June 1992

UILU-ENG-92-2213
CRHC-92-09

Center for Reliable and High-Performance Computing

AUTOMATIC GENERATION OF INSTRUCTION SEQUENCES FOR TESTING MICROPROCESSORS

Jaushin Lee and Janak H. Patel

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-92-2213 CRHC-92-09	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL (If applicable) N/A
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7a. NAME OF MONITORING ORGANIZATION Semiconductor Research Corporation
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 7a	8b. OFFICE SYMBOL (If applicable)
8c. ADDRESS (City, State, and ZIP Code) 7b	7b. ADDRESS (City, State, and ZIP Code) Research Triangle Park, NC 27709
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Automatic generation of instruction sequences for testing microprocessors

12. PERSONAL AUTHOR(S) LEE, Jaushin and J. H. Patel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	92-06-05	23

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	VLSI, CAD, high level test generation, microprocessors

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Hierarchically designed microprocessor-like VLSI circuits have complex data paths and complex embedded control machines to execute instructions. When a test pattern has to be applied to the input of an embedded module, determination of a sequence of instructions, which will apply this pattern and propagate the fault effects, is extremely difficult. In this paper, we present a new methodology for automatic assembly of a sequence of instructions to satisfy the internal test goals. Combined with the previous equation-solving approaches, this new high level ATPG methodology forms a complete solution for a variety of microprocessor-like circuits. This new approach has been implemented and experimented on three high level circuits. The results show that our approach is very effective in achieving complete automation for high level test generation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

Automatic Generation of Instruction Sequences for Testing Microprocessors

Jaushin Lee and *Janak H. Patel*

University of Illinois at Urbana-Champaign
Center for Reliable and High-Performance Computing
1101 W. Springfield Ave.
Urbana, IL 61801, U.S.A.

Contact Person:
Jaushin Lee
Phone: (217)-244-7170
FAX: (217)-244-5685
lee@crhc.uiuc.edu

Abstract:

Hierarchically designed microprocessor-like VLSI circuits have complex data paths and complex embedded control machines to execute instructions. When a test pattern has to be applied to the input of an embedded module, determination of a sequence of instructions, which will apply this pattern and propagate the fault effects, is extremely difficult. In this paper, we present a new methodology for automatic assembly of a sequence of instructions to satisfy the internal test goals. Combined with the previous equation-solving approaches, this new high level ATPG methodology forms a complete solution for a variety of microprocessor-like circuits. This new approach has been implemented and experimented on three high level circuits. The results show that our approach is very effective in achieving complete automation for high level test generation.

This research was supported by the Semiconductor Research Corporation under Contract SRC 91-DP-109.

1. Introduction

A number of novel approaches for gate level sequential circuit test generation have been proposed [1-3]. These approaches accept a completely flattened gate level structure without looking at any of the inherent functional or behavioral information in the sea of logic gates. The efficiency of the gate level ATPG approaches will soon degrade as the size of the VLSI circuits increases. Most of the microprocessor-like VLSI circuits are designed hierarchically. The circuit hierarchy information has been exploited to speedup the test generation process in [4-7]. However, the ATPG algorithms used in these approaches are extended from D-algorithm[8] or PODEM[9] , so that the module partition cannot go much further than gate level. A higher level module can be accepted in [5]. However, it needs to be flattened to gate level if any interior single stuck-at fault is considered. The speedup achieved by these approaches is therefore limited due to the inherent property of the ATPG algorithms. Sarfert, et. al. updated SOCRATES[10] by including higher level modules which implement non-trivial functions, like multiplexer, decoder, adder, etc [11]. Kunda, et. al. take advantage of both higher level primitives and bit-vector representation of signals to speedup the test generation process [12]. However, these two approaches can only be applied to combinational circuits, and the stuck-at faults should be injected one at a time. Murray and Hayes[13] assume that the test set for the module under test has been precomputed, and an *identity propagation criterion* is used to propagate the whole test set without modifying its responses. However, this approach cannot accept circuits with an embedded control unit, and the identity propagation criterion is not always satisfied. Lee and Patel[14] proposed a data type manipulation technique to model all possible high level fault effects to replace the identity propagation criterion. A modified PODEM is interfaced with the high level branch-and-bound algorithm to handle the control unit. This technique has adopted various important factors to speedup the ATPG process. However, the high level branch-and-bound algorithm cannot efficiently solve the **data flow path conflicts** and **data flow value conflicts** which happen during searching.

One common feature in the above approaches is that only structure information is used in the ATPG process. The circuit hierarchy information and functional information is limited to local modules only. The global behavioral information of the machine under test is not adopted to speedup the ATPG process or to avoid

conflicts. Brahme and Abraham[15] propose an instruction execution process at a behavioral level. However, a functional fault model is used and the fault coverage for a low level fault model, like single stuck-at fault model, cannot be reported. Roy and Abraham[16] used data flow descriptions to perform hierarchical test generation. A data flow graph is recursively selected for propagating or justifying signals. However, their approach has the similar drawbacks as Murray's approach, namely, conflicts due to reconvergence are not handled for fault effect propagation. The algorithm of justifying multiple testing objectives is also not addressed in the paper. Wu, et. al. used high level synthesized RTL description to compute tests at a high level [17, 18]. A behavioral testability measure analyzing value ranges of registers is utilized in path justification and propagation in test generation. However, the test generator can only justify one testing objective at a time, and in general, it cannot handle the value conflict and path conflict problem when several testing objectives need to be justified simultaneously. Lee and Patel [19] have proposed a relaxation-based test generation algorithm at an architectural level. This algorithm has shown the efficiency for avoiding value conflicts for circuits with very complex data path configurations. However, the algorithm of avoiding global path conflicts using behavioral information has not been addressed yet in that paper.

From the previous work, some guidelines emerge for the high level ATPG. Due to a better correspondence to physical failures of chips, a lower level fault model like single stuck-at fault model should be targeted and a hierarchical structure information should be used. The circuit hierarchy and module functional information has to be adopted to speedup the test generation process. In addition to speedup, the test generator needs to be capable of avoiding both path conflicts and value conflicts at the high level. In this paper, we separate the ATPG process into two phases to test microprocessor-like circuits. In the first phase, a new instruction sequence assembling algorithm at an architectural level is proposed to solve global path conflicts using behavioral information. The behavioral knowledge of instructions is derived from techniques in the preprocessing phase. User of the tool does not need to supply this information explicitly. The equation-solving approach[19] is then applied to compute a global value solution at the module structure level in the second phase, and the value conflicts are avoided efficiently. Since the goal of assembling the instruction sequence at a

higher level is to avoid path conflict, we propose a **Flow-Influence model** for each primitive to capture its behavior for activating data paths only. A detailed data value analysis can be avoided at this stage. The important techniques used in this paper are highlighted as follows:

1. For each instruction, a **Structural Data-flow Graph (SDG)** is *automatically derived* from the corresponding system of equations which fully characterizes both module structure information and the register transfer level behavioral information. Each node in the graph is assigned with attributes of strong or weak influence, which are used to determine justification and propagation paths.

