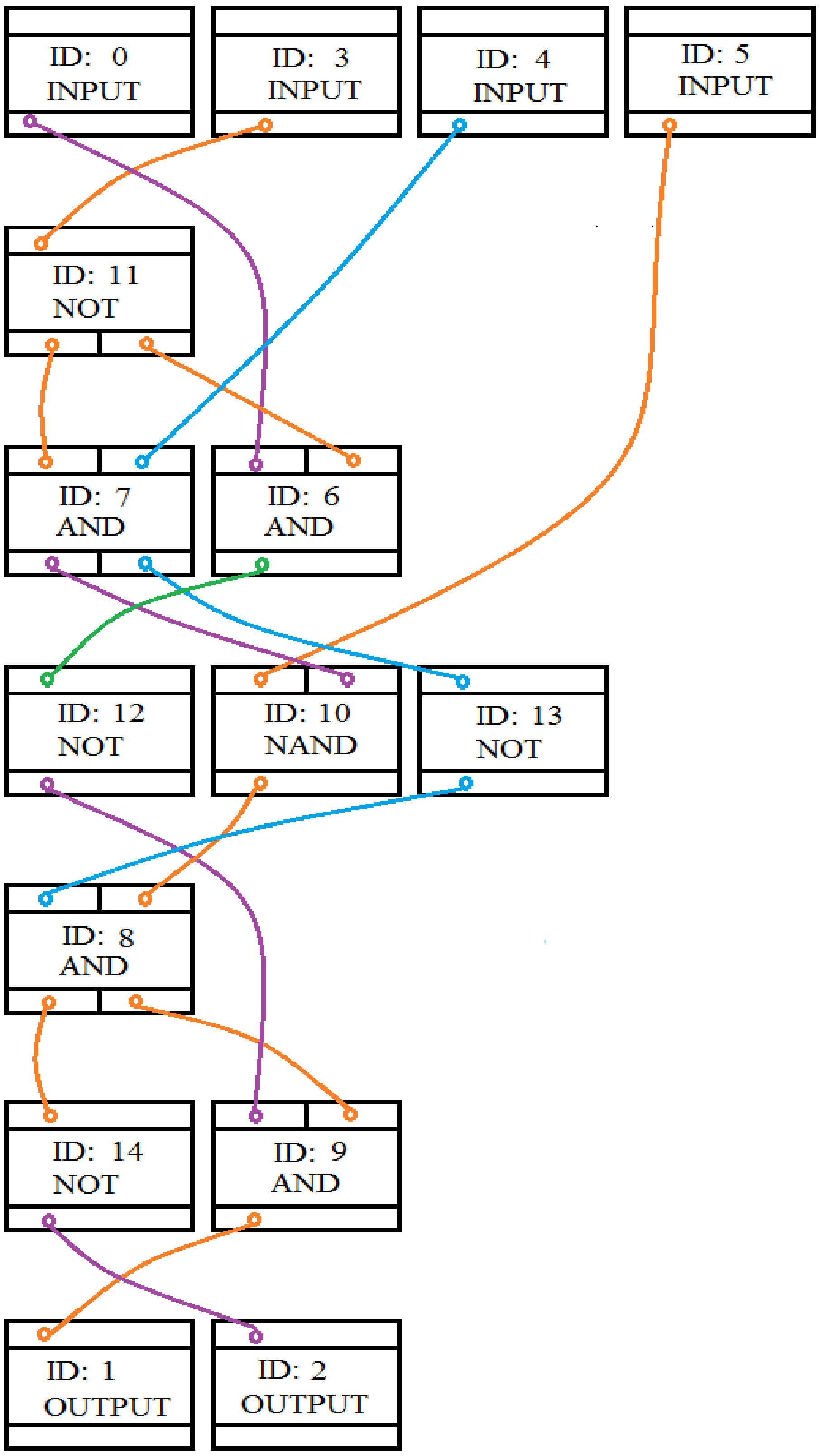


Figure 2.4: Leveled graph representation of figure 2.3.

2.6.3 Faults

In ADAMA, a leveled graph keeps a list of hooks which represent the faults injected in the circuit: ArrayList<ILeveledGraphSimHook> hooks. Any class representing a fault in ADAMA must implement the interface ILeveledGraphSimHook so that it can be injected in the CUD. Injecting such a fault is simply done by calling the method addHook(fault). The injected faults can be removed by calling the method clearHooks(). Calling the method sim(patternBlock) on a leveled graph runs the simulation of the given pattern block considering the hooks added, hence imitating the behavior of a faulty circuit.

2.6.4 Patterns

The class Pattern in ADAMA stores both the inputs and the responses of a test pattern. When a test pattern is simulated, responses are updated in-place [8]. The class Pattern-Block represents a group of at most 64 patterns which can be simulated in parallel. ADAMA offers two possibilities for constructing a new PatternBlockList:

1. Reading patterns from an existing file: new PatternBlockList (”patternsFile-Name.p”).
2. Generating random patterns: PatternBlockList.randomList(leveledGraph, number-OfPatterns, leveledGraph.numPorts(), random, careBitWeight).

This existing representation, however, only supported patterns of the signals 0, 1 and x.

Chapter 3

Extending the Existing Framework

In this chapter, the methodology followed in extending ADAMA is presented. The purpose was creating a powerful framework facilitating the implementation and evaluation of backtracing based diagnosis algorithms. Section 3.1 presents a list of the common properties, tools and functions extracted. Section 3.2 shows how this list was implemented to extend the existing framework.

3.1 Extraction of Common Properties, Tools and Functions

This section presents the result of an in-depth study of two modern backtracing based diagnosis algorithms: DERRIC [11] and DSIE [13]. For a summary of the two algorithms, please refer to section 2.4.
The two algorithms were found to exhibit many similarities in many aspects as listed below.

The CUD Reading the gate-level description of the CUD.

CUD lines CUD lines are used to report the locations of possible defects in the CUD by diagnosis algorithms.

Algebra The algebra of a diagnosis algorithm defines the set of required signals and their propagation tables for different logic gates.

Test patterns A generic test pattern was extracted to fit the requirements of different backtracing based diagnosis algorithms. It can be easily constructed from two-valued (0,1) algebra test vectors. The extracted test pattern keeps two test vectors, current and previous. According to the algebra used by the algorithm, the values applied to the CUD inputs are obtained from the current test vector only or from both the current and the previous test vectors. Furthermore, this extracted test pattern keeps track of the responses at the outputs for both the fault-free simulation and the actual simulation of the CUD. Failing outputs are marked and the test pattern itself is failing if it has at least one failing output. A test pattern also has a list of potential defect lines.

Sensitive inputs The ability to determine the sensitive inputs to a gate in the CUD is a common function required in backtracing based diagnosis algorithms.

Fanout branches Backtracing a fanout branch requires the ability to determine its stem and continue backtracing from there.

Fault injection Fault injection is a common tool needed in evaluating backtracing based diagnosis algorithms. It is used to simulate the presence of defects in the CUD so that the correctness and accuracy of the diagnosis algorithm can be evaluated.

Fault-free simulation Fault-free simulation of the CUD is used to provide information for determining the sensitive inputs of the CUD gates, determining passing and failing test patterns, possibly assigning fault models to potential defect location or eliminating some defect locations.

Backtracing Backtracing is a common function used in backtracing based diagnosis algorithms to find the initial set of potential defect sites. The procedure starts from failing outputs and traces CUD lines back toward the CUD inputs. The CUD lines are traced according to certain rules. If the line is a fanout branch, backtracing continues from its stem. If the line is a gate output, backtracing continues only from the sensitive (controlling) inputs. In case all inputs are non-controlling, backtracing continues from all of them. During this procedure, CUD lines visited are marked as potential defect lines.

