Hierarchical Test Generation Based on Alternative Graph Models

A Master Thesis

Submitted to the Computer Engineering and Diagnostics

Department of the Institute of Computer Engineering

In fulfillment of the requirements for the

Degree of

Master of Science

Computer Engineering

---

by

Jaan Raik

Tallinn Technical University

April 1997

Table of Contents

1 INTRODUCTION 1

2 OVERVIEW OF PREVIOUS RELATED WORKS 2

3 ALTERNATIVE GRAPHS 5

3.1 BASIC DEFINITIONS 5
3.2 SIMULATION ON ALTERNATIVE GRAPH MODELS 6
3.3 GENERAL FAULT MODEL FOR ALTERNATIVE GRAPHS 7

4 REPRESENTING DIGITAL DEVICES WITH AGS 8

4.1 BEHAVIORAL LEVEL DESCRIPTIONS 8
4.2 REGISTER-TRANSFER LEVEL DESCRIPTIONS 10
4.3 BOOLEAN ALTERNATIVE GRAPHS 14

5 HIERARCHICAL TEST GENERATION WITH ALTERNATIVE GRAPHS 16

5.1 OVERVIEW OF THE TEST GENERATION SYSTEM 16
5.2 HIGH-LEVEL PATH ACTIVATION ALGORITHM 18
5.3 GENERATING SYMBOLIC TESTS FOR AGS 19
5.4 PROPAGATION AND JUSTIFICATION ON THE AG MODEL 23
5.5 TRANSPARENCY RULES AND IMPLICATIONS 24
5.6 STATE TRANSITIONS 28
5.7 PATH ACTIVATION CONSTRAINTS 31
5.8 INTERACTION BETWEEN HIGH- AND LOW-LEVEL PARTS 36

6 EXPERIMENTAL RESULTS 38

7 CONCLUSIONS 39

BIBLIOGRAPHY 41

List of Figures

Figure 1. Graphical Representation of an Alternative Graph 6
Figure 2. Example of a VHDL process 9
Figure 3. AG for a Process Variable 10
Figure 4. A Datapath Fragment 11
Figure 5. AG Generation for Datapath 11
Figure 6. Linking AG Instances 12
Figure 7. State Table Generation from VHDL 13
Figure 8. Control Part AG Generation 14
Figure 9. Alternative Graphs for a Combinational Circuit 15
Figure 10. The Design Cycle 17
Figure 11. Conformity Test 21
Figure 12. Scanning Test 22
Figure 13. Generating AGs from Rule Tables 27
Figure 14. State Transitions during Setup 29
Figure 15. State Transitions during Propagation 30
Figure 16. State Transitions during Justification 31
Figure 17. Flow Graph of Algorithm 32
Figure 18. Path Activation on Flow Graph of Algorithm 33
Figure 19. The State Sequence during Path Activation 33
Figure 20. Constraint Extraction Example 34
Figure 21. Interaction between High- and Low-Level Parts 36

List of Tables

Table 1. Test Generation Results..... 38

1 Introduction

Test generation for real-life digital circuits on the gate-level is an extremely complex task. It has been showed that test generation for combinational circuits is an NP-complete problem [Fujiwara 85]. However, algorithms have been developed which can handle test generation for relatively large combinational circuits in a reasonable computing time [Goel 81, Fujiwara 83]. Gate-level test generation for sequential circuits (i.e. circuits containing memory components) is even more complex and remains still an unsolved problem in practice.

During recent times, as a possible solution, hierarchical test generation methods have evolved [Lee and Patel 91] which take advantage of higher abstraction level (e.g. behavioral or register-transfer (RT) level) information while generating tests for the gate-level faults. Hierarchical test generation is based on the divide-and-conquer principle. Device under test is considered on different design abstraction levels, and test generation on these levels is performed by applying an appropriate test generation tool. In hierarchical testing, top-down and bottom-up strategies are known. In the bottom-up approach, tests generated at the low level will later be assembled at the higher abstraction level.

Current thesis considers a top-down approach [Krupnova and Ubar 94], where constraints extracted at the higher level (register-transfer level) are taken into account when deriving tests for the lower level (gate level). The hierarchical test generation approach is based on the multiple-level alternative graph (AG) representations. The main advantage of using AG models lies in the fact that AGs allow application of common procedures and uniform methodology throughout different design abstraction levels.

The thesis is organized as follows. Section 2 gives an overview of previously presented approaches in the field of high level testing. In Section 3, basic definitions of alternative graphs are given. In addition, general system modelling, simulation and fault diagnosis on AGs is described. Section 4 covers the representation of digital systems with AG models on different design abstraction levels. Section 5 explains different aspects of the AG-based hierarchical test generation algorithm. In Section 6, some preliminary experimental results achieved on prototype software are presented.

Section 7 is for final conclusions. In addition, Appendix A gives the description of general AG model file format syntax.

2 Overview of Previous Related Works

In current section, an overview of the state-of-the-art in the field of testing digital systems is given. The main features and problems of present hierarchical and high level testing methods are pointed out. The following sections introduce a test generation approach based on the theory of alternative graphs, which provides for a possible solution to overcome many of the disadvantages stated below.

Test generation methods can be divided into structural and functional approaches. At the structural level, a digital system is described by components and their interconnections. Complexity problems in test generation have forced to replace the "sea of interconnected gates" by more tractable networks of higher level components - macros or higher level primitives. Using truth tables or state tables to describe the macros and to develop fault models on their basis [Sridhar 79] will be possible only for not very complex modules. Attempts to develop special functional fault models for different components like decoders, multiplexers, memory blocks, PLAs, microprocessors [Abraham 86] lead to a lot of different test generation strategies.

The development of test generation algorithms that try to benefit from hierarchical circuit representation have been continuously on the forefront of current research [Sridhar 79, Somenzi 85, Chandra 87, Murray 88, Ghosh 92, Min 93]. An approach to exploit inherent hierarchy in complex combinational circuits was developed in [Chandra 87], high level primitives were introduced in [Sarfert 89, Calhoun 89]. An idea to high level test generation without the need for gate level structure was proposed in [Sridhar 79], where it was assumed that the stimulus/response package for each high level primitive would be predefined. The approach is not very consistent in dealing with path reconvergencies, and sequential primitives cannot be accepted either.

The use of high level primitives (macros) with pre-computed tests is a very attractive concept. It is based on creating a test program for a whole system by using

stimuli sets together with descriptions of the good behavior for each macro individually. However, this concept presents a problem: how can pre-computed tests for modules be assembled into a test program for the whole system? The simplest way to solve the assembly problem is to have a direct access to every macro input/output during test. The examples are Macro-Testing and Boundary-Scan [Bleeker 93]. The general way to assemble pre-computed tests is to carry out hierarchical multi-level test generation [Murray 88, Anirudhan 89, Leenstra 90]. In this case, the macro tests are calculated bottom-up by doing symbolic test generation on each level of hierarchy. Such test generation algorithms typically ignore the incompleteness problem: constraints imposed by other macros and/or the macro network structure may prevent all test vectors from being assembled. Top-down test analysis techniques can help to solve such problems by deriving environmental constraints for lower level modules. However, such techniques are not very effective in the phases where the system is still under development in a bottom-up fashion. In [Leenstra 90], a new incremental technique for the hierarchical assembly of macro tests into a complete system test program was developed. In contrast to other test generation approaches, the test pattern generation process is carried out here concurrently with the design of the system by assembling the test programs of the lower level parts. To prevent the overwhelming complexity of assembling each pattern individually, symbolic test generation is applied, thereby constructing a symbolic test program for each level. In [Lee 90], the most attention is devoted to the problem of assembling macro tests. The problem of generating tests for macros, especially for the control part of macros, has not been thoroughly handled.

In architectural level test generation approaches, digital systems are, in general case, divided into control and data portions [Anirudhan 89, Leenstra 90, Lee 92, Steensma 93]. This type of decomposition seems to be natural because of substantial differences in functionality and faults in these partitions. As a consequence, it also leads to the use of different models and tools in test generation process. In [Anirudhan 89], a hierarchical model of data path and a finite state model of the control part is used. In [Leenstra 90], for control unit the Petri-net model is used, whereas data part is represented by a network of macro blocks. In [Lee 92], the test generation process for data path is operated at a high level and gate level ATPG algorithms are used for the control unit. Such mixed approaches to different partitions of the same object during

the test generation have some disadvantages. They do not allow carrying out fault collapsing between control and data parts of a system and they do not allow using uniform tools for fault simulation and test generation for the respective parts.

In functional (behavioral) test generation approaches, only the behavior of the system will be exploited as the data about the unit under test. The architectural and structural implementation of the devices is not known. The works published earlier mostly concentrate on using hardware description languages [Lin 85, Shen 88], different types of system graphs [Thatte 80, Saucier 84], Petri nets [Lee 92, Santucci 93] binary decision diagrams (BDD) [Akers 78]. During recent period, a lot of attention is devoted for generating tests directly from descriptions in high level languages [Ward 90]. Functional test methods can be, in general, divided into identification and distinction testing. Identification test consists of the determination and verification of the functionality of the system, i.e. verifying if a system does what it is supposed to do [Akers 78, Lin 85, Saucier 84]. In general, it is impossible to estimate the quality of identification tests because of the lack of assumptions about faults or a fault model. Distinction test is based on using a fault model [Thatte 80, Ward 90, Corno 95]. Due to the lack of information about the structure and implementation of the unit under test, functional test generation does not guarantee a very good correspondence with real physical defects.

Current thesis presents a hierarchical test generation approach working on architectural (register-transfer) and gate levels. The approach uses multi-level alternative graph (AG) [Ubar 76, Ubar 96a] descriptions for design modelling. The main advantage of AG models lies in the fact that a uniform concept can be applied on different system abstraction levels. In addition, alternative graph models provide a powerful means for solving different diagnostic tasks.

3 Alternative Graphs

3.1 Basic Definitions

Definition: Alternative Graph (AG) [Ubar 76, Ubar 96a] can be defined as a directed non-cyclic labelled graph in the form of a quadruple $G=(V,A,Z,D)$, where V is a finite set of vertices (referred to as *nodes*), A is a finite set of arcs (*branches*), Z is a function which defines the *variables labelling the nodes* and the variable domains, and D is a function on A.

The function $Z(v_i)$ returns a pair (z_i,Z_i), where z_i is the variable letter which is labelling node v_i and Z_i is the domain of z_i. Each node of an AG is labelled by a variable. A single variable can label multiple nodes. In special cases of AGs, nodes can be labelled by constants or algebraic expressions.

A branch $a \in A$ of an AG is an ordered pair $a=(v_1,v_2) \in V^2$, where V^2 is the set of all the possible ordered pairs in set V. Graphical interpretation of a is a branch leading from node v_1 to node v_2. It is said that v_1 is a *predecessor node* of v_2, and v_2 is a *successor node* of the node v_1, respectively.

D is a function from A representing the activating conditions of the branches for the simulating procedures. The value of $D(a)$ is a subset of Z_i, where $a=(v_i,v_j)$ and $Z(v_i)=(z_i,Z_i)$. It is required that $Pv_i=\{D(a) \mid a=(v_i,v_j) \in A\}$ is a partition of the set Z_i. In other words, the subsets of the set Z_i labelled on the branches starting from a node v_i must not overlap and their union must be equal to Z_i.

AG has only one *starting node* (root), for which there are no preceding nodes. The nodes, for which successor nodes are missing, are referred to as *terminal nodes*.

Example: Figure 1 presents an example of a graphical interpretation of an AG:

G=(V,A,Z,D),

V=$\{v_1, v_2, v_3, v_4, v_5\}$,

A=$\{a_1, a_2, a_3, a_4, a_5\}$, $a_1=(v_1, v_2)$, $a_2=(v_1, v_4)$, $a_3=(v_1, v_5)$, $a_4=(v_2, v_3)$, $a_5=(v_2, v_4)$,

$Z(v_1)=Z(v_5)=(z_2,\{0,1,2,\ldots,7\})$, $Z(v_2)=(z_3,\{0,1,2,3\})$, $Z(v_3)=(z_4,?)$, $Z(v_4)=(z_1,?)$,

$D(a_1)=\{0\}$, $D(a_2)=\{1,2,3\}$, $D(a_3)=\{4,5,6,7\}$, $D(a_4)=\{2\}$, $D(a_5)=\{0,1,3\}$.