2. Given a set of SDGs for the instruction set, a justification cost will be calculated for each Next State Line (NSL) of the module diagram, which implies the minimum number of instructions needed for controlling that particular NSL. With the same fashion, a propagation cost can also be calculated for each Present State Line (PSL) showing the minimum number of instructions needed for observing a fault effect at this particular PSL. It should be noted that the number of instructions needed is estimated by checking the existing paths on the SDGs. The exact values or value ranges are not calculated so that the CPU run time can be significantly reduced. The advantage originates from the fact that, in the second phase, the equation-solving approach is applied to derive the exact values which satisfy the testing objectives at a lower level.

3. For a practical testing problem, *multiple objectives usually need to be justified simultaneously*. Even though we have reduced the complexity of the work of assembling instruction sequences using a Flow-Influence model, it can be shown that, given an SDG, checking the simultaneous justifiability of all objectives on the SDG is still an NP-complete problem. Therefore, instead of checking the simultaneous justifiability of all objectives, a branch-and-bound algorithm based on a concept of the network flow problem will be presented to search for the justification paths.

4. A special state line termed **Hard-To-Control (HTC)** state line can only be reset by instructions. The value at this line can never be loaded from input buses without depending on itself. A testability measure

usually computes higher costs for this HTC lines [20]. When the values of the HTC lines determine data flow paths, the instruction sequence assembling task turns out to be extremely difficult. Some special techniques are needed if some HTC state lines exist in the circuit under test.

The remaining part of the paper is outlined as follows. The global ATPG methodology is introduced in Section 2. An overview of the instruction sequence assembling process is offered in Section 3. Section 4 shows the definition of structural data-flow graph and the cost value calculation algorithm for controlling or observing a particular NSL or PSL. Section 5 shows the proof of the NP-completeness for the problem of checking simultaneous justifiability of multiple objectives on a given SDG. A branch-and-bound algorithm will be presented as an alternate approach to search for paths given a reasonable amount of time. For circuits with HTC state lines, some special techniques need to be developed and they are shown in Section 6. Sections 7 contains the experimental results and followed by the conclusions.

2. Global ATPG Methodology

The hierarchically designed circuits have two major characteristics that cause the high level ATPG task to be extremely difficult. The first characteristic is a complex data path configuration, in particular when some bus values determine control flows. A value conflict is very likely to happen when a solution is searched. The second type is a complex instruction set controlled by an embedded control machine. The control word sequence of each instruction determines a data flow graph in the data path. To satisfy multiple testing objectives in the data path, several instructions are usually needed for the purpose. A path conflict is therefore likely to happen in the instruction sequence derived. Due to these two major characteristics in high level circuits, we propose that the test generation process has to be separated into two phases in which each type should be handled by different techniques. The previous equation-solving approach in[19] has proved its efficiency in avoiding value conflicts on circuits with complex data path configurations. In this paper, a new instruction sequence assembling algorithm is proposed to handle the global path conflicts using behavioral information. The complete high level ATPG methodology is shown in Figure 1. Since the value conflicts will be handled in the

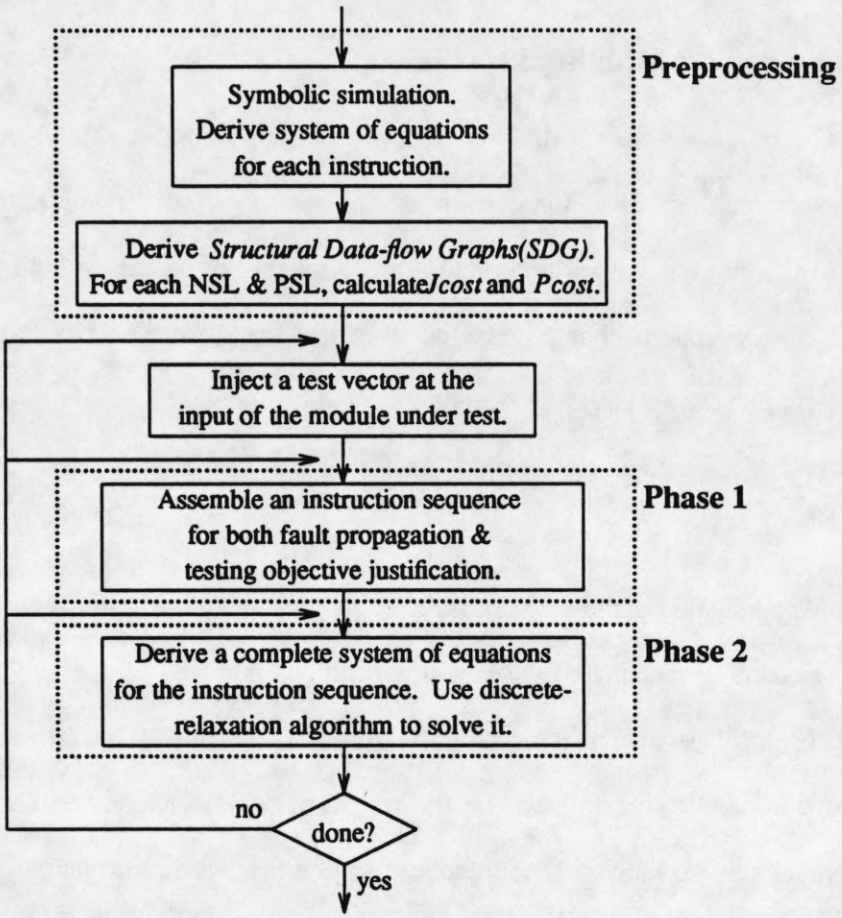

Figure 1. The high level ATPG methodology used in ARTEST.

second phase, only path analysis is required in the instruction sequence assembling phase. In this paper, only the techniques used in Phase 1 are addressed. The techniques in Phase 2 should be referred in [19].

It is assumed that the circuits under test are microprocessor-like circuits with an instruction set. The control machine is assumed embedded, and *no scan is assumed anywhere in the circuit*. The instructions under consideration are assembly level instructions, and the control word sequence of each instruction is assumed to be known. It should be noted that this assumption is **not** equivalent to scanning all inputs and outputs of the control machine. The micro-instruction sequence of each instruction is fixed. Neither the control words nor the status feedback lines are directly controllable. The circuits are described as a high level module diagram in which each module is defined by the primitives kept in our library. In this work, only data path faults are

considered, so the instructions used are always valid instructions.