Diagnosis report A diagnosis report was extracted as a convenient way commonly used to display the diagnosis result. It reports the final list of suspect lines and the fault models assigned to them, if any. It also reports useful statistics about the accuracy of the diagnosis process.

3.2 Implementation

In this section, the implementation details of the general functionality extracted are presented. Figure 3.1 shows an overview of the classes, interfaces and methods implemented in the extended framework.

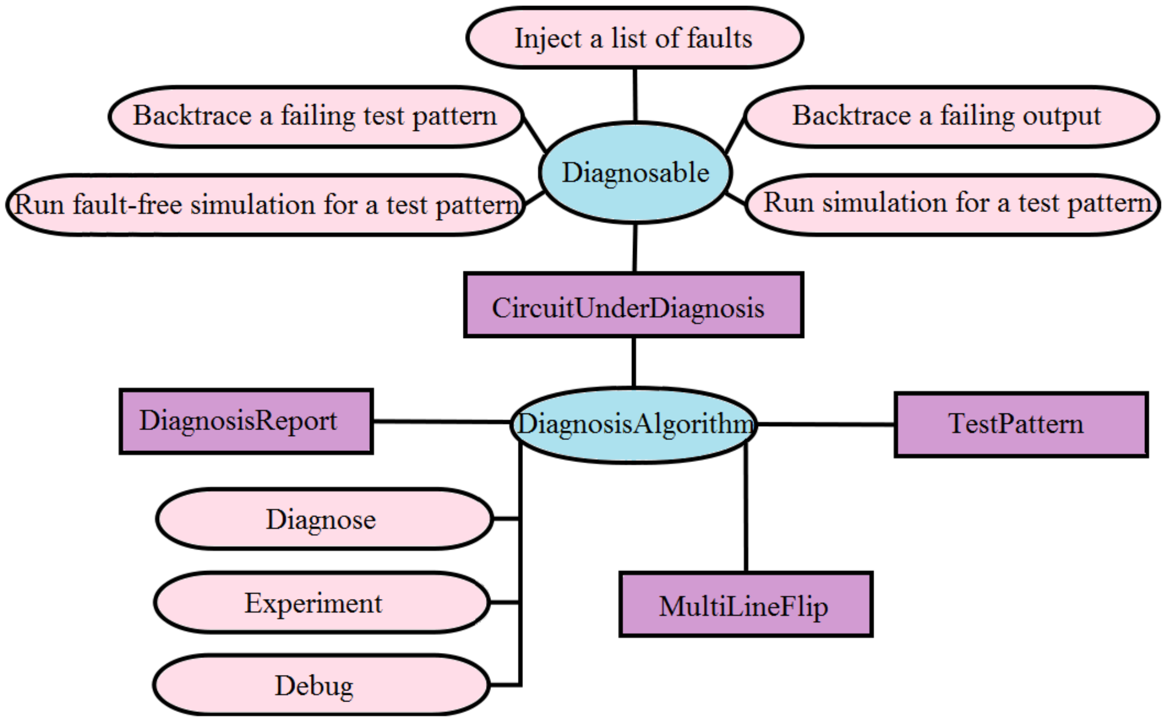

Inject a list of faults
Backtrace a failing test pattern
Backtrace a failing output
Diagnosable
Run fault-free simulation for a test pattern
Run simulation for a test pattern
CircuitUnderDiagnosis
DiagnosisReport
DiagnosisAlgorithm
TestPattern
Diagnose
Experiment
Debug
MultiLineFlip

Figure 3.1: An overview of the extended framework implementation.

3.2.1 The CUDSignal Class

Even though ADAMA efficiently provides the functions needed in logic simulation in the class LeveledGraphSim, these functions operated only on the (0,1) logic signals. Introducing more signals such as the unknown value X or six-valued algebra signals was a necessity for implementing and evaluating different backtracing based diagnosis algorithms. Hence, the class CUDSignal was implemented to play this role. Ten integer values were included as static final variables in this class, namely: ZERO, ONE, X, C0, C1, F0, R1, P0, P1 and D. ZERO and C0 can be considered as two synonyms for the same signal value. The name C0, however, emphasizes the idea of considering the initial and final values of the signal by the algebra. The same applies for ONE and C1.

Remark A CUDSignal with the value -1 is interpreted as undefined or invalid.

Every instance of the class CUDSignal has a private int signalValue which can be set to one of the ten previously mentioned values. The class offers the following functionalities for manipulating CUD signals:

public static int propagationTable[][][] This generic propagation table stores the results of propagating different signal values through different logic gates in a three dimensional array. The first index is a node type. The second index is the first input signal value. The third index is the second input signal value or 0 if the node has only one input signal. For instance, the cell propagationTable-[LeveledGraphNode.NODE_TYPE_AND][CUDSignal.ZERO][CUDSignal.ONE] stores the propagation value of the signals 0 and 1 through an AND gate.

public static int complement(int x) Looks up the negation of the given signal value in the propagation table and returns it.

public int getInitialValue() Returns the initial value of a CUDSignal, as listed in table 2.1.

public int getFinalValue() Returns the final value of a CUDSignal, as listed in table 2.1.

3.2.2 The CUDLine Class

The CUDLine class models lines in the CUD. A line has 'start' and 'end' nodes and a value (CUDSignal). This value can be used to store either the fault-free value on that line or the actual value according to the algorithm requirements. The ID of a line is the ID of its start node. Hence, all lines of a fanout branch have the same ID.

3.2.3 The TestPattern Class

The TestPattern class is a generalization of the existing Pattern class in ADAMA. For instance, let us consider the circuit in figure 2.3. A possible TestPattern is C0C1F0C1. There are two possible ways to create this TestPattern:

1. From the two consecutive binary algebra patterns: 0111 and 0101, and specifying the type to be six-valued algebra, a TestPattern is created with the vector C0C1F0C1.

2. Given the input signals directly in six-valued algebra as C0C1F0C1, a TestPattern is created with the vector C0C1F0C1 and the previous and current patterns are implied to be 0111 and 0101, respectively.

.
Table 3.1 lists some of the variables defined in this class and their descriptions.

int type	The type of the algebra used.
CUDSignal[] vector	A one dimensional array of CUDSignals storing the signals to be applied to the inputs of the CUD.
Pattern currentPattern, previousPattern	A TestPattern has two patterns, the previous and the current. According to the type of algebra used, the signals are determined from the previous and\or the current pattern and stored in the array "vector".
CUDSignal[] faultFreeResult	An array of length equal to the number of outputs in the CUD, used to keep the fault-free simulation output signals.
CUDSignal[] actualResult	An array of length equal to the number of outputs in the CUD, used to keep the actual simulation output signals.
boolean[] isFailing	Marks passing and failing outputs in the testPattern. isFailing[i] is true if $output_i$ in the CUD has an erroneous result for this TestPattern.
ArrayList<CUDLine> potentialDefectLines	A list of suspect lines responsible for the testPattern failure.

Table 3.1: Some of the variables defined in the class TestPattern.

3.2.4 The Diagnosable Interface

This interface represents a contract defining the possible interactions between the backtracing based diagnosis algorithm and the CUD. In other words, it defines the methods a circuit should support so that it can be diagnosed by a backtracing based algorithm. The methods are shown in listing 3.1.

```
public interface Diagnosable {
  public void runFaultFreeSimulation(TestPattern tp);
  public void backTrace(TestPattern tp);
  public ArrayList<CUDLine> backTrace(int failingOutput);
  public void runSimulation(TestPattern tp);
  public void injectFaults(ArrayList<MultiLineFlip> injectedFaults);
}
```

Listing 3.1: The Diagnosable interface.