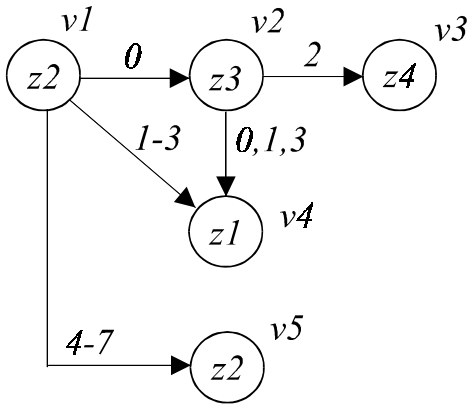

Figure 1. Graphical Representation of an Alternative Graph

Alternative Graphs were originally introduced and proposed for diagnostic purposes by R.Ubar in 1976 [Ubar 76]. Binary Decision Diagrams (BDDs) that were presented later by S.B.Akers [Akers 78] are in fact a special case of AGs. In other words, the concept of AGs is more general than the concept of BDDs. Systems on different abstraction levels can be modelled with AGs [Ubar 96]. Therefore, AGs provide for the application of common methodologies on different system abstraction levels and are suitable for hierarchical modelling.

3.2 Simulation on Alternative Graph Models

Consider a situation where all the node variables are fixed to some value. According to these values, for each non-terminal node a certain output branch will be chosen which enters into its corresponding successor node. Let us call such connections between nodes *activated branches* under the given values. Succeeding each other, activated branches form in turn *activated paths*. For each combination of values of all the node variables there exists always a corresponding activated path from the starting node to some terminal node. Let us call this path the *main activated path*.

From the previous definition it follows that an AG represents a functional relationship: for each combination of values for all the node variables there exists one and only one value which is equal to the value of the variable labelling the terminal

node of the main activated path. This relationship describes a mapping from a Cartesian product of the domains for variables in all the nodes to the joint set of values for variables in the terminal ones. Therefore, by AGs it is possible to represent arbitrary functions $Y = F(x)$, where Y is the variable whose value will be calculated on the AG and x is the vector of all variables which belong to the labels of the nodes in the AG.

When representing systems and functions with alternative graph models, in general case, a system of AGs rather than a single AG is required. During the simulation in AG systems, the values of some variables labelling the nodes of an alternative graph could be calculated by other alternative graphs.

Digital systems can be classified into different types and can be represented on different levels of hierarchy, e.g. behavioral (describing functionality), register-transfer (functional unit (FU) networks), gate or transistor levels. AGs, which describe digital systems at different levels, may have special interpretations, properties and characteristics. However, the same formalism and the same algorithms for test and diagnostic purposes can be used, which is the main advantage of using AGs. In the following section we will consider abstraction levels of digital systems and their respective representations by AGs.

3.3 General Fault Model for Alternative Graphs

Different fault models defined at different representation levels of digital systems are replaced on AGs by a uniform fault model [Ubar 1996b]:

1. The branch is always activated.
2. The branch is broken.
3. Instead of the given branch, another branch or a set of branches is activated.

Physical interpretation of faults associated with node outputs depends on the physical meaning of the node. Depending on the adequacy of representing the structure of the system, the fault model proposed can cover a wide class of structural and functional faults introduced for digital circuits and systems. Note that the stuck-at-value fault model used in the logic level can be treated as a special case of this general model.

AGs represent systems in form of questionnaires (decision diagrams) to explicitly reveal the cause-effect relationships among the system variables. By finding the main activated path of an AG for the given time moment (i.e. for the given vector of variable values), we determine the subset of nodes and, correspondingly, the subset of variables that are responsible of the system's behavior at the particular moment. In other words, only the nodes along the main activated path can cause the behavior of the system to be faulty. Hence, AGs as a way to represent concisely the diagnostic information of a system, provide a very powerful means for system diagnostics and modelling in general.

4 Representing Digital Devices with AGs

4.1 Behavioral Level Descriptions

On behavioral level, the behavior rather than the structure of a digital system is presented. Therefore, behavioral test generation can not assure a good correspondence with structural level faults unless the test generator has access to some additional information about how the structure will be implied or synthesized from the behavior. In current thesis, test generation on behavioral level will not be considered. However, a methodology for generating behavioral level AGs will be presented.

Alternative Graph generation from behavioral level hardware description languages (HDL) takes place as follows. In AG descriptions, for each HDL process variable and primary output a graph corresponds. In the graph, non-terminal nodes represent logical conditions, terminal nodes represent operations, and branches hold the subset of condition values by which the successor node corresponding to the branch will be chosen.

The VHDL example presented in Figure 2 explains the procedure of AG generation for a process variable. The AG corresponding to the process variable *cr* is presented in Fig.3. In the AG conditional HDL constructs are mapped into non-terminal nodes. Terminal nodes are formed of the right hand values of value assignments to the HDL variable under condition values corresponding to those that are marked on the respective branches. Quotation mark following a variable name denotes the previous value of the variable.

```
PROCESS
      VARIABLE ir       : bit_vector( 1 downto 0);
      VARIABLE cr       : bit_vector(15 downto 0);
      VARIABLE reg1     : bit_vector(15 downto 0);
      VARIABLE reg2     : bit_vector(15 downto 0);
      VARIABLE cond     : bit_vector(31 downto 0);
      ...
  BEGIN
      ...
      CASE ir IS
         WHEN "00" => cr := reg1;
                      ...
         WHEN "01" => cr := cr + reg2;

        WHEN "10" => ...
         WHEN "11" =>  IF (cond < 0) THEN
                         cr := cr + 1;
                       END IF;

       END CASE;
       cr  := cr + 1;
  END PROCESS;
```

Figure 2. Example of a VHDL process

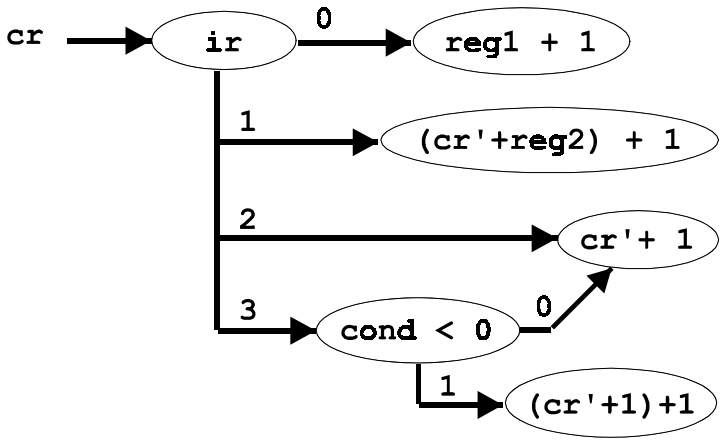

Figure 3. AG for a Process Variable

In similar way graphs will be generated for all the HDL variables and primary outputs of the design.

4.2 Register-Transfer Level Descriptions

During recent years, more-and-more commercial and non-commercial high-level synthesis tools have become available. These tools are used by designers to automatically generate register-transfer level (RTL) descriptions from behavioral descriptions of digital designs. In the RTL descriptions the design has been partitioned into a control part, i.e. a finite state machine (FSM), and a datapath part containing a network of interconnected *functional units* (FU). Usually the HLS tools take into account several constraints, as speed, area, or testability, and allow the designer to quickly compare the trade-offs between alternative RTL implementations.

Datapath is represented by a netlist of interconnected blocks. The building blocks of datapath are registers, multiplexers and functional units (i.e. blocks performing arithmetic and logic functions etc.). Figure 4 shows an example of a datapath fragment. Datapath can be described as a system of AGs, where for each primary output and each register an AG corresponds. In RTL AG models, the non-terminal nodes represent control signals coming from the control part and terminal nodes represent signals of the datapath, i.e. primary inputs, registers, operations, buses, constants.

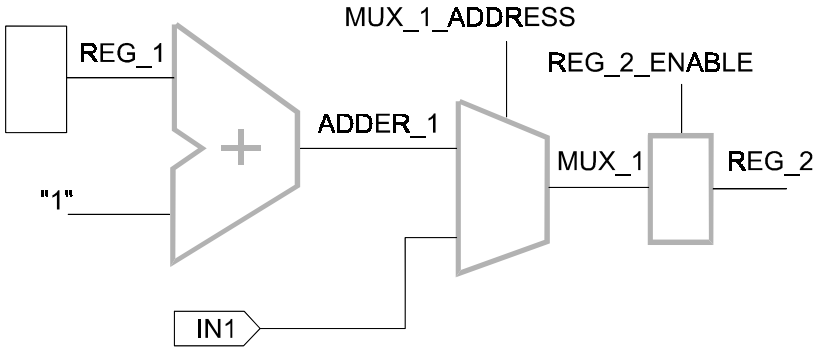

Figure 4. A Datapath Fragment

Figure 5 describes AG model generation from RTL VHDL descriptions [Krupnova 93] for the example given in Figure 4. In order to generate an AG model, a library containing AG descriptions for each type of datapath block must be implemented. In Figure 5, AG descriptions for adder, 2-input multiplexer and register library cells are shown in grey rectangles. The components of VHDL description are replaced by instances of AGs which are linked together by labelling AG nodes with corresponding VHDL signals.

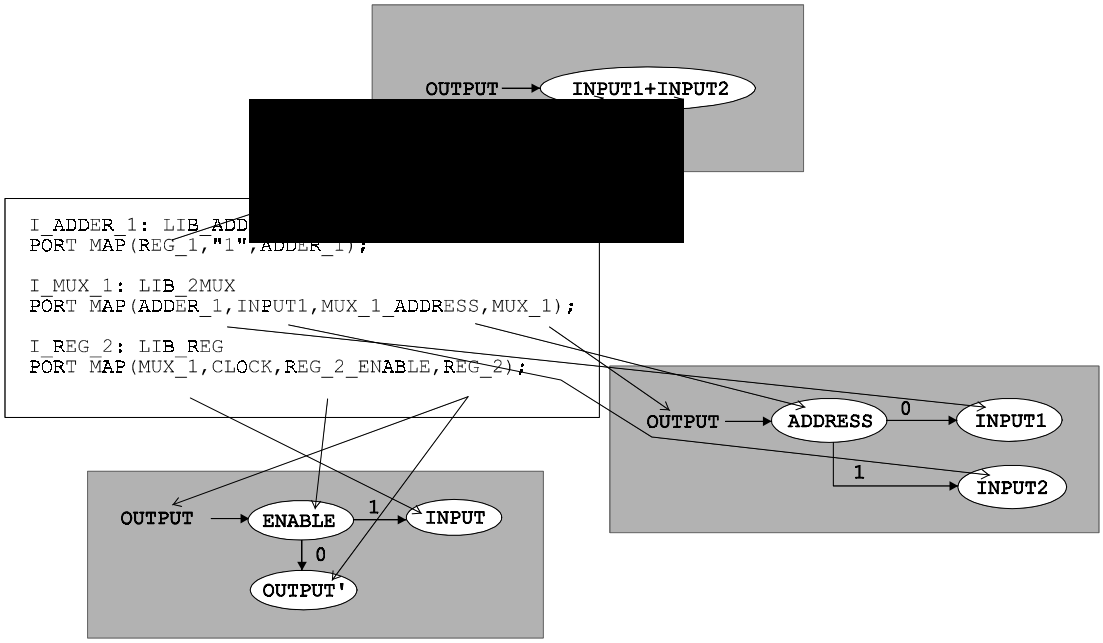

Figure 5. AG Generation for Datapath

Figure 6.a shows the instances of AG library components after labelling the nodes with signals from the VHDL description. In figure 6.b, the AG instances have been linked together into a single alternative graph. In similar way, AGs can be constructed for each datapath register and primary output.