3. Instruction Sequence Assembling Process: An Overview

Since the ATPG process has been partitioned into two phases, the exact data values at the interior bus lines need not to be estimated or calculated in this phase. The behavioral information of instructions is utilized to search for a valid instruction sequence which contributes a sequence of data flow graphs covering the testing objectives. In phase 2, an equation-solving technique is applied to derive the exact bus values in the given sequence of data flow graphs. Hence, in Phase 1, the functional information of modules is used only for determining active paths instead of computing the exact values so that the searching process can be accelerated significantly.

After the symbolic simulation in the preprocessing phase, a system of equation is derived for each instruction. This system of equations contains all the module operations involved in this particular instruction, and also the register transfer level behavioral information is implicitly embedded. A Structural Data-flow Graph (SDG) for this instruction can be created based on the system of equations, and the inherent register transfer level behavioral information is exploited to determine justification paths and propagation paths. Since the exact data values at buses need not to be calculated in the path searching procedure, we propose a **Flow-Influence model** to characterize the behavior of different primitives for selecting paths only. The detailed definition of this model will be explained in Section 4.

Based on the FLow-Influence model, a justification cost (*Jcost*) for each state line† of the data path can be calculated iteratively using all SDGs. These cost indicates the minimum number of instructions needed to set up a sequence of data flow graphs for justifying that particular state line **only**. A propagation cost (*Pcost*) can be also calculated in the same fashion. This cost values will be utilized to select the best instructions in the sequence assembling procedure.

---

† Present State Line (PSL) and Next State Line (NSL) are structurally identical except with one cycle difference in their timing. State line is used here as a general term to represent both.

The testing problem in Phase 1 needs to be clarified. The algorithm for fault effect propagation is similar to that of testing objective justification, so only justification task is addressed in the following discussions. When a test vector is to be justified at the input of the module under test, an instruction which covers all testing objectives will be selected first. With the SDG of that particular instruction, the corresponding PSLs, which are needed to justify all objectives in the selected instruction, can be determined. Afterwards, the instruction sequence assembling algorithm iteratively selects instructions to simultaneously justify all state lines. When an instruction is tried in the meantime, its SDG is used to check whether this instruction is able to simultaneously justify **all** objectives left by the previous instructions, and the cost of using this instruction is calculated based on the set of PSLs needed by this instruction. All instructions need to be tried, and the one having the least cost will be selected first. It is noted that if an instruction implements HOLD function for some particular state line, this line is treated as justified by this instruction, and this line is also left and needs to be justified by the next instruction. The concept of this process can be illustrated by Figure 2. It should be noticed that this algorithm needs to justify all state lines **simultaneously**. The *Jcost* is only used to calculated the total cost based on the state lines needed at the first cycle of each instruction.

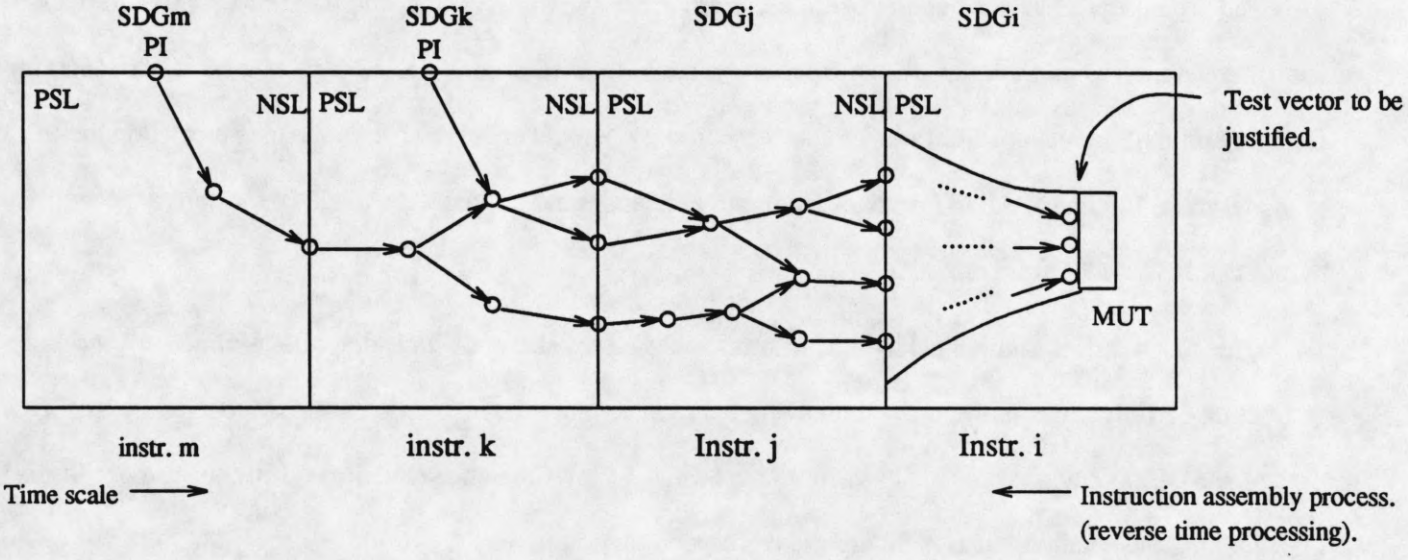

Figure 2. An picture illustrating the instruction sequence assembling process.

Given an SDG of an instruction, the task of checking the simultaneous justifiability of all state lines under the Flow-Influence model is conceptually similar to a network flow problem. Unfortunately we found that it is an *NP*-complete problem. Since the behavior of the instruction is explained by the Flow-Influence model, it is not practical to pay high run time to prove the simultaneous justifiability for all instructions. Despite the failure of the simultaneous justifiability of all state lines under the Flow-Influence model, there still exists a chance that they are justifiable at the lower level, even though the chance is very low. A branch-and-bound algorithm is developed to determine the justification paths for all state lines given a reasonable amount of CPU time, and derive the set of PSLs needed to be justified by the next instruction. The detailed algorithm will be illustrated in Section 5.

4. Structural Data-Flow Graph (SDG)

4.1. Definition of SDG

The structural data-flow graph of each instruction is derived from the system of equations produced by the symbolic simulation in the preprocessing phase. Each equation represents a operation at a module in the data path, and it contributes to one node in the SDG. Each arc in the SDG, corresponding to one symbol in the system of equation, indicates a signal flow from one module to another module. If there is a signal flow starting from any primary inputs or PSL, or arriving at any primary outputs or NSL, an extra node is created in the SDG to model these I/O lines. All the primary input and PSL nodes are leaf nodes of the SDG, whereas the primary output and NSL nodes are root nodes.

The SDG carries similar behavioral information as does the data flow graph determined by the same instruction. All modules in the data path having operations under the instruction will be modeled by the SDG. Since the SDG is only used for path selection, we design a Flow-Influence model to fulfill this purpose instead of examining the detailed functionality of the corresponding primitive.