3.2.5 The CircuitUnderDiagnosis Class

As shown earlier in figure 3.1, the CircuitUnderDiagnosis class implements the Diagnosable interface and, hence, represents a circuit that can be diagnosed by backtracing based diagnosis algorithms. It uses the variable "public LeveledGraphSim leveledGraph" in order to store the CUD structure. The five methods inherited from the interface are implemented as follows:

Fault-free Simulation for a Test Pattern

Listing 3.2 shows the Java implementation of the method "public void runFaultFreeSimulation(TestPattern tp)". As proposed in [3], sensitive gate inputs were marked during fault-free simulation for later reference during the process of backtracing.

```java
public void runFaultFreeSimulation(TestPattern tp){
    //Start at level zero, input gates read input values from the test
    //pattern
    int gateCount=leveledGraph.graph[0].length;
    for(int i=0; i<gateCount; i++){
      LeveledGraphNode currentNode=leveledGraph.graph[0][i];
      currentNode.faultFreeValue=new CUDSignal(
      currentNode.propagate(tp.vector[i].getSignalValue(), 0));
    }
    //Iterate over the next levels
    //Every gate reads the input values from gates at previous levels
    for(int level=1; level<leveledGraph.graph.length; level++){
      gateCount=leveledGraph.graph[level].length;
      for(int i=0; i<gateCount; i++){
        LeveledGraphNode currentNode=leveledGraph.graph[level][i];
        int a=currentNode.inputs[0].faultFreeValue.getSignalValue();
        int b=0;
        if(currentNode.inputs.length>1)
        b=currentNode.inputs[1].faultFreeValue.getSignalValue();
        currentNode.faultFreeValue=new CUDSignal(
        currentNode.propagate(a, b));

        //Mark sensitive inputs
        int n=currentNode.inputs.length;
        currentNode.isSensitive=new boolean[n];
        currentNode.hasAtLeastOneSensitiveInput=false;
        switch(currentNode.type){
        case LeveledGraphNode.NODE_TYPE_INPUT: break;
        //Gates having only one input
        case LeveledGraphNode.NODE_TYPE_NOT:
        case LeveledGraphNode.NODE_TYPE_BUF:
        case LeveledGraphNode.NODE_TYPE_OUTPUT:
                currentNode.isSensitive[0]=true;
                currentNode.hasAtLeastOneSensitiveInput=true;break;
        //AND and NAND have dominant logic value 0
        case LeveledGraphNode.NODE_TYPE_AND:
        case LeveledGraphNode.NODE_TYPE_NAND:
          if(currentNode.inputs[0].faultFreeValue.getFinalValue()==0
          && currentNode.inputs[1].faultFreeValue.getFinalValue()==1){
            currentNode.isSensitive[0]=true;
            currentNode.hasAtLeastOneSensitiveInput=true;
          }
          if(currentNode.inputs[1].faultFreeValue.getFinalValue()==0
          && currentNode.inputs[0].faultFreeValue.getFinalValue()==1){
            currentNode.isSensitive[1]=true;
            currentNode.hasAtLeastOneSensitiveInput=true;
          }
          if(currentNode.inputs[0].faultFreeValue.getFinalValue()==1
          && currentNode.inputs[1].faultFreeValue.getFinalValue()==1){
            currentNode.isSensitive[0]=true;
            currentNode.isSensitive[1]=true;
            currentNode.hasAtLeastOneSensitiveInput=true;
          }
          break;
        //OR and NOR have dominant logic value 1
        case LeveledGraphNode.NODE_TYPE_OR:
        case LeveledGraphNode.NODE_TYPE_NOR:
          if(currentNode.inputs[0].faultFreeValue.getFinalValue()==1
          && currentNode.inputs[1].faultFreeValue.getFinalValue()==0){
            currentNode.hasAtLeastOneSensitiveInput=true;
            currentNode.isSensitive[0]=true;
          }
          if(currentNode.inputs[1].faultFreeValue.getFinalValue()==1
          && currentNode.inputs[0].faultFreeValue.getFinalValue()==0){
            currentNode.isSensitive[1]=true;
            currentNode.hasAtLeastOneSensitiveInput=true;
          }
          if(currentNode.inputs[0].faultFreeValue.getFinalValue()==0
          && currentNode.inputs[1].faultFreeValue.getFinalValue()==0){
            currentNode.isSensitive[0]=true;
            currentNode.isSensitive[1]=true;
            currentNode.hasAtLeastOneSensitiveInput=true;
          }
          break;
        //XOR and XNOR have no dominant logic value
        case LeveledGraphNode.NODE_TYPE_XOR:
        case LeveledGraphNode.NODE_TYPE_XNOR:
          currentNode.isSensitive[0]=true;
          currentNode.isSensitive[1]=true;
          currentNode.hasAtLeastOneSensitiveInput=true;
          break;
        }
      }
    }
    //Update the test pattern
    for(int i=0; i<leveledGraph.numOutputs(); i++)
      tp.faultFreeResult[i]=new CUDSignal(leveledGraph.graph
      [leveledGraph.graph.length-1][i].faultFreeValue.getSignalValue());
   }
```

Listing 3.2: Fault-free simulation for a test pattern.

Backtracing a Failing Test Pattern

Listing 3.3 shows the Java implementation of the method ”public void backTrace(TestPattern tp)”. The implementation is based on the idea of breadth-first search described in subsection 2.5.2.

```
public void backTrace(TestPattern tp) {
    //Initially all nodes are unvisited
    HashSet<Integer> visited=new HashSet<Integer>();
    LinkedList<LeveledGraphNode> queue=new LinkedList<LeveledGraphNode>();
    tp.potentialDefectLines=new ArrayList<CUDLine>();
    //Start BFS from the failing outputs of the given test pattern
    for(int i=0; i<tp.getOutputsCount(); i++)
      if(tp.isFailing[i]){
        LeveledGraphNode failingOutputNode=
        leveledGraph.graph[leveledGraph.graph.length-1][i];
        failingOutputNode.pre=null;
        queue.add(failingOutputNode);
      }
    while(!queue.isEmpty()){
      LeveledGraphNode current=queue.removeFirst();
      if(visited.contains(current.gate_id))continue;
      //Mark the node as visited
      visited.add(current.gate_id);
      //If it is not an output node, mark the CUD line as
      //a potential defect site
      if(current.pre!=null){
        CUDLine line=new CUDLine(current, current.pre,
        CUDSignal.complement(current.faultFreeValue.getSignalValue()));
        tp.potentialDefectLines.add(line);
      }

      //BFS terminates when reaching an input node
      if(current.type==LeveledGraphNode.NODE_TYPE_INPUT){
        continue;
      }
      else{
        int n=current.inputs.length;
        for(int i=0; i<n; i++){
          LeveledGraphNode neighbour=current.inputs[i];
          if(current.hasAtLeastOneSensitiveInput&& !current.isSensitive[i])
          continue;
          neighbour.pre=current;
          queue.addLast(neighbour);
        }
      }
    }
  }
```

Listing 3.3: Backtracing a failing test pattern.

Backtracing a Failing Output

Backtracing a failing output can be considered as a special case of the general case of backtracing a failing test pattern with one or more failing outputs. Hence, the implementation of the method ”public ArrayList<CUDLine> backTrace(int failingOutput)” is also based on the idea of BFS with the difference that exactly one node (the failing output) is initially added to the queue.

Injecting a List of Faults

Injecting a given list of faults is simply implemented by calling the ADAMA method leveledGraph.addHook(f) for every fault f. More about fault injection in ADAMA was explained earlier in subsection 2.6.3.

Simulation for a Test Pattern

Implementing the method ”public void runSimulation(TestPattern tp)” was mainly done in two steps:

1. Run the simulation of the current test pattern using ADAMA’s method: leveledGraph.sim(tp.currentPattern). As mentioned earlier in subsection 2.6.4, the responses of the current pattern are then updated in-place.
2. Updating *tp*’s actualResult and isFailing arrays.