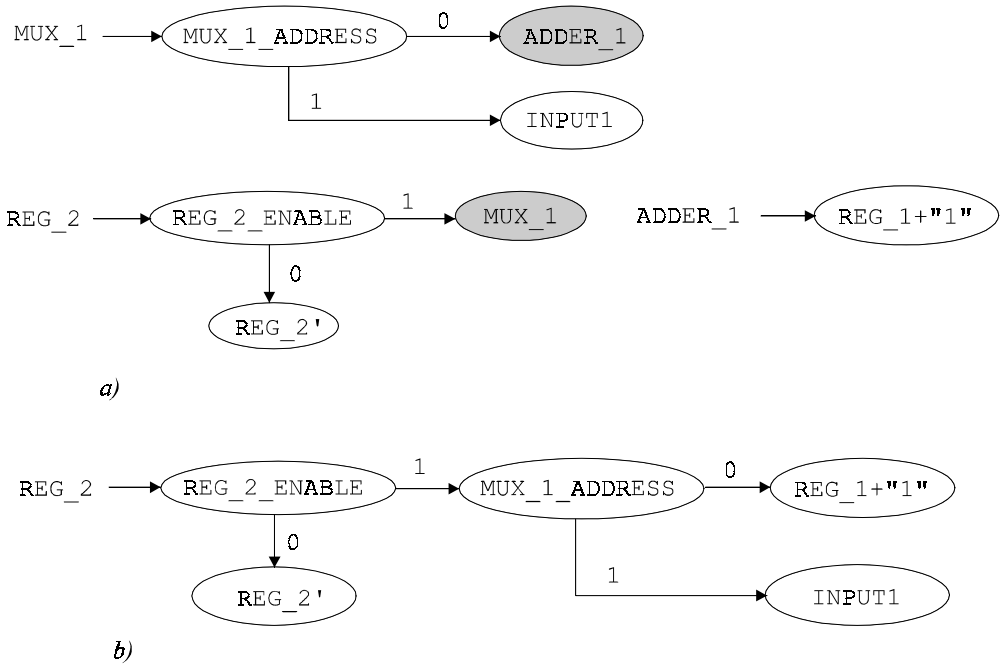

Figure 6. Linking AG Instances

Control part's finite state machine can be described by an AG which calculates the value for a vector of variables and whose terminal nodes are labelled by constant vectors [Krupnova 93]. Let us refer to this type of AGs as *vector AGs*. Non-terminal nodes of the control part's vector AG represent inputs for the control part (i.e. logical conditions) and current state, and terminal nodes represent the corresponding values of the next state and control signals leading into the datapath. In the following, AG generation for the control part will be explained.

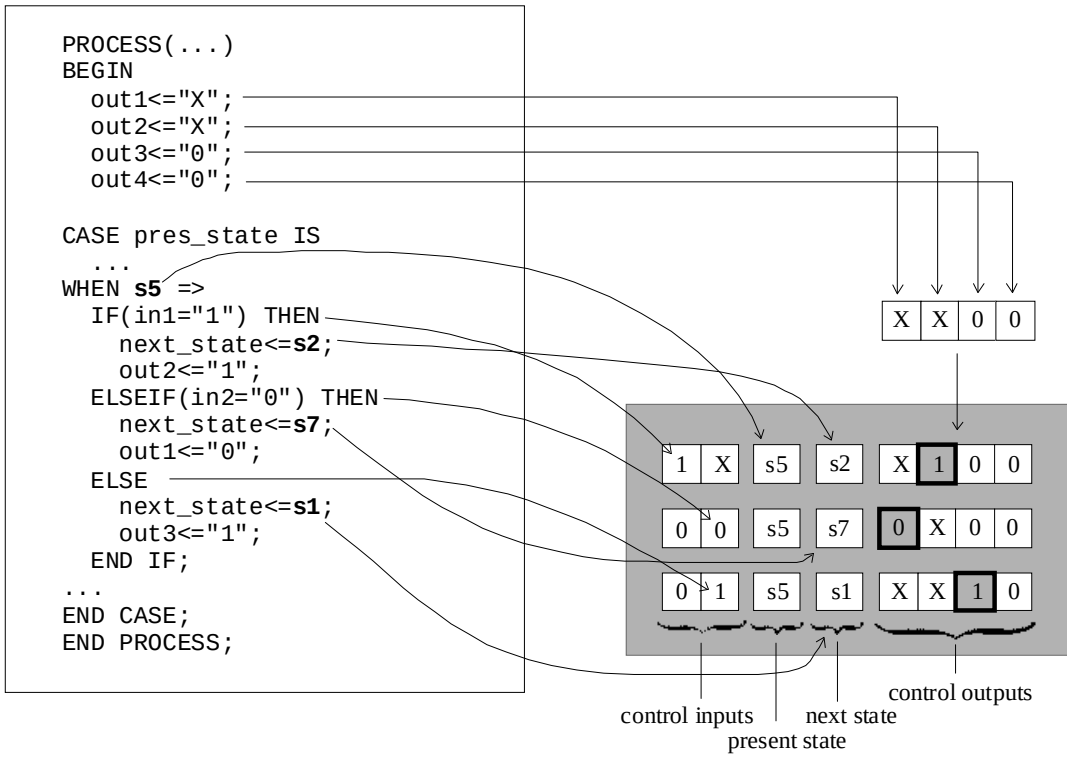

Figure 7. State Table Generation from VHDL

Figure 7 explains how FSM state tables can be generated from RTL VHDL descriptions. The VHDL example represents a partial next-state-logic description of a Mealy automaton for present state being equal to s5. In Figure 8, AG generation from the state machine is shown. The terminal nodes have been labelled by constant vectors, which will be assigned to the vector of variables corresponding to the graph if a path from the root node to the particular node is activated. The vector contains variables representing the state of FSM and the control signals leading to the datapath.

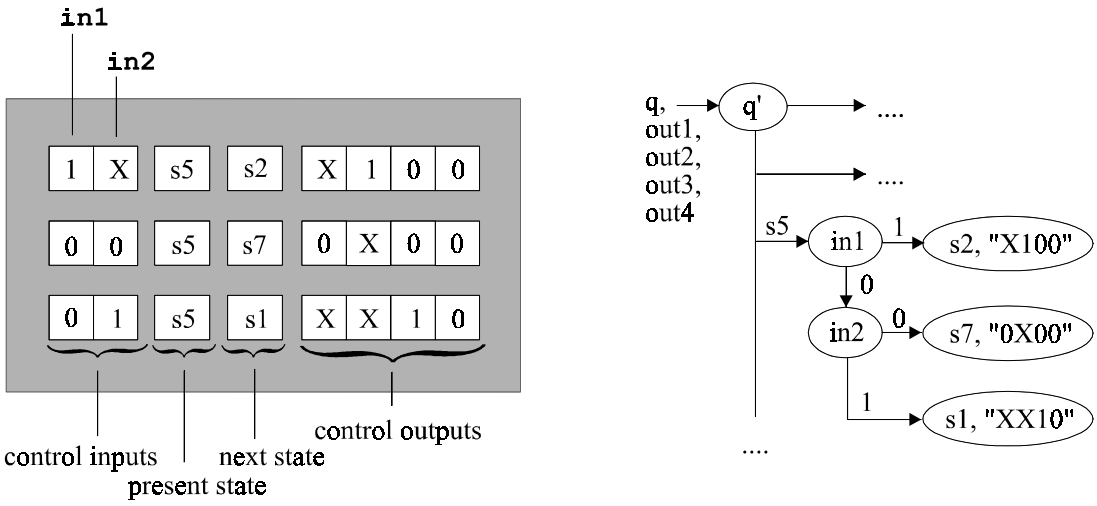

Figure 8. Control Part AG Generation

4.3 Boolean Alternative Graphs

Each output of a gate-level combinational circuit can be defined by a Boolean function, which can be represented as an AG. The nodes of this type of AG are labelled by Boolean variables and have consequently only two output branches. The terminal nodes are labelled by logical constants 0 and l.

We call this type of AGs *Boolean Alternative Graphs* (BAG). BAGs can also be further divided into *Structural AGs* (SAG) [Ubar 76] and Binary Decision Diagrams (BDDs). As an example, in Figure 9 two representations of a combinational circuit by AGs are given. For the sake of simplicity, the values of variables on branches are omitted. By convention, the right-hand branch corresponds to 1 and the lower-hand branch to 0. In addition, terminal nodes holding constants 0,1 are omitted. Exiting the AG rightwards corresponds to y =1, and exiting the AG downwards corresponds to y = 0.

Using the concept of SAGs, it is possible to rise from gate level descriptions to higher level structural descriptions without loosing the capability of representing gate-level structural faults. The task of simulating structural stuck-at faults in a given signal path of a tree-like subcircuit can be replaced by the task of simulating faults at the corresponding node of a SAG.

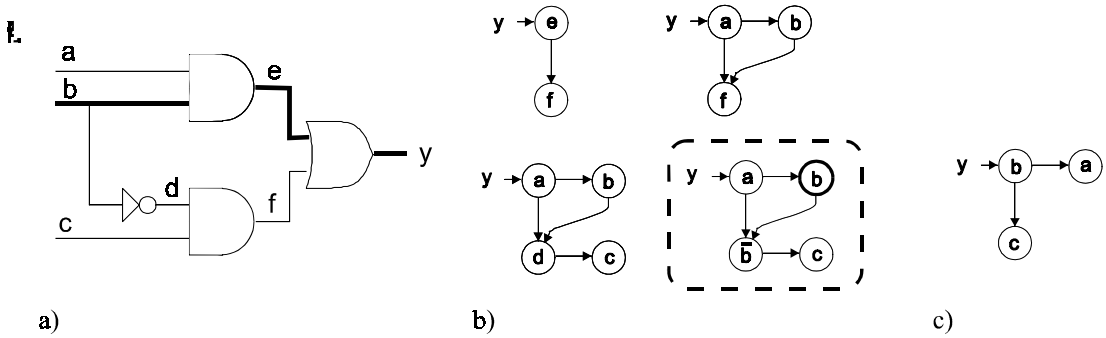

Figure 9. Alternative Graphs for a Combinational Circuit

An SAG corresponding to a circuit shown in Figure 9a is depicted in a dotted rectangle in Figure 9b. For each of the four paths in the circuit, one and only one node in the SAG corresponds. For example, the node denoted with a bold circle in Figure 9b represents the path marked with bold lines in Figure 9a.

SAG models for combinational circuits are generated as follows. The generation starts from a fanout point or a primary output. While moving from the starting point towards primary inputs, logic gates are substituted by respective elementary AGs for the gates. The procedure of superponing nodes by their respective AGs continues recursively and terminates in those nodes, which represent a primary input or a fanout branch. Figure 9b explains SAG generation for the primary output *y* of the circuit in Figure 9a. In similar way AGs are generated for each primary output and each fanout point in the circuit. SAG model for a combinational circuit is a system of AGs where for each fanout-free partition an AG corresponds.

Another way to generate logic level AGs lies in using implementation-free descriptions of digital devices (Boolean expressions, truth tables etc.). In this case, AGs would not differ from BDDs and we can use the methods developed for synthesis of BDDs [Akers 78]. Example of a BDD for the circuit in Figure 9a is shown in Figure 9c. A dynamic combination of BDDs and SAGs can contribute to test generation and fault simulation for large digital circuits. In general, BDDs afford more concise descriptions compared to SAGs, and therefore they can be used successfully for solving justification tasks or generating sensitivity conditions for fault propagation

between inputs and outputs in macro components considered as black boxes. The role of SAGs lies in representing implementation-dependent faults in given subcircuits.

5 Hierarchical Test Generation with Alternative Graphs

5.1 Overview of the Test Generation System

As the degree of integration in VLSI designs has been growing, so has the need for automation of different design tasks. Design automation helps to shorten the time-to-market cycle and improves significantly designer's productivity. The automation was first introduced on the lower levels of design tasks, like placement and routing, and together with the growth of design complexities, moved gradually to higher levels, e.g. logic synthesis, high-level synthesis (HLS) and hardware/software co-design. Nowadays the goal is to automate the entire design cycle from conceptualization to generation of silicon layout.