4.2. Flow-Influence Model

For each type of primitive in our library, a Flow-Influence model is assigned for justification and propagation path selections. Each input of a primitive is classified as **strongly** or **weakly influencing**. Their definition is as follows:

Definition 1:

1. A strongly influencing input is one which must be justified to justify an output of the module. □
2. A weakly influencing input is one which need not be justified to justify the output, and exact one of all weakly influencing inputs is able to affect the output. □

Several examples are shown in Figure 3. The first one is an Adder with all its inputs with strong influence. It implies that if the output of the Adder needs to be justified, all its inputs also need to be justified. The second example is a 2-to-1 multiplexer. The input bus A and B have weak influences, whereas the select

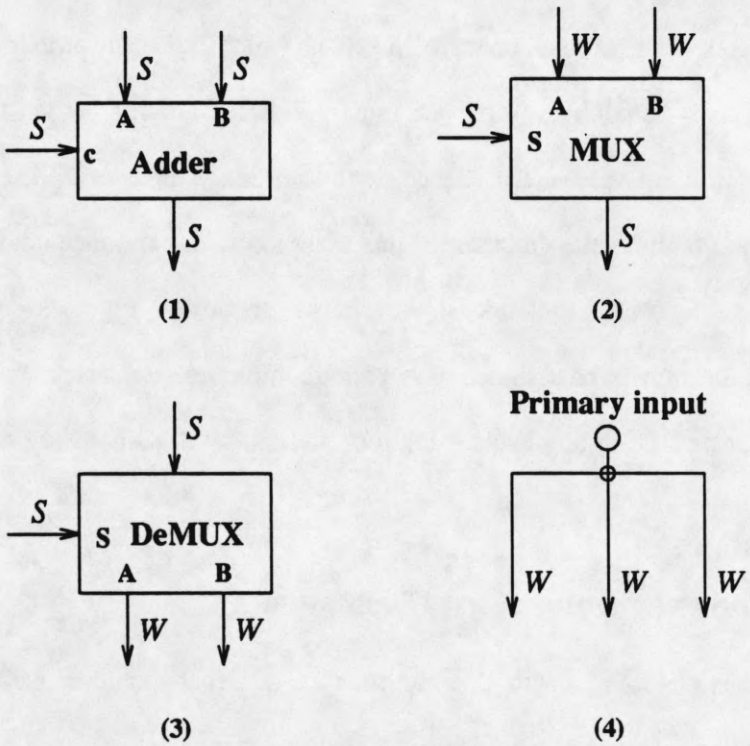

Figure 3. Examples illustrating the Flow-Influence model.

input has a strong influence. When the output of the multiplexer needs to be justified, the select input and either bus A or bus B need to be justified.

With a similar concept, each output of a primitive is classified as **strongly** or **weakly influenced**. Only one of all weakly influenced outputs of a primitive can be used to justify objectives since only one of them can be independently controlled by the inputs of this primitive. Once we decide to control a weakly influenced output, the other outputs automatically take on values dependent on the controlled output. The third example in Figure 3 is a 1-to-2 De-multiplexer with all its output buses weakly influenced. These two weakly influenced output buses imply that (normally) only one of them can be used to justify another objectives. Exceptions do exist in which both outputs can be simultaneously used to justify other objectives at the high level. However, this is a rare occurrence and is not accepted in our model.

Based on the characteristic of the functionality of each primitive, the influences at its inputs and outputs can be determined. This influence information is sufficient for determining active paths in finding justification and propagation sequences. For the leaf nodes of an SDG which represent primary inputs or PSL, a weak influence is assigned to their output arcs. This assignment is based on an assertion that, because most of the testing objectives have different values, the chance for one primary input or NSL signal to satisfy multiple objectives is very low at a high level. Violation of this assertion would produce much higher chance of value conflicts in Phase 2. This concept is illustrated by the fourth example in Figure 3. However, if the leaf node has only one output arc, the arc will be assigned with a strong influence. Because only one signal can arrive at a particular primary output or NSL at any cycle, each root node of an SDG has only one input arc. This arc is assigned with a strong influence.

4.3. Justification and Propagation Cost Calculation

The justification cost (*Jcost*) of a state line indicates the cost that should be paid if this node needs to be controlled **individually** through several instructions. The propagation cost (*Pcost*), on the other hand, indicates the minimum cost for observing a particular state line. These values will be used heuristically to compute the

total cost for each instruction which passes the simultaneous justifiability check in the instruction sequence assembling process.

A minimum dependence cone on each SDG is derived by a depth-first search algorithm for each state line. A minimum dependence cone of a state line *n* contains minimum number of PSLs which need to be justified before justifying the line *n*. A controllability measure similar to SCOAP[21] is calculated on each SDG using the influences assigned at the arcs, and a depth-first search algorithm can easily find out the nearest paths from line *n* to primary inputs and PSLs. An example in Figure 4 shows a minimum dependence cone of line *n* which contains one primary input line and two PSLs. For the justification purpose, only these two PSLs are recorded. If a minimum dependence cone on an SDG is empty, it means the line *n* can be justified by primary inputs only in this SDG. If an instruction initializes a line *n* by forcing its value to be a fixed value, the corresponding minimum dependence cone is not defined. Now we are ready to define a **Hard-To-Control (HTC)** state line.

Definition 2:

A state line *n* is an HTC state line if it satisfies following criterions:

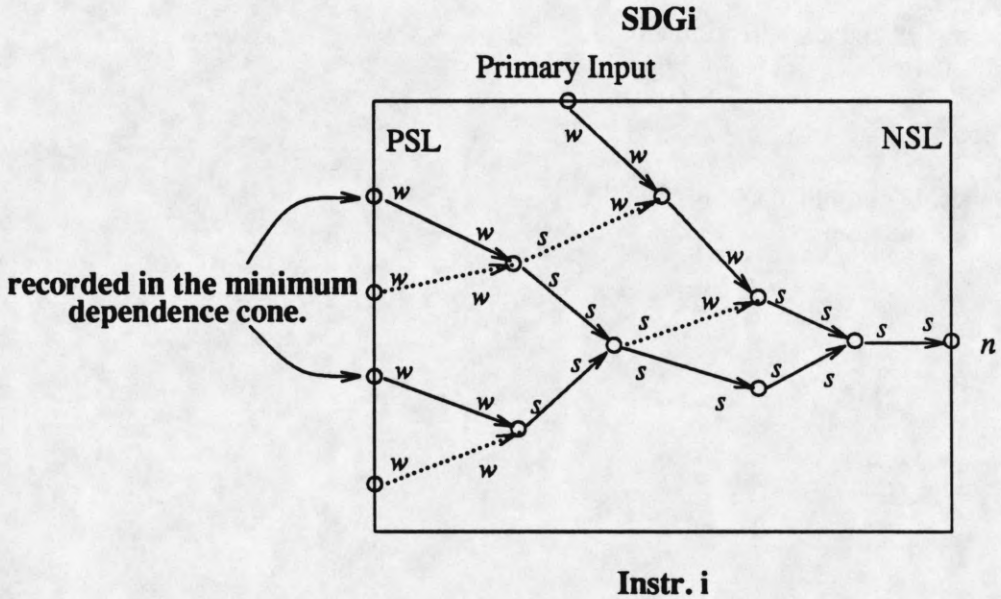

Figure 4. An minimum dependence cone of the state line *n*.