3.2.6 The DiagnosisReport Class

The DiagnosisReport class was added to provide a report of the final list of suspect lines and their assigned fault models (if any) in addition to some important statistics about the diagnosis process such as the actual injected faults, the number of test patterns used, the number of failing test patterns and the number of actual defect lines successfully identified by the diagnosis algorithm.

3.2.7 The DiagnosisAlgorithm Class

The DiagnosisAlgorithm class is an abstract class representing a generic backtracing based diagnosis algorithm and providing a partial implementation for the common tools used in backtracing based diagnosis algorithms. Each specific diagnosis algorithm is then created as a subclass of this class and completes its own functionality by providing the implementation to the abstract methods inherited. Listing 3.4 shows a list of these abstract methods.

```java
public abstract void run(CircuitUnderDiagnosis circuit);
public abstract void debug(CircuitUnderDiagnosis circuit, TestPattern tp);
public abstract void prepareFaults();
public abstract Point experiment(int repeat, CircuitUnderDiagnosis
circuit);
```

Listing 3.4: Abstract methods in the DiagnosisAlgorithm class.

Some of the main variables defined in the class DiagnosisAlgorithm and inherited by its subclasses are shown in figure 3.2.

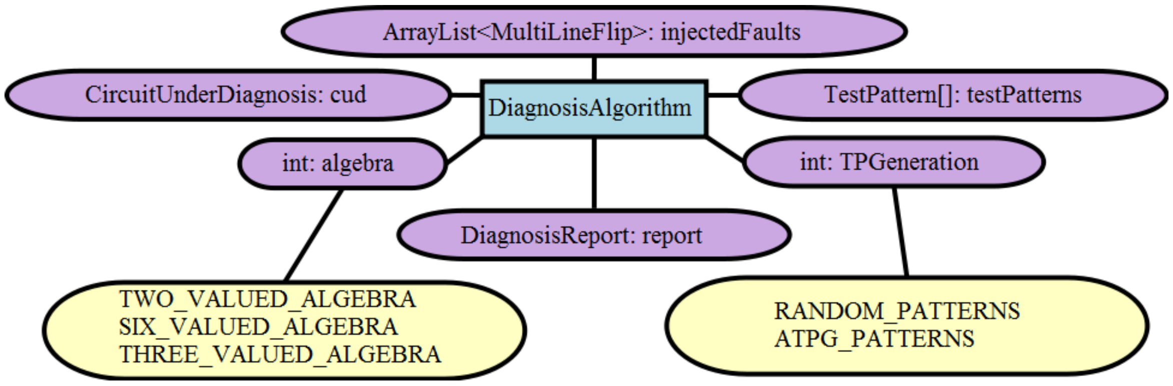


Figure 3.2: Main variables in the DiagnosisAlgorithm class.

Chapter 4

Implementation of Two Backtracing Algorithms

This chapter presents how the two different backtracing based diagnosis algorithms: DERRIC and DSIE, were implemented on top of the extended framework.
Two new classes were added to implement the two algorithms: DERRICAlgorithm and DSIEAlgorithm. Both classes extend the abstract class DiagnosisAlgorithm described in subsection 3.2.7. Being both non-abstract classes, they (must) provide their own implementation of the inherited abstract methods. Referring to the description of DERRIC in subsection 2.4.1, the description of DSIE in algorithm 1 and the functionality already provided in the extended framework, one can notice that a lot of the implementation work is already supported by the extended framework such as: Backtracing, fault-free simulation and signal assignment and propagation. Nevertheless, each algorithm class needs to define a number of auxiliary variables and methods to support its algorithm-specific heuristic(s) as described in sections 4.1 and 4.2.
Finally, the implementation of the method ”run” (inherited from DiagnosisAlgorithm) in each class basically calls the required newly added subroutines and existing extended framework subroutines in order to carry out the diagnosis process on a given circuit and prepare a diagnosis report.

4.1 DERRIC

The algorithm-specific heuristics in DERRIC are the intersection procedure and fault model allocation.
The intersection procedure required the following addition in the CUDSignal class:

public static int intersectionTable[][] This two dimensional array stores the intersection results between different CUD signals as defined in table 2.2.

Furthermore, fault model allocation required the following addition in the CUDLine class:

public static String[][] faultModel This two dimensional array stores the names of the fault models associated with the different CUDSignal values as defined in table 2.3.

The DERRICAlgorithm class also defined a new variable: ArrayList<CUDLine> suspectLines. This list represents the result of the intersection between the lists obtained via CPT.

4.2 DSIE

The algorithm-specific heuristic in DSIE is conservative implication and site elimination. Translating the implication and assignment rules into code, they can be interpreted as a group of subroutines that trigger one another as follows:

- Start by triggering a number of implications from the passing outputs (assigned their fault-free values) according to the implication rule.
- Every triggered implication triggers a number of assignments in the neighboring nodes according to the assignment rule.
- Every node n assigned to a known value v triggers (if applicable):
  - Eliminating the site (n, v') as a potential defect site.
  - A number of new implications according to the implication rule.

Throughout the following explanation, each iteration of the inner for loop in algorithm 1 will be referred to as an implication round.
The DSIEAlgorithm class defined three new variables:

HashSet<LeveledGraphNode> implicationVisited a set used to mark nodes as visited during a certain implication round.

LinkedList<LeveledGraphNode> q a queue used in the implication process

HashSet<Point> suspectLineSet a set gathering all potential defect sites. Each potential defect site is uniquely identified by its CUDLine ID and CUDLine signal value. Hence, a site is represented by a unique point (pair).

Moreover, the following list shows some of the new auxiliary methods defined:

public boolean eliminateSite(CUDLine s) If suspectLineSet contains s, it is removed and true is returned (a successful elimination). Returns false otherwise.

public boolean notAPotentialDefectSiteForAnyFailingPattern(LeveledGraphNode n) Returns false if suspectLineSet contains s, true otherwise.

public void performForwardImplication(LeveledGraphNode node) Performs forward implication from the given node. If a new node is assigned a known value, it gets enqueued to the implication queue q.

public boolean performBackwardImplication(LeveledGraphNode node) Performs backward implication from the given node. If a new node is assigned a known value, it gets enqueued to the implication queue q. If the backward implication lead to a successful site elimination, it returns true. Returns false otherwise.

public boolean performConservativeImplicationAndSiteElimination(TestPattern tp)

Code shown in listing 4.1.

```java
public boolean performConservativeImplicationAndSiteElimination
(TestPattern tp){
    boolean siteEliminated=false;
    LeveledGraphNode[] outputs=cud.leveledGraph.getOutputs();
    q=new LinkedList<LeveledGraphNode>();
    //Enqueue all passing outputs
    for(int i=0; i<outputs.length; i++){
      if(!tp.isFailing[i]){
        q.add(outputs[i]);
      }
    }
    while(!q.isEmpty()){
      LeveledGraphNode c=q.removeFirst();
      if(implicationVisited.contains(c))continue;
      //Mark c as visited in the current implication round
      implicationVisited.add(c);
      if(notAPotentialDefectSiteForAnyFailingPattern(c)){
        siteEliminated |= performBackwardImplication(c);
      }
      performForwardImplication(c);
    }
    return siteEliminated;
  }
```

Listing 4.1: Conservative implication and site elimination method in the DSIEAlgorithm class.

Chapter 5

Graphical User Interface

A simple GUI was developed for the extended framework. The purpose of such an addition was to help the user visualize how a circuit is internally stored in ADAMA as a graph. Moreover, it provides a user-friendly interface for loading circuits and running diagnosis algorithms.
The GUI was implemented using Java's Swing and AWT packages. This chapter presents the implementation details of the GUI as well as the possible interactions it provides.