During recent years, more-and-more commercial and non-commercial high-level synthesis tools [Gajski 89] have become available. These tools are applied for automatically generating a register-transfer level (RTL) description from a behavioral description of the circuit. In the RTL descriptions the design has been partitioned into a control part, i.e. a finite state machine, and a datapath part containing a network of interconnected functional units (FU). Usually the HLS tools take into account several constraints, as speed, area, or testability [Flottes 95, Bhatia 94], and allow the designer to quickly compare the trade-offs between alternative RTL implementations.

With the appearance of high-level synthesis a number of automated test generation approaches were developed which take advantage of register-transfer level information while generating tests for gate-level faults. Current thesis presents a hierarchical alternative graph based test generation system, which takes as an input the RTL and gate-level descriptions of the design under test. Figure 10 shows the place of the system in the design cycle.

The hierarchical test generation environment consists of a hierarchical test generator and high-level and low-level AG model generators. From RT-level VHDL descriptions, high-level AG generator generates high-level AG models, which will be applied as input for the test generator for symbolic path activation. Structural

alternative graph models will be required when generating local, structural level tests for the functional units (FU) of the design.

The test generation process takes place in the following way. Tests are generated sequentially for each node in the high-level AG model of the datapath. Symbolic path activation on RT-level AG model is performed. During the path activation, functional constraints are extracted which are applied to a constraint satisfaction problem (CSP) solving algorithm [Montanari 74]. Subsequently, gate-level structural tests will be generated for the functional units (FU) and the local tests will be assembled with the symbolic tests to achieve the final test set for the node. In order to generate the gate-level tests, a random test generator working on structural AG-models is applied. Input data at the target module boundary will be obtained by applying random vectors to the inputs of the reduced constraints driven model and performing simulation on this model.

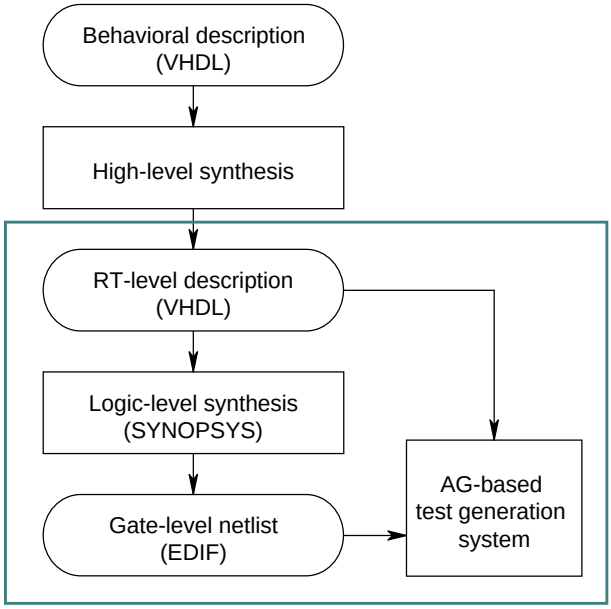

Figure 10. The Design Cycle

Due to the fact that the low-level test generator operates with structural AG (SAG) representations, low-level AG generator is required to generate SAG models from gate-level netlists. The low-level AG generator creates SAG representations from EDIF 2.0.0 netlist descriptions [EDIF 87]. EDIF is a technology-dependent design format. Therefore, appropriate technology libraries have to be included while performing EDIF to SAG conversions.

In order to test the FUs, gate-level models of the FUs must be synthesized. Current system uses Design Compiler by Synopsys Inc. for the logic-level synthesis. As an input for logic-level synthesis are the RT-level VHDL description and a VHDL library of FUs, containing generic bit-width behavioral descriptions of the FUs.

5.2 High-Level Path Activation Algorithm

In the following, a pseudo-programming language description of the general structure of high-level symbolic path activation is presented.

```
while exist graphs in the datapath AG model do
   while exist nodes in the graph do

      Set all the variables labelling the nodes to X in current timeframe;

      /* Setup objective: */

      if the node is a terminal node labelled by an operation then
         Set up the scanning test for the node;
      else if the node is a non-terminal node labelled by a control signal then
         Set up another conformity test for the node;
         // There are several tests per node!
      else
         Continue with the next iteration;
      end if;

      /* Propagation objective: */

      while fault effect F has not reached a primary output do
         Propagate;
      end while;

      /* Justification objective: */

      while exist unjustified variables do
         Justify;
      end while;

      /* Constraint satisfaction objective: */

      Satisfy the path activation constraints;

   end while;
end while;
```

The high-level path activation algorithm is a complete algorithm based on systematic search. In other words, it guarantees that a high-level path will be activated for any node under test when such a path exists. However, the methodology for providing the systematic search and backtracking is omitted from the above description. In order to fulfill the previously mentioned tasks, a decision tree (DT) is required. The DT used in current approach is a stack of nodes, where each node is a pair *(O,D),* where *O* refers to current objective and *D* refers to decision. *O* can be of one of the four types: *setup*, *propagation*, *justification*, or *constraint satisfaction*. Possible types for a decision *D* are: choosing the FSM states, choosing the terminal nodes for justification and conformity test, and choosing the graphs and terminal nodes for fault effect propagation.

When a backtrack occurs, then all the variable assignments that have been made subsequent to the previous decision are cancelled, i.e. the variables are set to the symbolic value X.

The procedures of conformity and scanning tests, propagation, justification, and satisfaction of path activation constraints are described in detail elsewhere in the thesis.

5.3 Generating Symbolic Tests for AGs

Current test generation algorithm uses the following symbolic values for variables:

- X – don’t care
- A known constant value
- S1 – the first symbol to be backtraced
- S2 – the second symbol to be backtraced
- F – the fault effect to be propagated

When the test generation starts, all the variables except constant variables will be set to X in current time frame. Subsequently, a respective symbolic test (i.e. conformity test or scanning test) will be performed to a node *n* in a graph *g*. The test will try to set up the symbolic value F to the variable calculating the value for the graph *g* and values S1 and S2 to two terminal nodes of *g*. During the high-level path

activation the fault effect F must be propagated to the primary outputs, and the symbols S1 and S2 backtraced to the primary inputs of the design.

Symbolic values S1 and S2 have different interpretations in different types of tests. In scanning test the symbolic values denote local test patterns for the FU corresponding to the operation labelling the node under test. However, in conformity test, symbolic values S1 and S2 are the values to be distinguished from each other.

Conformity test tests these nonterminal nodes of an AG, which are labelled by control signals (e.g. register enable signals, multiplexer control signals). This type of test is aimed at testing the multiplexers, decoders and control signals of the device. Conformity test on an AG takes place as follows. A non-terminal node *n* (the node under test) is selected and one of its output branches is activated. The values of the variables labelling all the other successor nodes to *n* have to be distinguished from the value of the variable labelling the node at the end of the activated branch. The distinguished values must be propagated to the terminal nodes of the graph and a path from the root node to *n* has to be activated in order to propagate the fault effect through the graph. This process will be repeated for each output branch and each pair of successor nodes. Thus, there are multiple conformity tests for each nonterminal node.

An example of conformity test is shown in Figure 11. Let us generate the test for the node labelled by variable B, when B=0. The algorithm is as follows.

1. Activate the path from the root node to the node labelled by B. (In our example A=1).
2. Find all the terminal nodes that can be reached if B=0. (Push the two nodes labelled by E and D to decision tree). Take the first node from the decision tree and assign the symbol S1 to the variable labelling it. (E=S1).
3. Find all the terminal nodes that can be reached if B=1. There is only one terminal node and it is labelled by E. It is not possible to assign S2 to E, because the value of E is already S1. Backtrack occurs and after repeating steps 2. and 3. The following result will be achieved as a symbolic conformity test when B=0.

   Y=F, A=1, B=0, C=1, D=S1, E=S2.

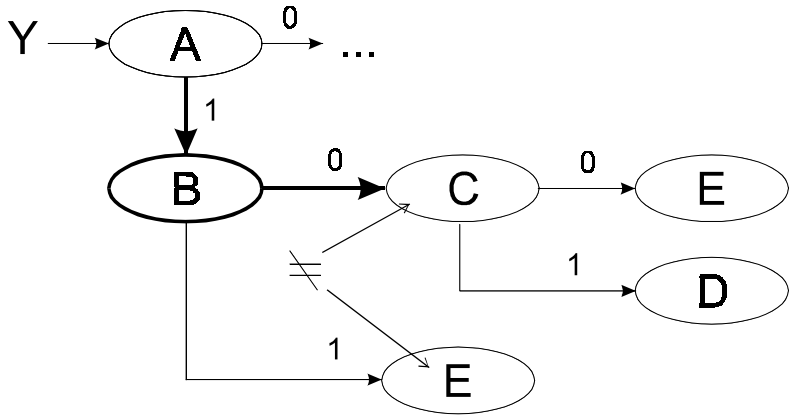

Figure 11. Conformity Test

In current approach, the conformity tests with distinguishing by 2 successor nodes only are considered. In general case, the number of such tests for a node would be n(n-1), where n is the number of successor nodes for the node under test. If we consider only the successors, which correspond to the branches whose labels differ by the Hamming distance equal to one then the number of conformity tests will be considerably lower. However, in the latter case the decoder part of the multiplexer control signals will not be tested.

In order to entirely test the multiplexer signals, the set of conformity tests for a nonterminal node must meet the following additional criterion. *Each bit of the variable values labelling the successor nodes of the node under test must be distinguished in at least one pair of successor nodes so that the value of the distinguished bit is zero in the variable labelling the node corresponding to the activated branch.*

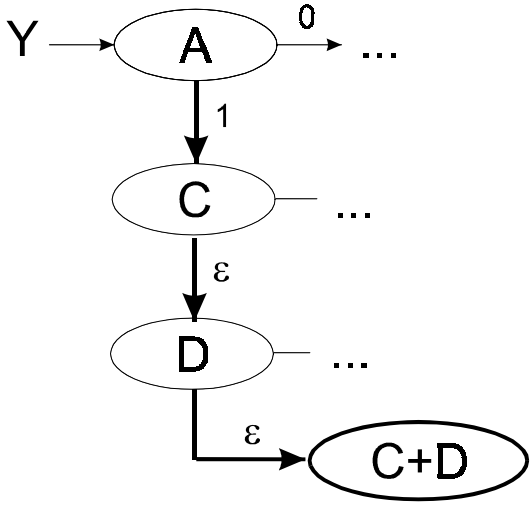

Figure 12. Scanning Test

Scanning test is applied to the terminal AG nodes, which are labelled by operations. The goal of scanning tests is to test the functional units of the device under test. Algorithm for scanning test is not as complex as the algorithm for conformity test, because during scanning test no decisions are made. In order to perform the scanning test for a node *n*, the path from the root node to *n* will be activated and the values of the variables corresponding to the operation arguments are set to S1 and S2, respectively.

The following example explains the generation of scanning test to the node labelled by operation C+D in Figure 12.

1. Activate the path from the root node to the node labelled by the operation. (In our example A=1). Note that activated branches labelled by ε will imply no uniquely defined values for the corresponding variables. The concept of using ε values is briefly explained in the Subsection “Transparency Rules and Implications”.
2. Assign symbolic values S1 and S2 to the corresponding inputs of the operation.

The scanning test for the addition operation in Figure 12 is as follows:

Y=F, A=1, C=S1, D=S2.

5.4 Propagation and Justification on the AG Model

Due to the fact that, in current approach, transparency rules and implications are reflected implicitly in the alternative graph model itself, justification and propagation procedures on that model are extremely simple. Previous high-level test generation approaches have used dedicated libraries for establishing transparent paths through functional units [Lee and Patel 91].

Let us consider a situation, where current objective is to propagate the fault effect from a variable *A* towards the primary outputs of the design. The steps required to fullfil the operations are as follows.

1. Choose a graph *Gy* calculating value for variable *Y* from all the graphs, where *A* labels at least one terminal node.
2. Choose from *Gy* a terminal node *T* labelled by *A*.
3. Activate the path from root to *T*.
4. If at least one of the values of variables $V_1,\ldots,V_n$ labelling the nodes of the activated path is inconsistent then backtrack.

If *Y* corresponds to a primary output then current objective will be justification. Otherwise current objective will be to propagate the fault effect from *Y* towards primary outputs.

During propagation, for all the nodes labelled by comparison operations, which are on the main activated path of *Gy*, corresponding stacks of conditional constraints are created.