1. There exists at least one instruction which initializes line *n* to a constant value.

2. Line *n* has no empty minimum dependence cone for all instructions.

3. All the defined minimum dependence cones of *n* for all instructions include *n* itself. □

The definition of an HTC state line can be generalized to a group of state lines among which a loop dependence relationship exists given the instruction set. One example of an HTC state line is the state line in a counter sitting in a large circuit which cannot be loaded. Some instructions can be used to reset the counter, and count the value up and down. With the minimum dependence cone information for all state lines on all SDGs, the *Jcosts* for all state lines can be calculated iteratively. The *Jcosts* for all state lines are initialized with infinity. For a state line *n*, if there exists at least one minimum dependence cone to be empty, that is, this instruction allows line *n* to be loaded from primary inputs only, its *Jcost* is set to zero. The algorithm of calculating *Jcosts* is as follows:

```
Calc_Jcost() {
  modify_flag = TRUE;
  while (modify_flag) {
    modify_flag = FALSE;
    for (all state lines, n) { /* update Jcost for line n */
      if (n is an HTC state line)
        continue;
      for (all instructions, i) {
        sum_cost = 0;
        for (all state lines, k, k ∈ min_cone[n][i]) {
          if (k is an HTC state line)
            continue;
          else
            sum_cost += Jcost[k];
        } /* for */

        if (sum_cost + 1 < Jcost[n]) {
          Jcost[n] = sum_cost + 1;
          modify_flag = TRUE;
        }
      } /* for */
    } /* for */
  } /* while */
} /* Calc_Jcost() */
```

The *Pcosts* for all state lines can be calculated with a similar algorithm. The value of *Jcost* indicates the minimum number of instructions needed for justifying the particular state line. An HTC state lines have a very large *Jcost* indicating it is very difficult to control.

5. Simultaneous Justification of Multiple Objectives

A global view of the instruction sequence assembling process has been presented in Section 3. In this Section, we will concentrate on one iteration of the process and discuss the justifiability of a given set of objectives on an SDG. The example in Figure 2 is used again, and without losing the generality, the instruction j is focused. The set S is defined to be the set containing multiple state lines to be justified. Each state line, $n \in S$, would correspond to a root node in the SDG_j, or it implements a HOLD function under the instruction j. Those state lines implementing a HOLD function will not cause a justifiability problem since they are not in the SDG_j. These lines are not considered in the discussion. The simultaneous justifiability problem can be conceptually treated as a network flow problem. Each root node $n \in S$ can be imagined as a faucet delivering one unit volume of water which will flow through SDG_j. All nodes and arcs in SDG_j have no maximum capacity limitation. However, *at the inputs and outputs of each node, only one arc with a weak influence can be used to transport water*. At each node, the total volume of water is evenly distributed to all input arcs which are eligible for transportation. For example, if a node has one strong input arc and two weak input arcs, the water received at this node should be divided into two barrels and sent to the strong arc and one of the weak arcs. As the capacities of nodes and arcs are not limited, the only locality causing the unconservation of the network flow is at the nodes with weakly-influenced output arcs. Only one of the weakly-influenced arcs can transport water through the node. Based on the above model, following corollary and theorem are deduced.

Lemma 1:

If the network flow is conservative everywhere in SDG_j, then S is simultaneously justifiable.

Proof:

Since the network flow is conservative everywhere in SDG_j, there exists some pipes (paths) connecting all

roots in S to some leaf nodes of SDG_j. This directly leads to the definition of simultaneous justifiability of S under the Flow-Influence model. □

Theorem 1:

S is simultaneously justifiable if all nodes in SDG_j have strongly-influenced arcs at their outputs.

Proof:

Since there is no node having weakly-influenced output arcs in SDG_j, the network flow is always conservative. Since the network flow is conservative everywhere in SDG_j, from Lemma 1, the simultaneous justifiability can be concluded. □

Most general SDGs contain some nodes with weakly-influenced output arcs. In order to ensure the simultaneous justifiability of *S*, all we need to check is whether any node, having a group of weakly-influenced arcs at its outputs, is required to transport water from more than one of its weakly-influenced arcs. Unfortunately the complexity of this checking process on a given SDG is found to be very high. The following theorem and proof offer a theoretical argument.

Theorem 2:

Given an SDG with some nodes having k weakly-influenced output arcs, to check the simultaneous justifiability of a given set (S) of root nodes is *NP*-complete.

Proof:

Please refer to Appendix I.

An alternate approach based on a branch-and-bound algorithm is developed to search for valid simultaneous justification paths given a time limitation. Either the time limit is expired or a backtracking happens when the decision stack is empty, this process returns a failure. If neither of the above situations happens, this algorithm would search for valid justification paths for all root nodes in *S*. SDG_j is levelized from the root nodes to leaf nodes, and all root nodes in *S* and their input arcs are initially marked. The algorithm proceeds level by level to search for justification paths. When a node has any of its output arcs marked, this node has to be marked. Whenever a node is marked, all its strongly-influencing input arcs should also be marked. If a node

having weakly-influencing inputs arcs is confronted, only one of them will be chosen and marked, and this node will be put into the decision stack. When an arc is marked, the node feeding this arc has to be checked if the arc is one of its weakly-influenced output arcs. If more than one weakly-influenced output arcs need to be marked under current status, a conflict happens and the backtracking mechanism is invoked. The backtracking mechanism is similar to the one in PODEM [9]. When the backtracking mechanism is invoked, the top node in the decision stack is checked and the next weakly-influencing input arc will be tried. If all its weakly-influencing arcs have been tried, this node is popped out and the next node will be backtracked. After backtracking, the process restarts at the node being backtracked, and proceeds level by level. The branch-and-bound algorithm named **Network_Flow()** is shown as follows:

```
Network_Flow() {
  turn on the policy flag for using Flow-Influence model;
  for (each node n in SDG, arranged by levels) { /* starting from roots to leaves */
    if (any of output arcs of n is marked) {
      mark node n;
      for (each strong input arc i of node n) {
        mark arc i;
        if (arc i is a weakly-influenced output arc of its feeding node k) {
          if (node k has been marked by other node) {
            unmark node k, node n, and all input arcs of node n;
            success = backtrack();
            if (success)
              restart from the backtrack node;
            else {  /* no solution exists under Flow-Influence model */
              turn off the flag for using Flow-Influence model and restart;
              a large cost will be added at the end for this instruction;
            }
          }
          else
            mark node k;
        } /* if */
      } /* for */

      if (node n has weak input arcs) {
        select a most controllable weak input arc;
        if (not successful) {
          unmark node n and all ints input arcs;
          success = backtrack();
          if (success)
            restart from the backtrack node;
```

```
        else { /* no solution exists under Flow-Influence model */
          turn off the flag for using Flow-Influence model and restart;
          a large cost will be added at the end for this instruction;
        }
      }
      else
        put node n into the decision tree;
    } /* if */
  } /* if */
 } /* for */
} /* network_flow() */
```