5.1 The GateButton Class

The GateButton class was created to represent a node in the CUD graph as a button. It extends the predefined JButton class. Two variables are defined in this class:

LeveledGraphNode node The node represented in the button.

Color color The background color of the button.

5.2 The AdamaGUI Class

The AdamaGUI class was created to display the GUI frame for the user. It extends the predefined JFrame class and contains the different controls.
The following subsections explain the different interaction between the GUI frame and the user in three different modes.

5.2.1 Loading a Circuit

At the beginning, a JFileChooser is displayed for the user to load the CUD. The file chooser's directory is set to the current project directory. From this directory, .lg files are filtered and displayed for the user to choose from. Figure 5.1 shows a screen shot of the file chooser.

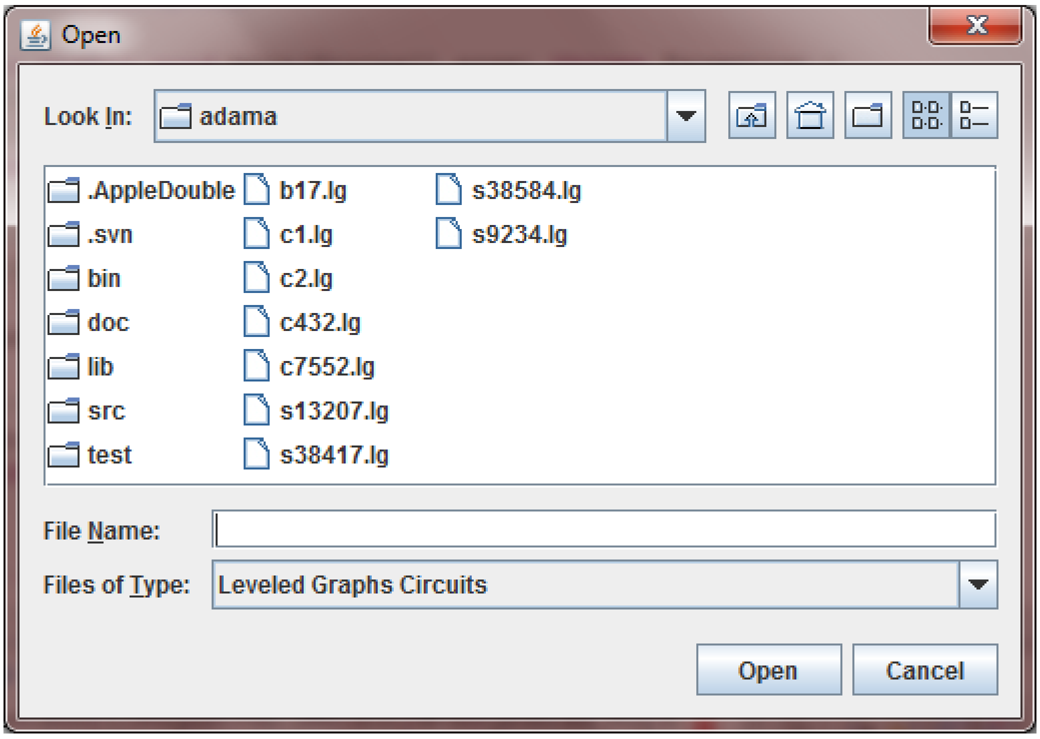


Figure 5.1: Loading a circuit

5.2.2 The Menu Bar

The menu bar holds the following menus:

Algorithm displays a list of the different available diagnosis algorithms a user can run on the CUD.

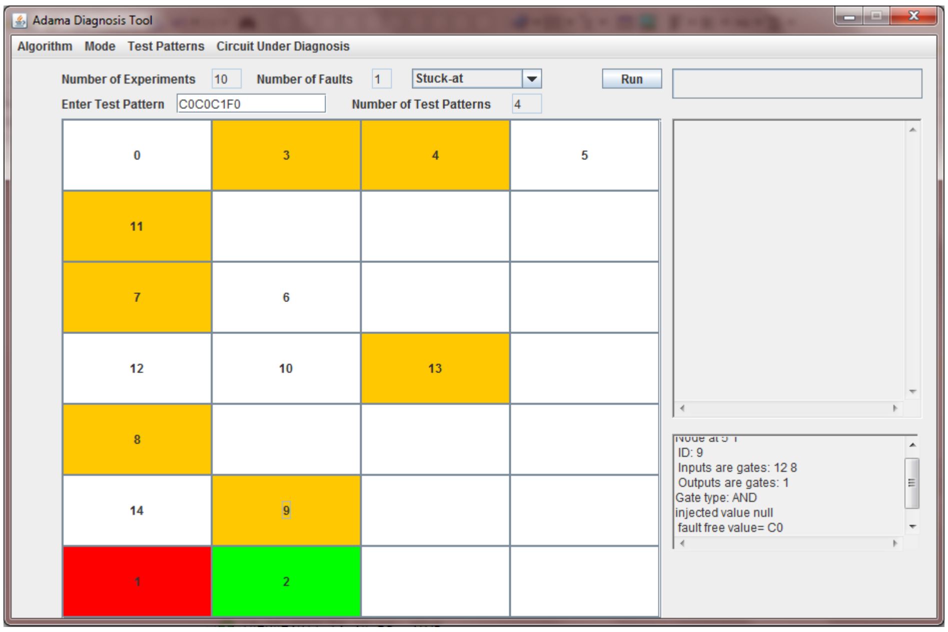
Mode displays a list of the different modes of running the diagnosis process. A diagnosis process can run in: The debugging mode, the diagnosis mode or the experimenting mode.

Test Patterns displays a list of the different test pattens available: Random test patterns or ATPG test patterns.

Circuit Under Diagnosis the only item in this menu is "Load Circuit…" which displays the file chooser again to allow the user to load a different CUD.

5.2.3 The Debugging Mode

The debugging mode is an interactive mode in which the nodes of the CUD are displayed as an array of GateButton components. This mode allows the user to enter a specific test pattern to be applied to the CUD inputs. The pattern is simulated and failing output nodes of the CUD are colored in red while passing output nodes are colored in green. Backtracing is then performed and potential defect nodes are colored in yellow. Figure 5.2 shows a screen shot of the frame in the debugging mode. In the example shown in the figure, the test pattern was set to C0C0C1F0. One of the output nodes failed while the other passed. Seven nodes were marked as potential defect sites.

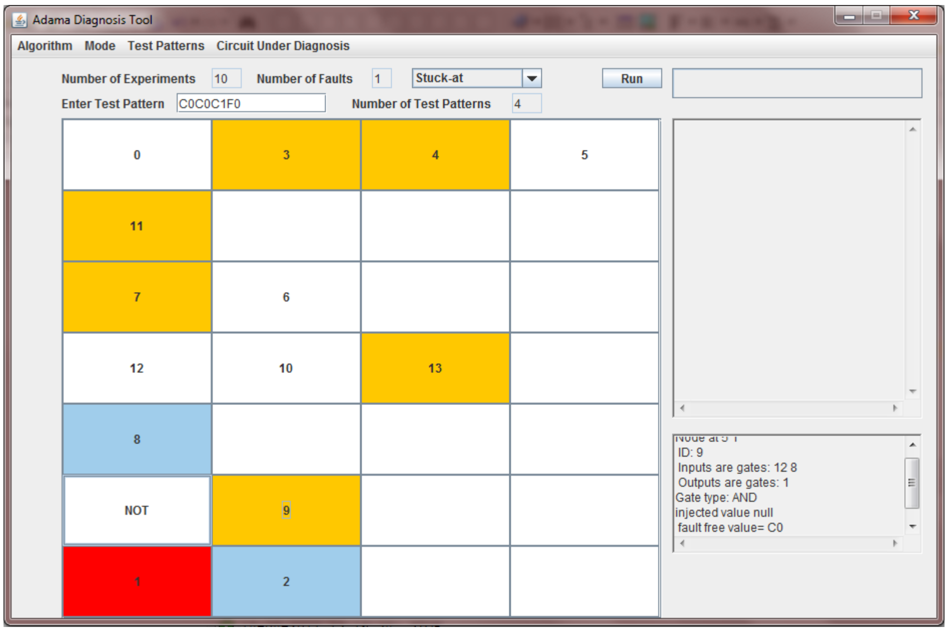


Number of Experiments 10
Number of Faults 1
Stuck-at
Enter Test Pattern C0C0C1F0
Number of Test Patterns 4

ID: 9
Inputs are gates: 12 8
Outputs are gates: 1
Gate type: AND
injected value null
fault free value= C0

Figure 5.2: Debugging mode.