In the following, the algorithm for justification on AG model is explained. Let us consider justification for variable *Y*, whose value is calculated by graph *Gy*.

1. Choose a terminal node *T* labelled by a simple variable (i.e. not an operation) or a constant whose value is consistent with the required value of *Y*.
2. Activate the path from root to *T*.
3. If at least one of the values of variables $V_1,\ldots,V_n$ labelling the nodes of the activated path is inconsistent then backtrack.

If *T* is labelled by a primary input then the justification will be complete. Otherwise current objective is justification for variable labelling the node *T*. If no variables to be justified exist then current objective will be to satisfy the path activation constraints.

In current algorithm, fault effect propagation along a single path is implemented, i.e. the D-frontier approach used by most of gate-level test generators [Roth 66, Goel 81, Fujiwara and Shimono 83] is neglected. The drawback of the propagation along a single path is that it ignores fault masking. On the other hand, fault masking on the RTL symbolic value level is of probabilistic nature and it is more complex to handle than on the gate-level. In addition, the single-path approach simplifies the algorithm significantly.

5.5 Transparency Rules and Implications

Transparency rules are rules for propagating values through an FU unchanged (referred to as I-path [Abadir and Breuer 85]) or in the way, where each change in the input value would reflect as a change in the propagated value (F-path [Freeman 88]). It is possible to additionally define less strictly transparent paths for propagating values through FUs, if neither I-paths nor F-Paths can be activated.

An implication is an additional information, which can be implemented to imply the value of the FU output from the value of an FU input.

Transparency rules and implications can be represented with tables, where each row represents a rule in form of the value of FU output depending on the value of an FU input. The following symbols are used:

- X - don’t care
- $Input_i$ - the value of the i-th input of the FU
- a constant value

Let us call the variables corresponding to FU inputs, whose value is symbol X in the current rule, *undetermined variables*. Otherwise the FU input variables are referred to as *determined variables,* respectively.

Transparency rules and implications can be described implicitly in the AG input model. This property reduces significantly the complexity of propagation and implication procedures during the test generation. In the following, two algorithms for AG generation from the tables of rules will be presented. It is assumed that the rules are given in the form where the value of the FU output is derived from the value of one and only one FU input. In order to reduce the complexity of the problem, only one- or two-input FUs with a single output will be considered.

Consider the description of Algorithm 1. The algorithm is more general than Algorithm 2, which is in turn less complex, differing from Algorithm 1 by considering only non-terminal nodes when substituting nodes with the subgraphs corresponding to the rule. Elimination of obviously redundant successors to a node is shown in Figure 13. The nodes to be eliminated are denoted with dotted circles. The procedure has been omitted from Algorithm 2 and is not described in detail in current thesis. Figure 13 explains AG generation from a rule table for a multiplier operation.

Algorithm 1:

Let *current subgraph* consist of the node *N* labelled by an operation;

while exist rules in the table **do**

Let V_d be the determined variable of the rule;

// In Algorithm 2, only non-terminal Mi-s are considered:
if there exist at least one node labelled by variable V_d in *current subgraph* **then**
while exist nodes M_i labelled by V_d **do**
Create a new node K_i labelled by the variable or constant
corresponding to the value of the FU output in the rule;
Create a branch from M_i to K_i labelled by the value of V_d in current rule;
if it is the only branch starting from M_i **then**
// The previous condition is always false in Algorithm 2:
Create a new node M_i*'* labelled by V_d ;
Create a branch from M_i to M_i*'* and label it by the rest of V_d *'*s domain
(i.e. the value of V_d in the rule excluded); *// Denoted with ε in Figure11*
else
Exclude the value of V_d from the subset of values labelling a
corresponding branch starting from M_i;
end if;

// Variable elimination is excluded from Algorithm 2:
Eliminate the redundant branches of the node M_i ;

end while;

else if there exist no nodes labelled by variable V_d in *current subgraph* **then**
Create a new node K_1 labelled by V_d;
Create a new node K_2 labelled by the variable or constant
corresponding to the value of the FU output in the rule;
Create a branch from K_1 to K_2 labelled by the value of V_d in current rule;
Create a branch from K_1 to *current subgraph*, label it by the rest of V_d *'*s domain
(i.e. the value of V_d in the rule excluded); *// Denoted with ε in Figure 11*
end if;

Include the newly created nodes to *current subgraph*;

end while;

Replace the node *N* with *current subgraph*;

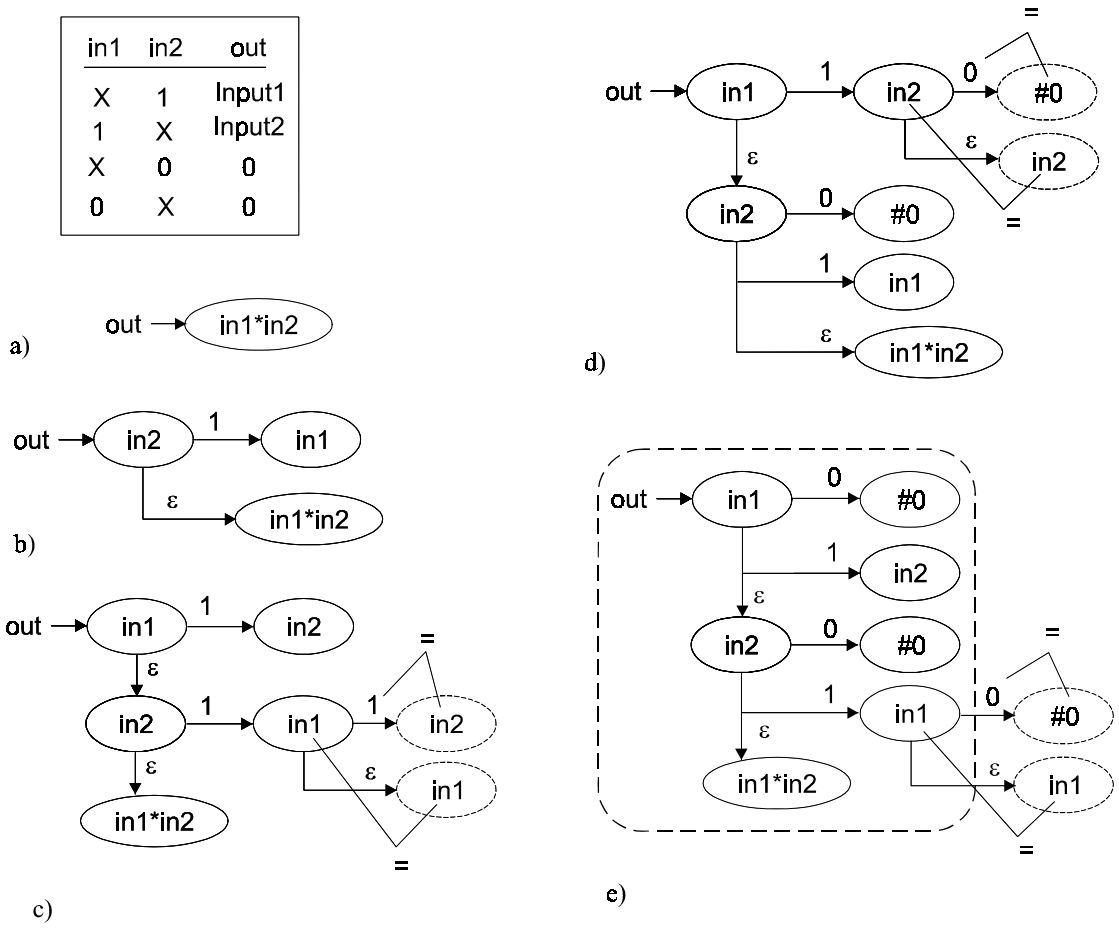

Figure 13. Generating AGs from Rule Tables

For the given example, as well as for the other common arithmetic functions, Algorithm 2 is applicable. However, for a rule given in the form where one of the inputs is equal to constant C_1 implying that the output must be equal to constant C_2, where $C_1 \neq C_2$, Algorithm 2 will give faulty results.

5.6 State Transitions

Current test generation algorithm is not directly targeting the faults in the control part of the device under test. However, the finite state machine (FSM) is functioning interactively with the datapath part of the circuit under test and the decisions made in state sequences are very important to the behaviour of the device as a whole. In the following, the use of state transition information for implications and decision making in current algorithm is explained.

There are three types of processes, where state transitions can be chosen or implied. The processes are setup, propagation and justification, respectively. Let us consider the algorithm for finding state transitions during the setup phase. An example of the process is presented in Figure 14, where the objective is to set up a conformity test for the node labelled by variable B. The following steps have to be performed.

1. *Set up a symbolic test.*

   In current example, the conformity test for the node *n* labelled by variable B, when B=1 is set up. The path from the root node to the corresponding terminal node is activated and the successor nodes of *n* are distinguished from each other by assigning symbols S1 and S2 to the respective terminal nodes. In the example, Y=F, A=1, B=1, K=S1 and L=S2.

2. *Create justification constraint stacks.*

   Justification constraint stacks labelled by current time frame are created for backtracing the symbols S1 and S2. In current example the stacks are created for K=S1 and L=S2 The process of creating constraint stacks is explained in detail in Subsection ‘Path Activation Constraints’.

3. *Select the next state.*

   While selecting next state, a terminal node *T* of the control part AG is chosen whose label values are consistent with the values of corresponding control signals activated during the setup. There exists always at least one such node. In the example, the chosen next state is state 3.

4. *Create conditional constraint stacks.*

   The path from the root node to *T* is activated and conditional constraint stacks are created for all the non-terminal nodes labelled by logic conditions. In the example, a stack is created for M<N=1.

5. *Imply current state.*

   Current state will be equal to the label of the activated branch of the node labelled by the state variable *q*. In the example, current state is 2.

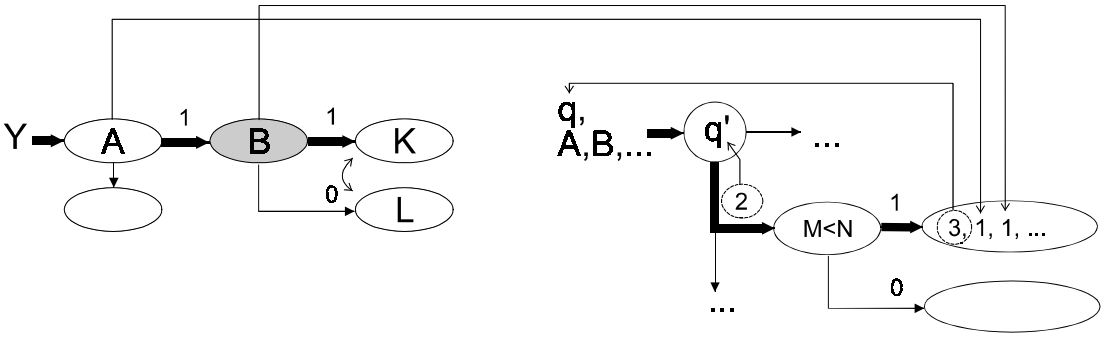

Figure 14. State Transitions during Setup

Figure 15 explains the algorithm for finding state transitions during the propagation phase. The objective of the example is to propagate the fault effect from variable L towards primary outputs, when current state is 3. In order to achieve that, the following steps are required.

1. *Propagation.*

   (Explained in Subsection ‘Propagation and Justification on the AG Model’).

2. *Select the next state*.

   While determining next state, a terminal node *T* of the control part AG is chosen whose label values are consistent with the values of corresponding control signals activated during the propagation. Different from determining next state during setup, the candidates for the terminal nodes are constrained to those following the branch of the node labelled by the state variable, when the branch is activated to the value of previous state. In the example, the chosen next state is state 4.

3. *Create conditional constraint stacks.*

   The path from the root node to *T* is activated and conditional constraint stacks are created for all the non-terminal nodes labelled by logic conditions. In the example, a stack is created for M==N=1.

If Y is not a primary output then current objective will be to propagate the fault effect from Y.