Since the Flow-Influence model is only used for modelling instruction behavior, it does not veto the entire justification possibility even though it is low. A remedy policy for SDG_j failing in the checking is to augment a very large cost to the instruction indicating a very high probability of a value conflict at a low level. When the process terminates, the marked leaf nodes corresponding to some PSLs form a set which needs to be justified by next instruction, if the instruction *j* is used at the current slot. The cost value of instruction *j* is calculated by accumulating the *Jcosts* of these leaf nodes, plus the large failure cost if it is necessary. The justification process will be applied to all instructions and the one with the least cost will be selected first.

An example illustrating the branch-and-bound checking algorithm is shown in Figure 5. Nodes D, E, F, and H have weakly-influencing input arcs so they are decision points. Nodes G, H, I, and K have weakly-influenced output arcs, which are the possible conflict locations. The example shows that the algorithm explores the justification paths for node A, B, and C with one backtracking.

6. Circuits with HTC State Lines

When an HTC state line or a node affected by an HTC state line needs to be justified, it is very difficult to assemble the instruction sequence, since the instruction sequence depends on the value required at the HTC line. A special instruction sequence is usually needed to first reset the HTC line and push its value to the required one. For instance, the state line in the stack pointer of the Am2910[22] is an HTC line. When the multiplexer in the stack is under test as shown in Figure 6, it turns out to be extremely difficult for a high level test generator to assemble a valid instruction sequence. For the particular test vector injected at the input of the

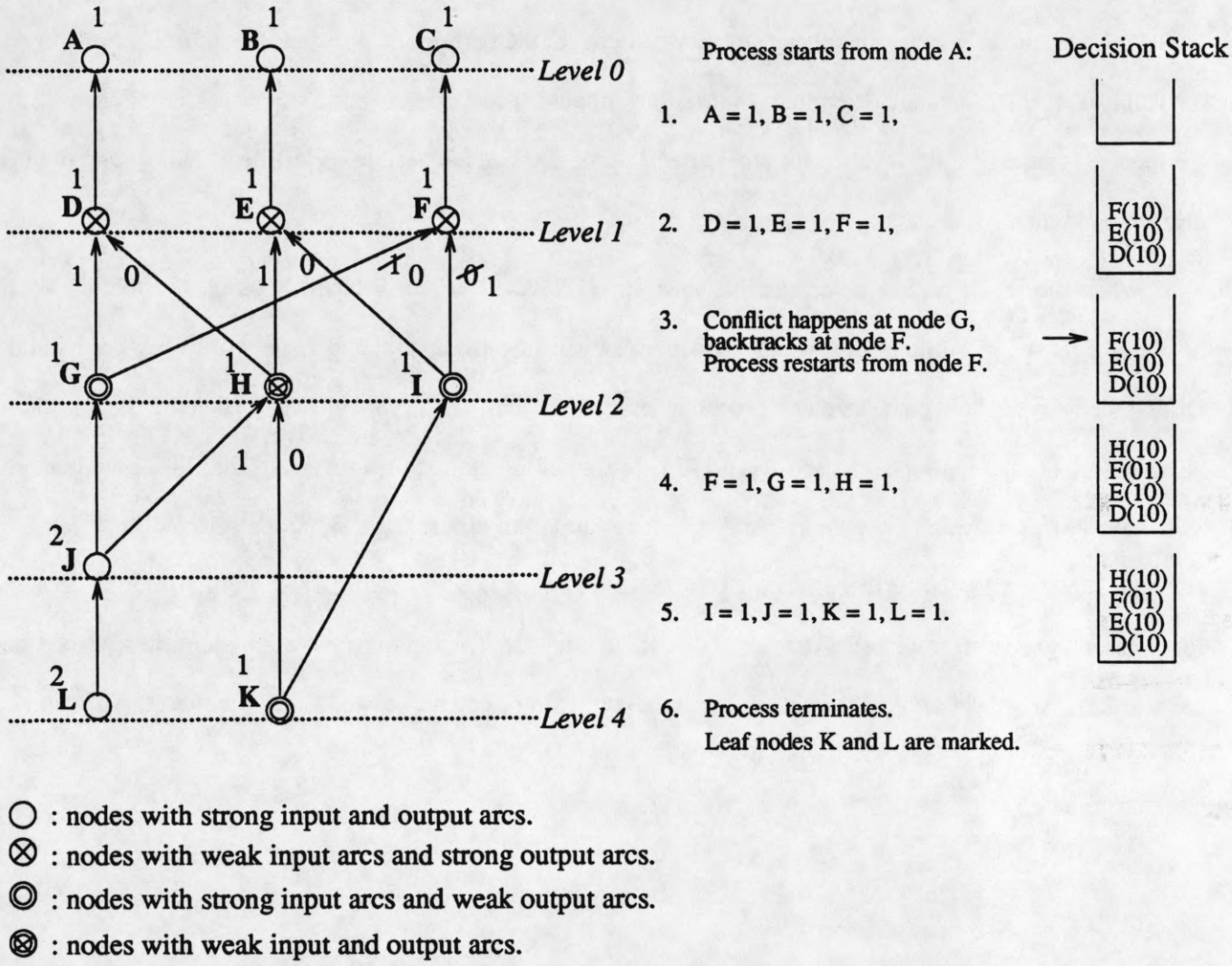

Figure 5. An example illustrating the simultaneous justification checking algorithm.

multiplexer in Figure 6, a possible instruction sequence for justifying it is to reset the stack pointer first. Then five PUSH instructions are applied to store values into all registers, followed by three POP instructions to achieve the value 2 at the select pin. Apparently it is very difficult to automate the tool to search for this sequence without any value analysis on the HTC lines.