When the mouse cursor enters a GateButton, its text is set to its node type and its input and output nodes are highlighted in blue as shown in figure 5.3.



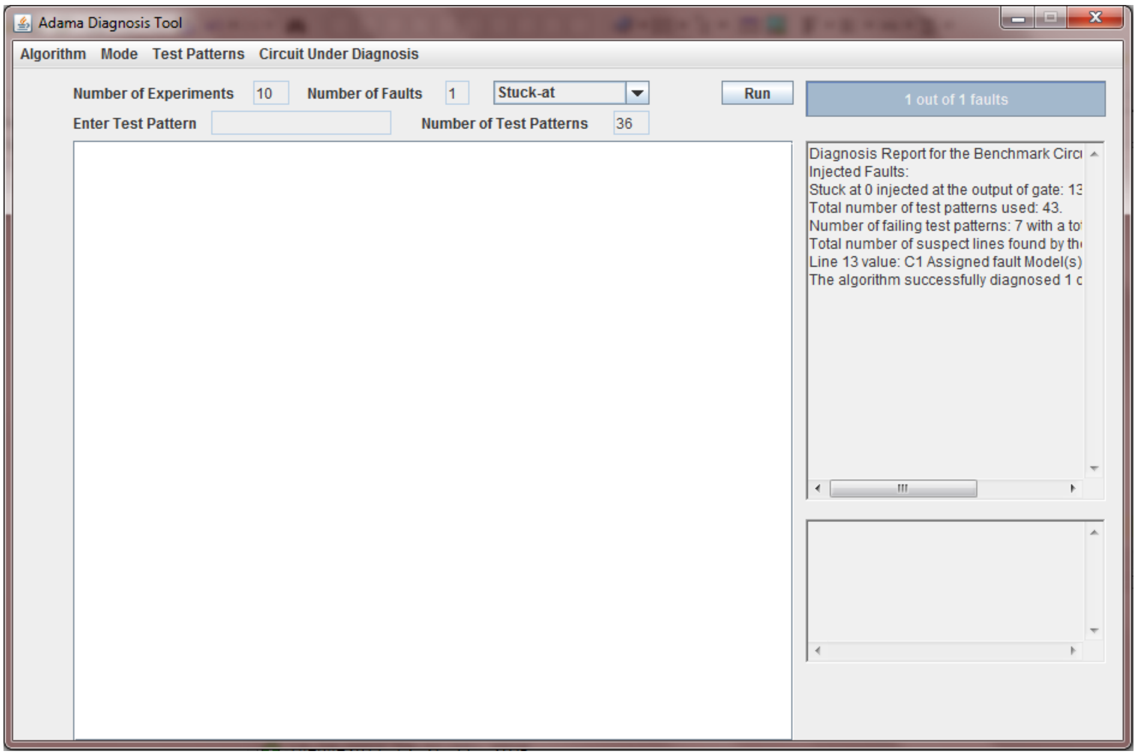
Number of Experiments 10
Number of Faults 1
Stuck-at
Enter Test Pattern C0C0C1F0
Number of Test Patterns 4

ID: 9
Inputs are gates: 12 8
Outputs are gates: 1
Gate type: AND
injected value null
fault free value= C0

Figure 5.3: Mouse entered on a NOT node. Input and output nodes are highlighted in blue.

5.2.4 The Diagnosis Mode

The diagnosis mode runs the selected diagnosis algorithm steps completely using the specified test patterns type and generates a diagnosis report. The diagnosis report is displayed and the progress bar is updated according to the success of the diagnosis process. Figure 5.4 shows an updated progress bar and a diagnosis report displayed after DERRIC was run in the diagnosis mode.



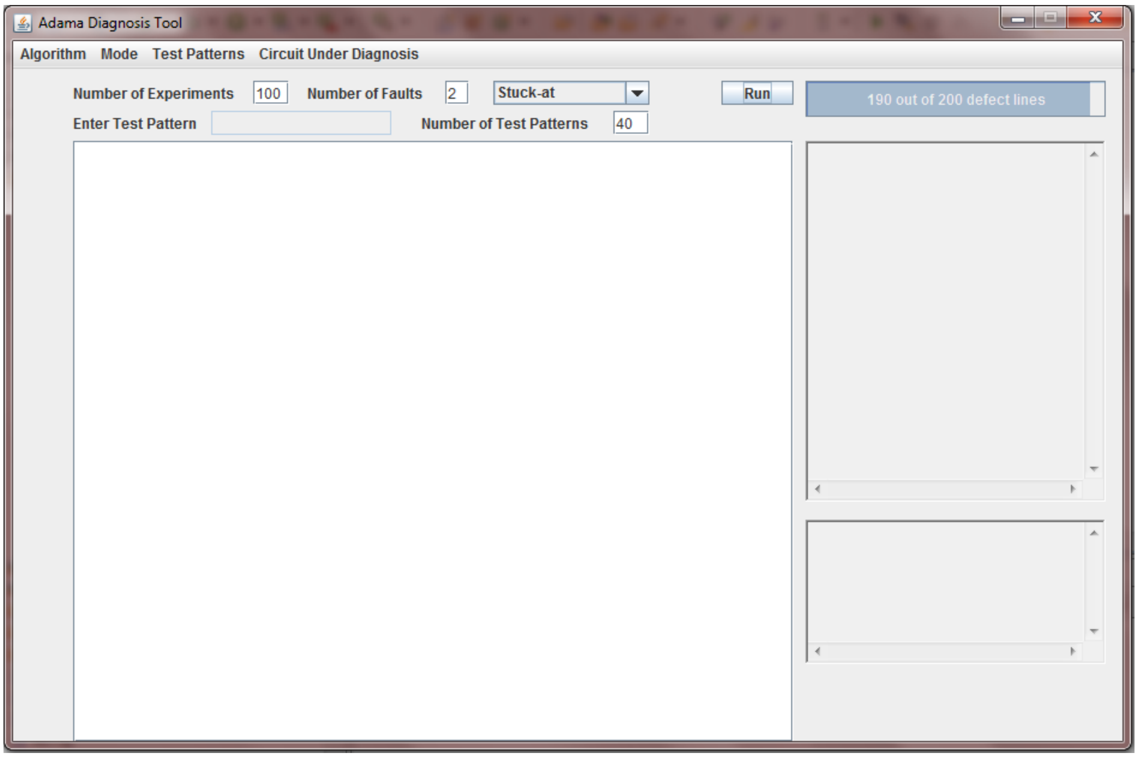
Number of Experiments 10
Number of Faults 1
Stuck-at
Enter Test Pattern
Number of Test Patterns 36

Diagnosis Report for the Benchmark Circı
Injected Faults:
Stuck at 0 injected at the output of gate: 13
Total number of test patterns used: 43.
Number of failing test patterns: 7 with a to
Total number of suspect lines found by th
Line 13 value: C1 Assigned fault Model(s)
The algorithm successfully diagnosed 1 c

Figure 5.4: The diagnosis mode.