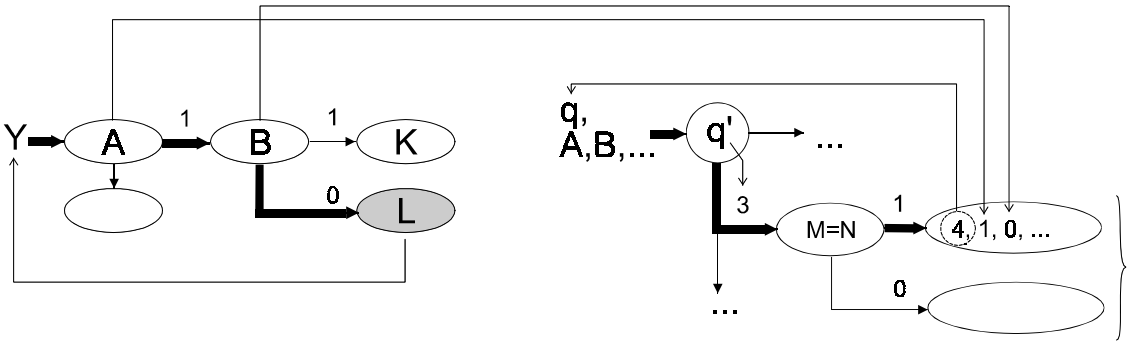

Figure 15. State Transitions during Propagation

In Figure 16, the algorithm for finding state transitions during justification is explained. The objective of the example is to select current state and to perform justification for variable Y, when the next state is 2. The algorithm consists of the following steps.

1. *Justification.*

   (Explained in Subsection 'Propagation and Justification on the AG Model').

2. *Determine current control vector.*

   While determining current control vector, a terminal node *T* of the control part AG is chosen whose label values are consistent with the value of the next state and with the values of corresponding control signals activated during the justification.

3. *Create conditional constraint stacks.*

   The path from the root node to *T* is activated and conditional constraint stacks are created for all the non-terminal nodes labelled by logic conditions. In the example, a stack is created for M>N=1.

4. *Imply current state.*

   Current state will be equal to the label of the activated branch of the node labelled by the state variable *q*. In the example, current state is 1.

5. *Update the constraint stacks.*

   (Explained in Subsection 'Path Activation Constraints').

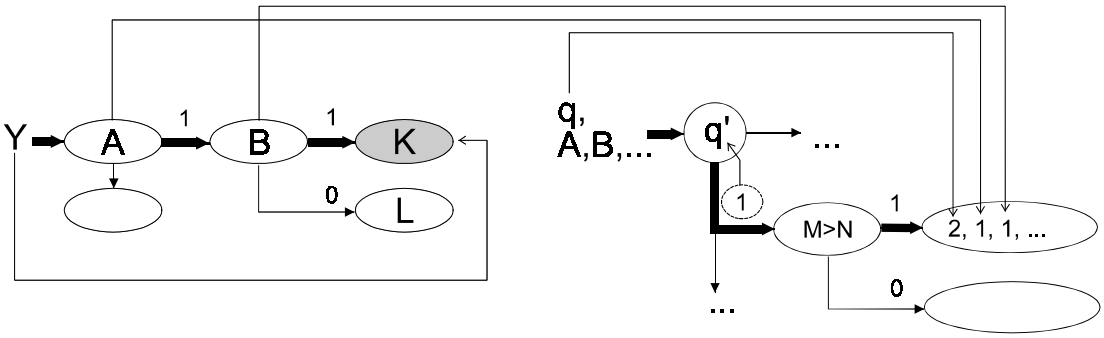

Figure 16. State Transitions during Justification

5.7 Path Activation Constraints

High-level test generator activates paths from primary inputs of the device up to the inputs of the component (i.e. multiplexer or functional unit) under test, and from the outputs of the component up to the primary outputs of the whole device. Constraints required to obtain an active path through the device are called *conditional constraints*. As was mentioned in the previous chapter, due to the presence of functional data and control constraints it is not always possible to activate an I-path [Abadir and Breuer 85] (i.e. absolutely transparent path). Thus, in general case, test values will pass through some other functional units and the values will be changed before they reach the inputs of the component under test. These changes are reflected in *justification constraints*.

It is possible to consider the behaviour of an RTL design as a flow graph of algorithm (FGA). FGA is a directed graph, which can be represented by a pair $G=(N,A)$, where N is a set of nodes and A is a set of directed arcs between the nodes. $N=C \cup O \cup S \cup E$, where C is a set of conditional nodes, O is a set of operator nodes, S is the starting node and E is the ending node. The algorithm starts from node S and

terminates in *E*. According to the result of the operation in conditional node *c*, the arc starting from *c* labelled by corresponding value will be selected. Operator nodes are labelled by operations, which can be executed simultaneously in a single clock cycle. From this type of nodes, always only a single arc starts. In the case of Moore automata, operator nodes are additionally labelled by states, in Mealy automata the labels are added to the arcs leading to operator nodes, respectively. Figure 17 shows an FGA corresponding to an RTL description of a multiplier. In the example, a Mealy automaton is implemented.

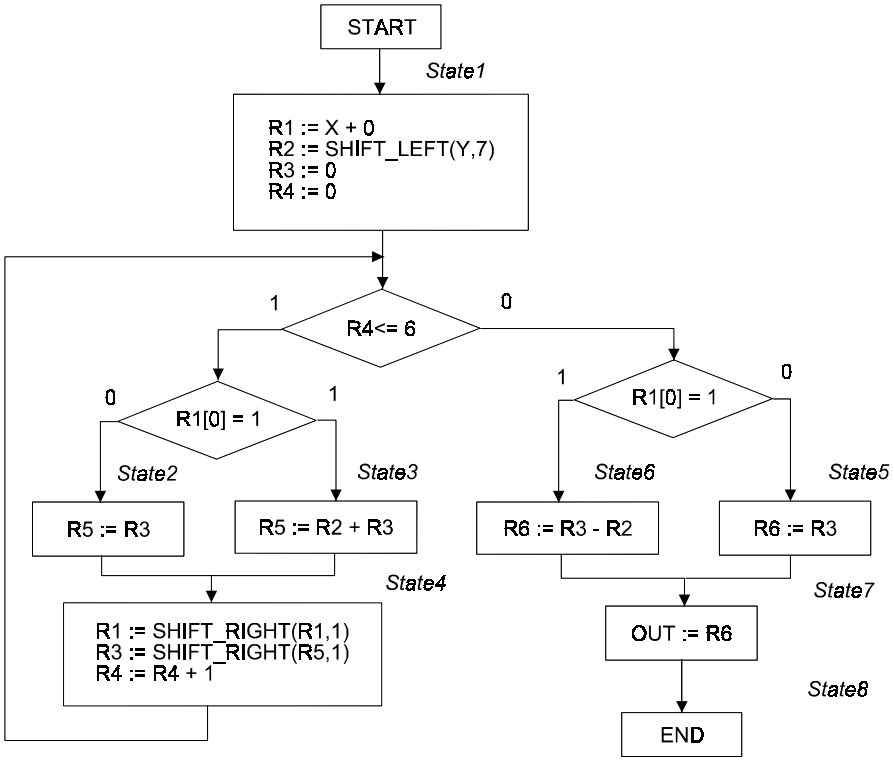

Figure 17. Flow Graph of Algorithm

Figure 18 explains the propagation and justification procedures during test generation based on the FGA model in Figure 17. In the example, a scanning test is generated for the operation in the grey rectangle. Propagation steps are denoted with black bold arrows and justification steps with grey bold arrows, respectively.

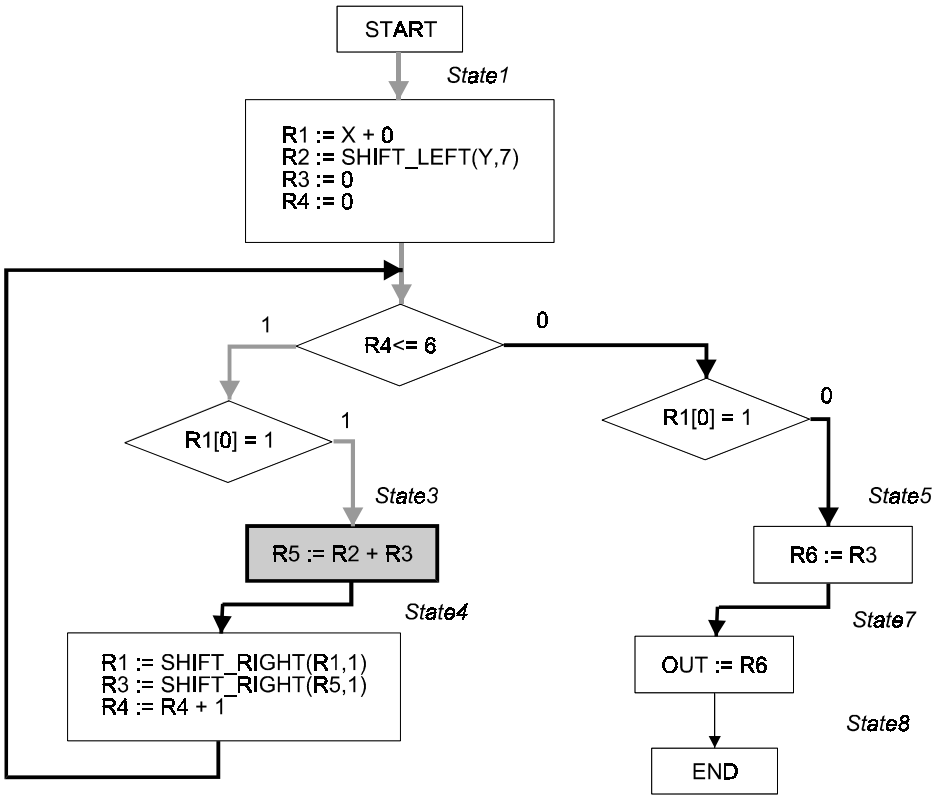

Figure 18. Path Activation on Flow Graph of Algorithm

In Figure 19, the state transitions during high-level symbolic path activation are shown.

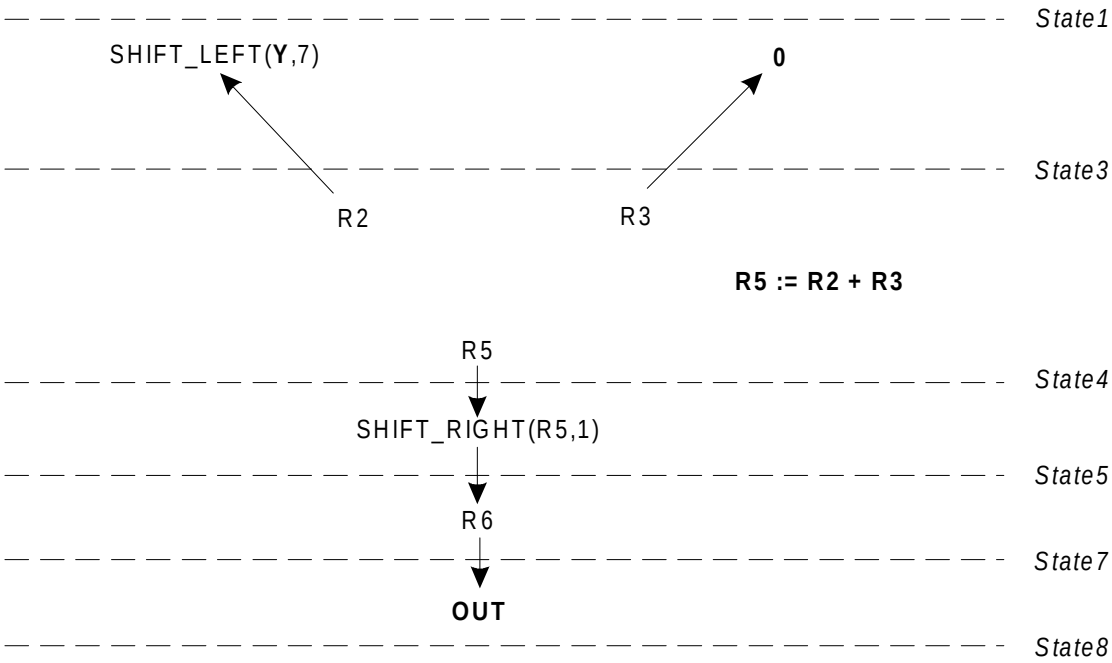

Figure 19. The State Sequence during Path Activation

Figure 20 presents the extraction of path activation constraints for the above example. The final results are shown on top of a grey background. Note that the extracted conditional constraints are inconsistent. According to the first conditional constraint the expression 0 + 1 <= 6 must be false, which is obviously not the case. Thus, backtrack will occur and the test generation algorithm will try to activate an alternative symbolic path.