All HTC state lines can be identified by the cone analysis in the preprocessing phase. If the circuit under test contains some HTC state lines, an extra value analysis needs to be performed. First the exact value change

produced by each instruction at each HTC line is derived and recorded. If an instruction has no operation on one HTC line, or the exact value change cannot be derived, the corresponding value record is **Unknown**. As the value of an HTC line may determine the data flow graphs for some instructions, it is necessary to investigate the required values of this HTC line for other affected state lines which need to be justified. This information is also stored in the data structure.

When the set of testing objectives includes an HTC state line, its value will be passed through each instruction. Some arcs in the SDG which cannot be activated due to this known value will be marked invalid. During the network flow path searching process, these invalid arcs will never be used. When the cost is calculated after the searching process, each *Jcost* will be weighted by the required values of the HTC state lines so that the instructions which are able to justify the state lines with higher costs would be selected first. For the example shown in Figure 6, those instructions implementing a POP function have a higher priority because Reg5 has a higher weighted cost. After justifying Reg5, the PUSH instructions have a higher priority since the costs of Reg3 and Reg1 attract the value of the stack pointer to be counted down. Finally the reset instruction is applied to reset the stack pointer to be zero.

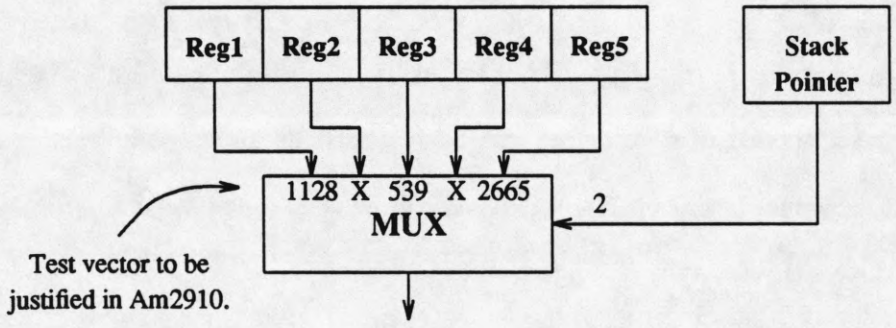

Figure 6. The multiplexer(5-word Stack_CL2) in Stack in the Am2910.

7. Experimental Results

The instruction sequence assembling techniques have been implemented in our hierarchical test generator, ARTEST. The current version of ARTEST contains about 20,000 lines of C codes, and it is able to handle microprocessor-like circuits and circuits with very complex data path configurations. The current implementation of ARTEST contains a high level test generator and a gate level test generator which handles the module test generation. A highly efficient gate level fault simulator, PROOFS[23] is interfaced with ARTEST for collapsing the module level faults. Three circuits are tested in this experiment to show the efficiency of the new techniques. The first one is a 16-bit general purpose microprocessor (MP2) with 9 instructions. The second one is a microprogram sequence controller with the same functional implementation as that of Am2910. As mentioned in the previous section, the stack pointer in the data path significantly enhances the difficulties for high level test generation. However, the new algorithm shows the efficiency in testing this circuit. The third one is division circuit containing 8 instructions. The original design of the devision circuit is not an instruction-based circuit, and it has been redesigned for this experiment. Table 1 shows the circuit information. It should be

circuits	# instr.	primitive	# modules	# gates	# faults
MP2	9	16-bit ALU	1	400	930
		16-bit Incrementer	1	127	311
		4-to-16 Decoder	1	42	130
		2-to-1 Multiplexer	31	3038	4092
		16-to-1 Multiplexer	2	1104	3168
Am2910	25	12-bit Incrementer	1	108	184
		4-to-1 Multiplexer	1	171	294
		RegCnt_CL2†	1	158	367
		Stack Pointer_CL1	1	57	126
		Stack Pointer_CL2	1	56	124
		5-word Stack_CL1††	1	275	620
		2-to-1 Multiplexer	5	370	500
Idivckt	8	16-bit ALU	1	400	930

† RegCnt_CL2 is the combinational part of the register counter.
†† Stack_CL1 is actually a 5-to-1 multiplexer.

Table 1. The circuit information.

noted that all modules in Table 1 are combinational. The sequential modules have been broken into combinational blocks interconnected with sequential elements. This approach has been proved to be very efficient to avoid global functional constraints [24]. The single stuck-at fault model is used in this experiment. Each of the three circuits contains one HTC state line. For instance, the program counter contains an HTC state line in MP2, and the stack pointer in Am2910 creates another one.

The results derived by ARTEST is shown in Table 2. For the modules having more than one outputs like the 4-to-16 decoder in MP2, all outputs are tried to propagate fault effects to ensure the validity of each test vector being injected at the high level. ARTEST derives very high test generation efficiency for most of the modules. For the 5-word Stack_CL1 (multiplexer) in Figure 6, a 100% ATPG efficiency is achieved. This result demonstrates the effectiveness of the instruction sequence assembling algorithm. However, the ATPG efficiencies for the Stack Pointer_CL1 and some 2-to-1 multiplexer in Am2910 are relatively low. By tracing

circuit	primitive	# faults	# det	# drop	# unt	eff.(%)	CPU
MP2	16-bit ALU	930	914	0	16	100.0	492 sec
	16-bit Incrementer	311	311	0	0	100.0	638 sec
	4-to-16 Decoder	130	113	17	0	86.9	101.4 min
	2-to-1 Multiplexer	4092	3774	318	0	92.9	435.8 min
	16-to-1 Multiplexer	3168	3168	0	0	100.0	164.2 min
Am2910	12-bit Incrementer	184	184	0	0	100.0	4 sec
	4-to-1 Multiplexer	294	294	0	0	100.0	40 sec
	RegCnt_CL2	367	322	2	43	99.5	41 sec
	Stack Pointer_CL1	126	52	43	31	65.9	19.4 min
	Stack Pointer_CL2	124	89	1	34	99.2	39 sec
	5-word Stack_CL1	620	525	0	95	100.0	225 sec
	2-to-1 Multiplexer	500	368	132	0	73.6	905 sec
Idiv	16-bit ALU	930	788	6	136	99.4	17 sec

faults = total number of stuck-at faults.
det = total number of faults detected.
drop = total number of faults dropped.
unt = total number of faults proved to be untestable.
eff.(%) = test generator efficiency = (#faults - #drop)/#faults.

Table 2. The results generated by ARTEST.

through the circuit structure, we found that for a high percentage of the injected test vectors, the fault effects cannot be ensured to be propagated at the high level. Since a pessimistic data type manipulation policy is used, a high level fault effect will be lost if not all possible unknown faulty values can be guaranteed to be propagated. That is, this pessimistic policy guarantees the detection of all faults which produce the fault effects that are propagated at the high level. Many faults can be treated as potentially detected by the test sets that ARTEST generates, so the real fault coverage could be much higher than the reported one. The fault coverage for the decoder in MP2 is not high due to the same reason.