5.2.5 The Experimenting Mode

This mode helps the user evaluate a certain algorithm by running a number of experiments. The number of experiments, the type of test patterns, the number of random faults to be injected in each experiment and their types are specified by the user. The progress bar is then updated according to the success of the experiments. Figure 5.5 shows an example screen shot of running DSIE in the experimenting mode. The number of experiments was specified to be 100 experiments. In each experiment, two stuck-at faults were randomly inject in the CUD and 40 random test patterns were generated to diagnose the CUD. Th progress bar shows that 190 defect lines were successfully diagnosed out of 200 actual defect lines.



Number of Experiments 100
Number of Faults 2
Stuck-at
190 out of 200 defect lines
Enter Test Pattern
Number of Test Patterns 40

Figure 5.5: The experimenting mode.

Chapter 6

Experimental Evaluation

In this chapter, the experiments performed to evaluate the two backtracing based diagnosis algorithms, DERRIC and DSIE, are described in detail along with their results.

6.1 Benchmark Circuits

In order to test and evaluate the implemented algorithms, a number of ISCAS'85, ITC'99 and ISCAS'89 benchmark circuits of varying sizes was used. Each benchmark circuit was first converted to a leveled graph using ADAMA's import task. In this step, complex gates are dissolved, flip-flops are replaced by PPI and PPO, constants are propagated, dangling logic is removed and the circuit is stored as a leveled graph of primitive logic gates [8]. Table 6.1 lists the benchmark circuits used and the number of gates (nodes) in their leveled graphs.

Family	Name	Total number of gates
ISCAS'85	c432	259
ISCAS'85	c7552	4358
ITC'99	b17	38513
ISCAS'89	s13207	10158
ISCAS'89	s38584	24656

Table 6.1: List of the benchmark circuits used.

6.2 Test Patterns

Some of the experiments were carried out using ATPG test patterns while others were carried out using both ATPG and random test patterns in order to compare the performance of each algorithm in both cases.
The random test patterns were generated by making a call to the method prepareRandomTestPatterns in the DiagnosisAlgorithm class, which in turn makes a call to the static method randomList in ADAMA's PatternBlockList class.
The ATPG test patterns were generated using a commercial ATPG tool. They were then stored in .p files using ADAMA's pimport task. Finally a call to the constructor

new PatternBlockList(”patternsFileName.p”) is made in order to construct a new list of pattern blocks read from the .p file.

6.3 Evaluation Measures

The performance of the implemented diagnosis algorithms was evaluated in terms of execution time, diagnosability and resolution.

Diagnosability is defined as D/I, where D is the number of actual defect sites identified by the diagnosis algorithm and I is the number of injected defect sites [13].

Resolution is defined as I/R, where I is the number of injected defect sites and R is the total number of potential defect sites reported by the diagnosis algorithm [13].

The higher the diagnosability and resolution, the better the diagnosis algorithm is [13].

6.4 Fault Injection

Each experiment specified the number of injected faults. The type of injected faults was partially specified by the experiment to be either a stuck-at fault or a bridging fault. A stuck-at fault is then randomly chosen to be either stuck-at-0 or stuck-at-1. Similarly, a bridging fault is randomly chosen to be either a two-line AND bridging fault or a two-line OR bridging fault. Nodes, where faults were injected, were randomly chosen such that no single node could have more than one injected fault.

6.5 Experimental Results

The experimental results of DERRIC and DSIE are presented in subsections 6.5.1 and 6.5.2, respectively.

6.5.1 DERRIC

DERRIC was evaluated using c432, c7552 and b17. Each experiment was repeated 100 times for c432 and c7552 and 50 times for b17 and results were averaged. Due to the fact that the diagnosis algorithms need failing test patterns to start the diagnosis process, experiments in which none of the test patterns failed were not considered in calculating the average results. Tables 6.2, 6.3 and 6.4 report the following results for every evaluation: The faults injected, the test patterns used (TP), the average number of failing test patterns (FP), the average diagnosability, the average resolution and the average execution time of the algorithm.
DERRIC achieved 100% stuck-at fault diagnosability for all three circuits. The maximum achieved bridging fault diagnosability was found to be 54%, 30% and 66% for c432, c7552 and b17, respectively. This diagnosability drop can be explained by the fact that the injected bridging faults affected two lines in the CUD. Thus, the single fault assumption employed in DERRIC's intersection procedure could have eliminated an actual fault location from the suspect list.

The execution time increased from smaller circuit to larger circuits. The first reason for this increase is that the average execution time per test pattern grows linearly with the size of the circuit. The second reason is a bigger number of test patterns is required for diagnosing larger circuits.

Faults	TP	FP (avg)	Diagnosability (avg)	Resolution (avg)	Time (avg)
1 SA	43 (ATPG)	7.15	1	0.16	1.62 ms
1 SA	40 (Random)	5.97	1	0.14	1.49 ms
1 SA	50 (Random)	6.7	1	0.18	1.66 ms
1 SA	60 (Random)	10.01	1	0.13	2.11 ms
1 SA	70 (Random)	10.10	1	0.16	2.34 ms
1 SA	80 (Random)	15.06	1	0.2	2.78 ms
1 BR	43 (ATPG)	7.32	0.46	0.22	1.43 ms
1 BR	40 (Random)	5.3	0.54	0.19	1.47 ms
1 BR	50 (Random)	7.64	0.5	0.19	1.74 ms
1 BR	60 (Random)	7.85	0.42	0.21	2.06 ms
1 BR	70 (Random)	9.84	0.36	0.25	2.27 ms
1 BR	80 (Random)	10.98	0.38	0.26	2.57 ms

Table 6.2: DERRIC evaluation on c432. Results averaged over 100 repetitions.

Faults	TP	FP (avg)	Diagnosability (avg)	Resolution (avg)	Time (avg)
1 SA	100 (ATPG)	22.01	1	0.18	38.55 ms
1 SA	100 (Random)	19.88	1	0.14	39.5 ms
1 SA	130 (Random)	27.02	1	0.17	52.91 ms
1 SA	160 (Random)	33.63	1	0.13	62.81 ms
1 SA	190 (Random)	45.1	1	0.15	76.06 ms
1 SA	220 (Random)	46.3	1	0.17	85.13 ms
1 BR	100 (ATPG)	22.03	0.12	0.02	37.88 ms
1 BR	100 (Random)	22.9	0.3	0.09	39.02 ms
1 BR	130 (Random)	26.18	0.21	0.08	50.61 ms
1 BR	160 (Random)	30.62	0.26	0.07	62.07 ms
1 BR	190 (Random)	40.84	0.28	0.09	73.53 ms
1 BR	220 (Random)	36.38	0.21	0.07	84.6 ms

Table 6.3: DERRIC evaluation on c7552. Results averaged over 100 repetitions.

Faults	TP	FP (avg)	Diagnosability (avg)	Resolution (avg)	Time (avg)
1 SA	1036 (ATPG)	123.6	1	0.08	5.24 s
1 SA	1000 (Random)	278.78	1	0.16	4.92 s
1 BR	1036 (ATPG)	157.46	0.3	0.09	4.75 s
1 BR	1000 (Random)	115.94	0.66	0.15	4.86 s

Table 6.4: DERRIC evaluation on b17. Results averaged over 50 repetitions.

6.5.2 DSIE

DSIE was evaluated using c7552, s13207 and s38584. Each experiment was repeated 100 times for c7552 and 50 times for s13207 and s38584 and results were averaged. As mentioned in subsection 6.5.1, experiments in which none of the test patterns failed were not considered in calculating the average results. Tables 6.2, 6.3 and 6.4 report the following results for every evaluation: The faults injected, the test patterns used (TP), the average number of failing test patterns (FP), the average diagnosability, the average resolution and the average execution time of the algorithm.