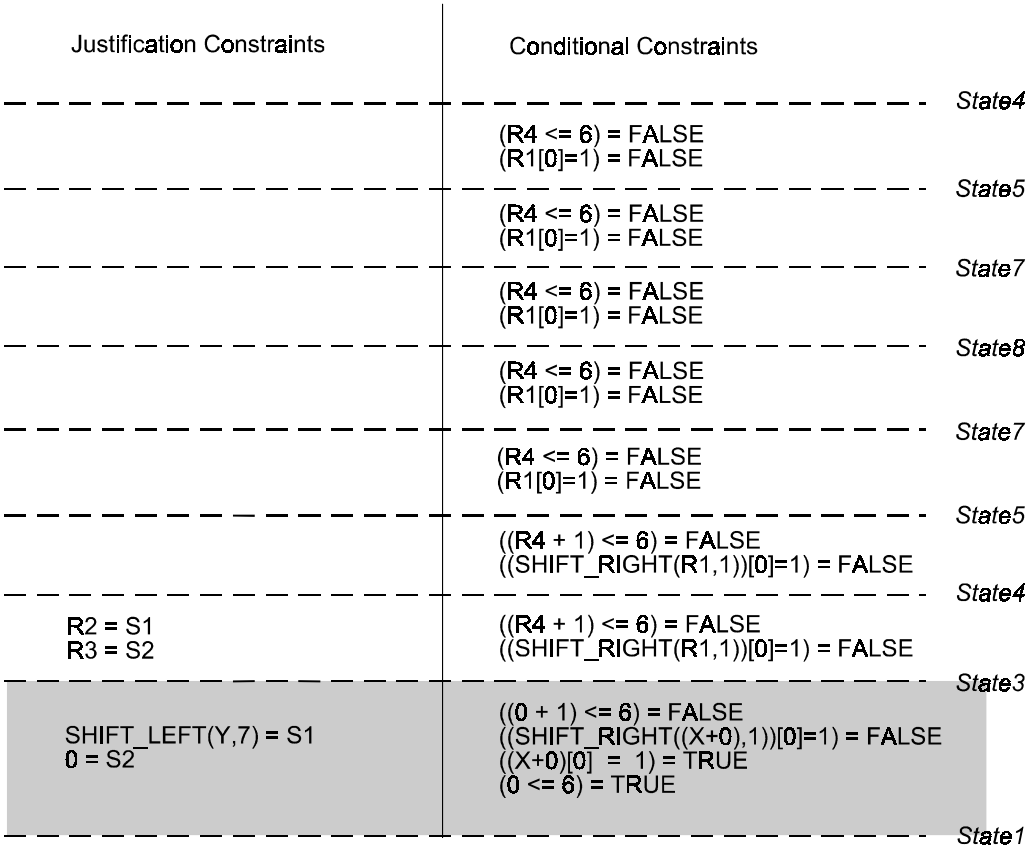

Figure 20. Constraint Extraction Example

As it can be seen from the above example, conditional constraints are updated during the backtrace from the final state as well as during the justification, and justification constraints are extracted during the justification process, respectively. A constraint stack is updated by simulating the values of the AGs corresponding to the variables contained in the constraint stack at current time frame. Subsection ‘State Transitions’ explains the situations when the steps of creating and updating constraint

stacks are applied. It is important to realize that only the stacks that were created during the later time frame than the current one can be updated.

Each time a constraint stack is created or updated, it will be simulated. This step makes it possible to find obvious inconsistencies at the early stage of path activation. It reduces the search space, and therefore, accelerates the path activation algorithm.

Subsequent to the constraint extraction, the constraints have to be solved. In order to achieve that goal, the following algorithm is used. If the justification constraint stacks and conditional constraint stacks do not contain any common variables then a single solution satisfying the conditional constraints will be calculated. Random values are generated for the variables of justification constraints, and the constraints will be simulated to obtain current values of the symbols S1 and S2. Due to the fact that these symbols have different interpretations in conformity and scanning tests, they are solved in different ways in scanning and conformity tests.

If there exists at least one variable which occur both in conditional constraints and justification constraints then a package of *n* solutions satisfying the conditional constraints will be found. These solutions will be used as the basis of solving the justification constraints. This step is necessary in order to derive more test vectors for a single activated path in the case when justification constraints depend on the values of variables in conditional constraints.

Solving the conditional constraints can be considered as a typical constraint satisfaction problem (CSP) [Montanari 74]. As the solution, any known CSP solving algorithm can be applied.

5.8 Interaction between High- and Low-Level Parts

During the scanning test for an FU, high-level part of the test generator calls the low-level generator repeatedly in a loop [Raik 96a]. When a path is activated, the high-level part will call low-level and will pass the extracted justification constraints to the latter. If the low-level part generates test vectors achieving 100 per cent fault efficiency for the FU, ‘success’ will be returned to the high-level generator. Current functional unit will be considered to be tested and next untested FU will be chosen by the high-level generator. In the case when the achieved fault coverage in current FU remains low or unchanged, high-level part of the test generator will be informed about it and it will try to activate an alternative path. Figure 21 shows the data flow of interaction between high-level and low-level parts.

In general case a single activated path is not enough to reach 100 per cent fault efficiency for a functional unit, i.e. test set for a FU can consist of vectors generated during different activated paths, and therefore, different calls to the low-level part. Thus, record has to be kept of the faults detected by previous low-level test generation runs.

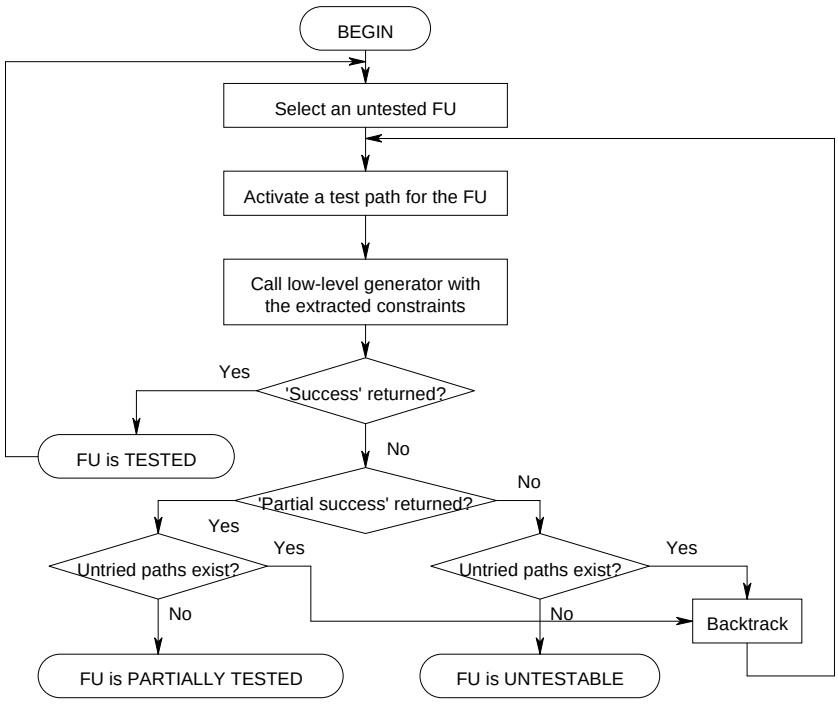

Figure 21. Interaction between High- and Low-Level Parts

Untestable faults for an FU are determined in current approach by the following method. Previous to starting test generation for the FU, deterministic gate-level test generator finds the list of redundant faults in the FU. If some inputs of the FU are directly tied to constant signals, it will be taken into account while determining the untestable faults. Fault efficiency is considered to be 100 per cent if *number of tested faults = total number of faults - number of untestable faults*. Due to possible functional constraints in the top-level circuit, 100 per cent fault efficiency may be unreachable.

6 Experimental Results

In the following, experiments carried out on prototype software will be presented. The software was implemented in co-operation with J.Krupnova and G.Jervan. The approach included scanning tests for FUs only. A dedicated library was used for applying transparency rules. Interactions between high- and low-level were realized via file transfer. A CSP solving algorithm called backtracking was applied to solve the conditional constraints.

Interaction limit	FUs	Interactions	Vectors	Result
1	Add1	1	7	success
	Add2	1	0	failure
	And1	1	3	success
	Sub1	1	0	failure
100	Add1	1	7	success
	Add2	100	0	failure
	And1	1	3	success
	Sub1	100	0	failure
1000	Add1	1	7	success
	Add2	1000	76	partial
	And1	1	3	success
	Sub1	244	0	failure
∞	Add1	1	7	success
	Add2	1093	78	success
	And1	1	3	success
	Sub1	244	0	failure

Table 1. Test Generation Results

As an input model for the datapath test generator a hierarchical model of a 16-bit multiplier was chosen. Test generation results for the four functional units (FU) in the circuit are given. By raising the interaction limit between high- and low-level generators three of the FUs were tested with 100 per cent efficiency. One of the FUs was not tested due to inaccurate modelling implemented in the prototype software. Test generation times are not included because at present file transfer is used for interaction between high and low level parts, which significantly reduces the speed. The test results are given on Table 1.

7 Conclusions

Current thesis presents a novel hierarchical test generation approach based on multi-level AG model descriptions. The main advantage of using AG models lies in the fact that the same, uniform concept can be applied on different system abstraction levels. In addition, AGs provide a powerful means for solving different diagnostic tasks. Another novelty introduced in current thesis is the symbolic path activation algorithm, where transparency and implication rules are described implicitly in the design. This property simplifies significantly the algorithm. Other recent approaches have been describing transparency rules explicitly in dedicated libraries.

The test generation algorithm considered in the thesis is aimed at testing the multiplexers, registers and functional units (FUs) of the datapath and the control signal decoders of the control part. However, the algorithm is not targeting directly all the control part faults and the faults in the FUs corresponding to logic conditions. Including test generation for the above mentioned targets to current approach could be the topic of future research.

The main problems during high-level path activation are the loss of accuracy during the symbolic fault effect propagation and the cases when the symbolic paths for a component under test can not be activated. The first problem can be handled by further improving the algorithm of fault propagation. Here, a trade-off between accuracy and processing speed should be chosen. The problem, where transparent paths for some modules can not be activated, can be managed in the following ways. The first way is to use design for testability methods, e.g. high-level synthesis for testability [Flottes 95, Bhatia 94]. These test synthesis algorithms allow the synthesis of register-transfer level (RTL) descriptions, where transparent paths for as many components as possible would exist, taking into account additional constraints, e.g. area, speed, power, etc. Although quite promising results have been achieved, the algorithms do not guarantee synthesis results with transparent paths for each component in general case. The other way to cope with the problem is to use different approach (e.g. bottom up approach, or less strict transparency rules during path activation) for the modules left untested during the first run.

These are the problems that still remain unsolved and should be overcome, in order to further improve the quality of the hierarchical test generation approach.

Bibliography

[Abadir 85] M.S. Abadir, H.K. Reghbati, "Functional specification and testing of logic circuits", Comp.& Math. with Appls. Vol.11, No.12, pp.1143-1153, 1985.

[Abadir and Breuer 85] M. S. Abadir and M. A. Breuer, "A Knowledge Based System for Designing Testable VLSI Chips", IEEE Design&Test, pp. 56-68, Aug. 1985.

[Abraham 86] J.A. Abraham, "Fault modeling in VLSI", VLSI testing, North-Holland, pp.1-27, 1986.

[Akers 78] S.B.Akers, "Binary Decision Diagrams", IEEE Trans. on Computers, Vol. 27, pp.509-516, 1978.