8. Conclusions

In this paper, it is proposed to separate the hierarchical test generation into two phases. Each of the two phases applies different techniques to solve major ATPG problems. An instruction sequence assembling algorithm has been introduced at an architectural level. The high level circuit behavior is modelled by a Flow-Influence model on a structural data-flow graph of each assembly instruction. Based on the derived instruction sequence, the previous published relaxation-based algorithm is applied to compute the exact value solutions for all interior buses. Three circuits have been tried and the results show that this new approach is able to handle the circuits with HTC state lines which create extreme difficulties. The future work would be to try some circuits with larger instruction sets.

Acknowledgement

Mr. Yachyang Sun's helpful discussion of the *NP*-complete problems is gratefully acknowledged.

Appendix I

Proof of Theorem 2:

It is obvious that the problem of checking the simultaneous justifiability is in *NP*. Then, we show that the *One-in-Three 3SAT* problem can be reduced to a special case of the problem of checking the simultaneous justifiability. Since the One-in-Three 3SAT problem has been proved to be *NP*-complete[25] , the simultaneous justification problem is therefore concluded to be *NP*-complete. The special case is formalized as follows. It is assumed that each node in SDG_j can have at most 2 weakly-influencing input arcs, and k weakly-influenced

output arcs, where $k \geq 3$. It is also assumed that each node having a weak input(output) arc does not have any strong input(output) arc. For each node with a name, say F, one of its weak input arcs is named F, and the other one is named $\bar{F}$. If this node has no weak inputs, all its output arcs are named F. These names are also treated as variables and can be assigned with logic 0 or 1. The logic 1 means the arc is used to transport water, and a 0 means the arc is blocked. If this node is not involved in the water transportation, the logic value is assigned with "don't care". We only need to concentrate on the nodes located in the dependence cone of S so this condition is not considered.

Now we need to show that, given a boolean expression in the *conjunctive normal form* in the One-in-Three 3SAT problem, there exists a corresponded SDG. An assignment satisfying the expression exactly corresponds to a set of simultaneous justification paths on the SDG, and vice versa. Each *clause* in the expression is mapped to a node, say m, in the SDG with each of its weakly-influenced output arc corresponding to one literal in the clause. Each type of literals, say F and $\bar{F}$, are from another node, say n, with two weakly-influencing input arcs. Each of these two input arcs of n comes from a fanout node if the corresponding literal appears in more than one clause. Therefore, the network can be structured by connecting the inputs of the fanout nodes (feeding into nodes like n) to the outputs of the nodes like m based on the boolean expression. An assignment satisfying the boolean expression determines logic values for all literals which also determines the validity of each path. Since only one literal can have a logic 1 in each clause, a conflict will not happen at the nodes like m. In another words, the path being determined is a valid path. The reverse argument can be directly deduced with a similar manner. Because the One-in-Three 3SAT problem is reduced to the special case of the simultaneous justification problem, the *NP*-completeness of the simultaneous justification problem is concluded. □

References

[1] H-K. T. Ma, S. Devadas, A.R. Newton, and A. Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits," *IEEE Trans. on Computer-Aided Design*, pp. 1081-1093, Oct. 1988.

[2] V.D. Agrawal, K.T. Cheng, and P. Agrawal, "CONTEST: A Concurrent Test Generator for Sequential Circuits," *Proc. 25th IEEE/ACM Design Automation Conference*, June 1988.

[3] T. Niermann and J.H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," *Proc. European Design Automation Conference*, Feb. 1991.

[4] S.J. Chandra and J.H. Patel, "A Hierarchical Approach to Test Vector Generation," *Proc. 24th IEEE/ACM Design Automation Conf.*, pp. 495-501, June 1987.

[5] B. Krishnamurthy, "Hierarchical Test Generation: Can AI Help?," *Internation Test Conf.*, pp. 694-700, Aug. 1987.

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

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

[8] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and A Method," *IBM J. Res. Develop*, vol. Vol. 10, pp. 278-281, 1966.

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

[10] M.H. Schulz, E. Trishler, and T.M. Sarfert, "SOCRATES: A highly Efficientc Automatic Test Pattern Generation System," *International Test Conf.*, pp. 1016-1026, Sept. 1987.

[11] T.M. Sarfert, R. Markgraf, E. Trischler, and M.H. Schulz, "Hierarchical Test Pattern Generation Based on High-level Primitives," *Proc. International Test Conf.*, pp. 470-479, Sept. 1989.

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

[13] B.T. Murray and J.P. Hayes, "Hierarchical Test Generation Using Precomputed Tests for Modules," *Proc. International Test Conf.*, pp. 221-229, 1988.

[14] Jaushin Lee and Janak H. Patel, "An Architectural Level Test Generator for a Hierarchical Design Environment," *Proc. 21th Symposium on Fault-Tolerant Computing*, pp. 44-51, June 1991.

[15] D. Brahme and J.A. Abraham, "Functional Testing of Microprocessors," *IEEE Trans. on Computers*, vol. C-33, pp. 475-485, 1984.

[16] K. Roy and J.A. Abraham, "High Level Test Generation Using Data Flow Descriptions," *Proc. IEEE European Design Automation Conf.*, pp. 480-484, 1990.

[17] C. Wu, "Test Generation for High Level Synthesis Circuits," *Master Thesis, Dept. of Computer Science at University of Illinois*, 1991.

[18] C. Chen, C. Wu, and D.G. Saab, "Beta: Behavioral Testability Analysis," *Proc. IEEE Int. Conf. on Computer-Aided Design*, pp. 202-205, Nov. 1991.

[19] Jaushin Lee and Janak H. Patel, "A Signal-driven Discrete Relaxation Technique for Architectural Level Test Generation," *Proc. IEEE Int. Conf. on Computer-Aided Design*, pp. 458-461, Nov. 1991.

[20] Vivek Chickermane, Jaushin Lee, and Janak H. Patel, "Design for Testability Using Architectural Descriptions," *Proc. International Test Conference (to appear)*, 1992.

[21] L.H. Goldstein, "ControllabilityObservability Analysis of Digital Circuits," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, NO. 9, pp. 685-693, Sept. 1979.

[22] Advanced Micro Devices, in *The Am2910, A Complete 12-bit Microprogram Sequence Controller*.

[23] T. Niermann, W.T. Cheng, and J.H. Patel, "PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator," *IEEE Trans. on Computer-Aided Design*, vol. Vol. 11, No. 2, pp. 198-207, Feb. 1992.

[24] Jaushin Lee and Janak H. Patel, "Hierarchical Test Generation under Intensive Global Functional Constraints," *Proc. 29th ACM/IEEE Design Automation Conf.*, June 1992.

[25] M.R. Garey and D.S. Johnson, "Computers and Intractability - A Guide to the Theory of NP-Completeness," *Freeman Press*, p. 259, San Francisco, CA, 1979.