Faults	TP	FP (avg)	Diagnosability (avg)	Resolution (avg)	Time (avg)
1 SA	101 (ATPG)	22.09	1	0.012	96.9 ms
1 BR	101 (ATPG)	22.17	0.93	0.01	98.73 ms
2 SA	101 (ATPG)	41.39	0.93	0.005	160.71 ms
2 BR	101 (ATPG)	39.13	0.79	0.007	153.87 ms
4 SA	101 (ATPG)	62.59	0.9	0.0142	225.45 ms
4 BR	101 (ATPG)	62.08	0.71	0.0128	222.09 ms
8 SA	101 (ATPG)	86.17	0.9	0.005	272.17 ms
8 BR	101 (ATPG)	85.47	0.62	0.011	298.25 ms

Table 6.5: DSIE evaluation on c7552. Results averaged over 100 repetitions.

Faults	TP	FP (avg)	Diagnosability (avg)	Resolution (avg)	Time (avg)
1 SA	266 (ATPG)	86.14	1	0.1	0.938 s
1 BR	266 (ATPG)	73.02	0.95	0.1	0.847 s
2 SA	266 (ATPG)	158.2	0.88	0.1	1.633 s
2 BR	266 (ATPG)	163.8	0.11	0.17	1.899 s
4 SA	266 (ATPG)	224.4	0.83	0.09	2.211 s
4 BR	266 (ATPG)	214.3	0.07	0.16	2.407 s
8 SA	266 (ATPG)	255.66	0.82	0.06	2.612 s
8 BR	266 (ATPG)	254.26	0.1	0.2	2.961 s

Table 6.6: DSIE evaluation on s13207. Results averaged over 50 repetitions.

Faults	TP	FP (avg)	Diagnosability (avg)	Resolution (avg)	Time (avg)
1 SA	150 (ATPG)	44.9	1	0.11	2.205 s
1 BR	150 (ATPG)	48.36	0.93	0.09	2.359 s
2 SA	150 (ATPG)	78.36	0.84	0.08	3.546 s
2 BR	150 (ATPG)	68.8	0.37	0.17	3.32 s
4 SA	150 (ATPG)	110.08	0.73	0.14	4.881 s
4 BR	150 (ATPG)	113.86	0.15	0.26	5.255 s
8 SA	150 (ATPG)	140.06	0.71	0.11	6.355 s
8 BR	150 (ATPG)	138.38	0.11	0.23	7.044 s

Table 6.7: DSIE evaluation on s38584. Results averaged over 50 repetitions.

For all three circuits, the average single stuck-at fault diagnosability achieved by DSIE was 100%. The average multiple stuck-at fault diagnosability was also reasonably good (>70%) for up to 8 stuck-at faults. Multiple bridging fault diagnosability scores, however, were lower for the bigger circuits: S13207 and s38584. Applying another elimination algorithm after DSIE that allows ranking the list of potential defect (such as path-based site-elimination [13]) can help improve the diagnosability and resolution scores. In [13], only the top 1.5×(no. of injected fault locations) potential defect sites are reported. This way, the resolution was controlled to be >0.67 and the diagnosability was better.
The average execution time per failing test pattern varied from 4.39 ms to 0.051 s, according to the size of the CUD, the number of injected faults and their type.

In comparison with DERRIC, both algorithms achieved the same (100%) single stuck-at fault diagnosability on c7552, using the same ATPG test patterns. However, they achieved different single stuck-at fault resolution and different single bridging fault diagnosability and resolution.
DSIE achieved much better single bridging fault diagnosability (93%) than DERRIC (28%). The resolution achieved by DERRIC was better than DSIE for both single stuck-at diagnosis (0.18 versus 0.012) and single bridging fault diagnosis (0.02 versus 0.01). Although the single fault assumption used in DERRIC caused the low diagnosability score in case of a bridging fault, it helped DERRIC eliminate more potential defect sites, achieving better resolution scores.

Chapter 7

Conclusion and Future Work

Backtracing based diagnosis algorithms depend on a backtracing procedure from the outputs of the CUD toward the inputs in order to locate faults. The goal of this project was developing a powerful tool for implementing and evaluating different backtracing based diagnosis algorithms.
The approach followed to reach this goal was extending an existing Java-based diagnosis framework: ADAMA. Hence, two backtracing based diagnosis algorithms: DERRIC [11] and DSIE [13], were studied in depth. DERRIC was developed for single fault diagnosis while DSIE targets multiple fault diagnosis. A list of common properties, tools and functions was extracted from the two algorithms in order to represent the common functionality among backtracing based diagnosis algorithms. This common functionality was then implemented and integrated with ADAMA. Moreover, a simple GUI was developed to provide the extended framework with a user-friendly interface.
In order to demonstrate the powerfulness of the extended framework, DERRIC and DSIE were implemented on top of it. The extended framework did facilitate their implementation by providing many of the tools and subroutines required. It then remained for each algorithm to implement its own heuristics. With the completed implementation of the two algorithms, a number of experiments was carried out on benchmark circuits. The extended framework made it possible to evaluate each algorithm in terms of diagnosability, resolution and execution time.

7.1 Future Work

Simulating delay faults will be a valuable addition to the extended framework. If implemented, DERRIC and other delay fault diagnosing algorithms can be evaluated using delay faults as well.
Moreover, the user interface currently provided by the extended framework was only programmed to handle a special expected user behavior. Hence, defensive programming is recommended to provide a more robust user interface and a diagnosis tool of a better quality.

Bibliography

[1] M. Abramovici and M.A. Breuer. Multiple fault diagnosis in combinational circuits based on an effect-cause analysis. *IEEE Transactions on Computers*, C-29(6):451–460, June 1980.

[2] M. Abramovici, M.A. Breuer, and A.D. Friedman. *Digital systems testing and testable design*. Electrical engineering, communications, and signal processing. IEEE Press, 1990.

[3] M. Abramovici, P.R. Menon, and D.T. Miller. Critical path tracing - an alternative to fault simulation. In *Proc. 20th Design Automation Conference (DAC'83)*, pages 214–220, June 1983.

[4] T. Bartenstein, D. Heaberlin, L. Huisman, and D. Sliwinski. Diagnosing combinational logic designs using the single location at-a-time (SLAT) paradigm. In *Proc. International Test Conference (ITC'01)*, pages 287–296, 2001.

[5] M.L. Bushnell and V.D. Agrawal. *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*. Frontiers in electronic testing. Kluwer Academic, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[7] P. Girard, C. Landrault, and S. Pravossoudovitch. Delay-fault diagnosis by critical-path tracing. *IEEE Design Test of Computers*, 9(4):27–32, December 1992.

[8] S. Holst. How to get things done with ADAMA. Open Seminar, June 2011.

[9] Y.-C. Lin, F. Lu, and K.-T. Cheng. Multiple-fault diagnosis based on adaptive diagnostic test pattern generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(5):932–942, May 2007.

[10] M.M. Mano. *Digital design*. Prentice Hall, 2003.

[11] A. Rousset, A. Bosio, P. Girard, C. Landrault, S. Pravossoudovitch, and A. Virazel. DERRIC: A tool for unified logic diagnosis. In *12th IEEE European Test Symposium (ETS'07)*, pages 13–20, May 2007.

[12] S. Venkataraman and W.K. Fuchs. A deductive technique for diagnosis of bridging faults. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD'97)*, pages 562–567, November 1997.

[13] X. Yu and R.D. Blanton. Multiple defect diagnosis using no assumptions on failing pattern characteristics. In *Proc. 45th ACM/IEEE Design Automation Conference (DAC'08)*, pages 361–366, June 2008.

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Maha Samir Badreldein

Maha Samir Badreldein
31 August, 2011

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Maha Samir Badreldein
31 August, 2011