[Anirudhan 89] P.N. Anirudhan and P.R. Menon, "Symbolic Test Generation for Hierarchically Modeled Digital Systems", International Test Conf., pp. 461-469, Sept. 1989.

[Bhattacharya and Hayes 90] D. Bhattacharya and J.P. Hayes, "A Hierarchical Test Generation methodology for Digital Circuits", Journal of Electronic Testing: Theory and Application (JETTA), vol. 1, pp. 103-123, 1990.

[Bhatia 94] S. Bhatia, N.K. Jha, "Genesis: A Behavioral Synthesis for Hierarchical Testability", Proc. of the European Design and Test Conference, pp. 272-276, 1994.

[Bleeker 93] H. Bleeker, P. van den Eijnden, F. de Jong, "Boundary-Scan Test", Kluwer Academic Publishers, 225 p, 1993.

[Brahme and Abraham 84] D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors", IEEE Trans on Comp. vol. c-33, pp. 475-485, June 1984.

[Calhoun and Berglez 89] J. D. Calhoun and F. Brglez, "A Framework and Method for Hierarchical Test Generation", Proc. International Test Conf., pp. 480-490, Sept. 1989.

[Chandra 87] S.J. Chandra, J.H. Patel, "A hierarchical approach to test vector generation", ACM/IEEE 24th Design Automation Conf., pp.495-501, June 1987.

[Cheng 90] K.-T. Cheng, J.-Y. Jou, "Functional test generation for finite state machines", IEEE International Test Conference, pp.162-168, 1990.

[Corno 95] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "GARDA: A Diagnostic ATPG for Large Synchronous Sequential Circuits", The European Design and Test Conference, pp.267-273, March 1995.

[Crestani, Aguila, Eudeline 93] D. Crestani, A. Aguila, L. Eudeline, "A Hierarchical Test Generation Using High-level Primitives", 6-th International Conf. on VLSI Design, pp. 124-127, Jan. 1993.

[Drechter and Pearl 88] R.Drechter, J.Pearl, "Network Based Heuristics for Constraint Satisfaction Problems", Artificial Intelligence, Vol.34, pp. 1-38, 1988.

[Flottes 95] M.L. Flottes, D. Hammad, B. Rouzeyre, "High-Level Synthesis for Easy Testability", Proc. of the European Design and Test Conference, pp. 198-206, March 1995.

[EDIF 87] "Electronic Design Interchange Format Version 2.0.0", EIA Interim Standard No. 44, May 1987.

[Freeman 88] Freeman, Test Generation for Data Path Logic: The F-Path Method. IEEE Journal of Solid State Circuits, vol.23, pp. 421-427, Apr. 1988.

[Fujiwara and Shimono 83] H.Fujiwara, T.Shimono, "On the Acceleration of Test Generation Algorithms", IEEE Trans. Comput., vol. C-32, pp. 1137-1144, December 1983.

[Fujiwara 85] H.Fujiwara, "Logic Testing and Design for Testability", MIT Press, Cambridge, Massachusets, 1985.

[Gajski 89] D.Gajski, N. Dutt, A. Wu, S. Lin, "High-Level Synthesis, Introduction to Chip and System Design", Kluwer Academic Publishers, 1989.

[Ghosh 92] A. Ghosh, S. Devadas, A.R. Newton, "Sequential Logic Testing and Verification", Kluwer Academic Publishers, 214 p, 1992.

[Goel 81] P.Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Trans. Comput., vol. C-30, pp. 215-222, March 1981.

[Karam, Leveugle and Saucier 91] M. Karam, R. Leveugle, G. Saucier, "Hierarchical Test Generation Based on Delayed Propagation", Int. Test Conf., pp. 739-747, 1991.

[Krupnova and Ubar 94] H.Krupnova, R.Ubar, "Constraints Analysis in Hierarchical Test Generation for Digital Systems", Proc. of the Baltic Electronic Conference, Tallinn (Estonia), Oct. 9-14, 1994.

[Krupnova 94] H.Krupnova, "Model Synthesis from VHDL for the Functional Test Generation System", Master Thesis, Tallinn Technical University, Nov. 1993.

[Kunda, Narian, Abraham and Rathi 90] R. P. Kunda, P. Narian, J. A. Abraham, and B. D. Rathi, "Speedup of Test Generation Based on High-Level Primitive", Proc. 27th ACM/IEEE Design Automation Conf., pp. 580-586, June 1990.

[Leenstra 90] J. Leenstra, L. Spaanenburg, "Hierarchical test assembly for macro based VLSI design", International Test Confernece, pp.520-529, 1990.

[Lee and Patel 91] J.Lee and J.H.Patel, "ARTEST: An Architectural Level Test Generator for Data Path Faults and Control Faults", Proc. Int. Test Conf.., pp. 729-738, Oct. 1991.

[Lee and Patel 92] J.Lee and J.H.Patel, "Hierarchical Test Generation under Intensive Global Functional Constraints", Proc. 29th ACM/IEEE Design Automation Conf., pp. 261-266, June 1992.

[Leenstra and Spannenburg 90] J. Leenstra, L. Spannenburg, "Hierarchical Test Assembly for Macro Based VLSI Design", Int. Test Conf., pp. 520-529, 1990.

[Lin and Su 84] T. Lin, S. Y. Su, "Functional Test Generation of digital LSI/VLSI systems using machine symbolic execution technique", I.T.C., 1984, pp. 660-668.

[Lin 85] T. Lin, S.Y.H. Su, "VLSI functional test pattern generation - a design and implementation", IEEE International Test conference, pp.922-929, 1985.

[Mackworth 79] A.K.Mackworth, "Consistency in Networks of Relations", Artificial Intelligence, Vol.8, pp. 99-118, 1979.

[Magdolen, Bezakova, Gramatova, Fisherova 93] A. Magdolen, J. Bezakova, E. Gramatova, M. Fisherova, “REGGEN-Test Pattern Generation on Register Tranfer Level”, EURO-DAC, pp. 259-264, 1993.

[Min, Luh and Rogers 93] H. B. Min, H. A. Luh and W. A. Rogers, “Hierarchical Test Pattern Generation: A Cost Model and Implementation”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, N. 7, july 1993.

[Montanari 74] U.Montanari, “Networks of Constraints: Fundamental Properties and Apllications to Picture Processing”, Information Sciences, Vol.7, pp. 95-132, 1974.

[Niermann and Patel 91] T. Niermann, J.H. Patel, HITEC: A Test Generation Package for Sequential Circuits. Proc. EDAC, Feb. 1991.

[Raik 96a] J. Raik, R.Ubar, G. Jervan, H. Krupnova, “A Constraint-Driven Gate-Level Test Generator”, Proc. of the 5-th Baltic Electronics Conference, pp. 237-240, Oct. 1996.

[Raik 96b] J. Raik, P. Paomets, “Test Synthesis from Register-Transfer Level Descriptions”, Proc. of the 5-th Baltic Electronics Conference, pp. 311-314, Oct. 1996.

[Roth 66] J.P. Roth, “Diagnosis of Automata Failures: A Calculus and a Method”, IBM Journal of Res. and Develop., vol. 10, pp. 278-291, Jul. 1966.

[Santucci, Courbis and Giambiasi 93] J.F. Santucci, A-L. Courbis, N. Giambiasi, “Speed up of Behavioral A.T.P.G. using a Heuristic Criterion”, 30th ACM/IEEE Design Automation Conference, pp. 92-96, 1993.

[Sarfert, Makrgraf, Trishler and Shulz 89] T. M. Sarfert, R. Makrgraf, E. Trishler, and M. H. Shulz, “Hierarchical Test Pattern Generation Based on High-Level Primitives”, Proc. Int. Test Conf., pp. 470-479, Sept 1989.

[Saucier 84] G. Saucier, C. Bellon, “CADOC: A system for computer aided functional test”, IEEE International Test Conference, pp.680-687, 1984.

[Sridhar 79] T. Sridhar, J.P. Hayes, “Testing bit-sliced microprocessors”. 9th International Symposium on Fault Tolerant Computing, pp.211-218, 1979.

[Steensma, Geurts, Gatthoor and De Maan 93] J. Steensma, W. Geurts, F. Catthoor and H. De Maan, “Testability Analysis in High Level Data Path Synthesis”, Journal of Electronic Testing: Theory and Applications, 4, pp. 43-56, 1993.

[Synopsys 92] “Design Compiler Reference Manual Version 3.0”, Synopsys Inc., Dec. 1992.

[Thatte and Abraham 80] S. M. Thatte, J. A. Abraham, “Test Generation for Microprocessors”, IEEE Tran. Comp., vol. c-29, pp. 429-441, Jun. 1980.

[Tilly 94] K.Tilly, “A Comparative Study of Automatic Test Pattern Generation and Constraint Satisfaction Methods”, Technical Report, Ser. Electrical Engineering, Technical University of Budapest, June 1994.

[Ubar 76] R.Ubar, “Test Generation for Digital Circuits Using Alternative Graphs”, Proc. of Tallinn Technical University, Estonia, No. 409, pp. 75-81 (in Russian), 1976.

[Ubar 83] R.Ubar, “Test Pattern Generation for Digital Systems on the Vector AG-model”, 13-th International Symposium on Fault Tolerant Computing, Milano, Italy, pp. 347-351, 1983.

[Ubar 86] R. Ubar, “Testing of Digital Devices, I and II part”. Tallinn Technical University, Estonia 1986 (In Russian).

[Ubar 88] R.Ubar, “Alternative Graphs and Technical Diagnosis of Digital Devices”, Electronic technique, (USSR) Vol.8, No.5 (132), pp.33-57 (in Russian), 1988.

[Ubar 96a] R.Ubar, “Test Synthesis with Alternative Graphs”, IEEE Design and Test of Computers, Vol.13, No. 1, pp. 48-57, Spring 1996.

[Ubar 96b] R.Ubar, A.Markus, G.Jervan, J.Raik, “Fault Model and Test Synthesis for RISC-Processors”, Proc. of the 5-th Baltic Electronics Conference, pp. 229-232, Oct. 1996.

[VHDL 88] “IEEE Standard VHDL Language Reference Manual”, IEEE, 1988.

[Ward 90] P.C. Ward, J.R. Armstrong, “Behavioral fault simulation in VHDL”, ACM/IEEE 27th Design Automation conference, pp.587-593, 1990.

Hierarhiline testigenereerimine alternatiivgraafide mudelil

Koostaja Jaan Raik

RESÜMEE

Väga suurte integraallülituste testimise probleem muutub üha tähtsamaks, seoses nende keerukuse pideva kasvuga. Senituntud struktuursed loogikalülituste tasemel töötavad testigenereerimisalgoritmid ei ole praktikas rakendatavad, sest testigenereerimisülesanne kuulub keerukuselt NP-täielike ülesannete klassi. Seega on tegemist probleemiga, mille lahendamise aeg kasvab halvimal juhul eksponentsiaalses suhtes skeemi suurusest.

Kõne all oleva probleemi lahenduseks on esitatud mitmeid algoritme, mis kasutavad ära kõrgemate kirjeldustasemete informatsiooni testide genereerimisel. Need jagunevad põhimõtteliselt kahte klassi: funtsionaalseteks ja arhitektuurseteks algoritmideks. Funktsionaalsete algoritmide puuduseks on aga vähene informatsioon testitava objekti struktuuri kohta. Seega ei saavutata funktsionaalsete testide korral üldjuhul kuigi head vastavust reaalsetele füüsikalistele defektidele.

Käesolevas töös on kirjeldatud hierarhiline testigenereerimisalgoritm, mis töötab arhitektuursel ja loogikalülituste tasemel. Lähenemise põhiliseks uudsuseks on alternatiivgraafide mudeli kasutamine erinevate süsteemitasemete kirjeldamisel. Uuenduseks võib pidada ka levitamisreeglite ja implikatsioonide kujutamist testitavas mudelis endas. Lisaks öeldule on töös kirjeldatud alternatiivgraafide genereerimine erinevatest tasemetest ning esitatud eksperimendid testigeneraatori prototüüptarkvaral. Lisas on kirjeldatud alternatiivgraafide failiformaadi süntaks. Töö on kirjutatud inglise keeles.

Töö juhendaja: prof. Raimund Ubar