A Graph Compaction Approach to Fault Simulation.

Dov Harel
Balakrishnan Krishnamurthy

Tektronix Laboratories
Tektronix, Inc.
Beaverton, Oregon 97077

Abstract: We describe a graph compaction based algorithm for fault simulation in combinational circuits. The algorithm consists of reducing the circuit graph by repeatedly removing non-reconvergent vertices. The algorithm have been implemented in Smalltalk and preliminary experimental results are presented. A version of the algorithm outperforms all known fault simulation algorithms on a family of hard circuits.

1. Introduction

It is well known that the major obstacle for fast fault simulation is reconvergence in the underlying circuit graph. In fact it was shown [7] that fault simulation is at least as hard as matrix multiplication and related problems for which there are no known linear time algorithms. In view of these facts, it would be interesting to find algorithms whose execution time is inversely proportional to the amount of nested reconvergence in the circuit. The hope is that analysis of the topology of the circuit may be helpful in attaining such a goal. Certain steps in this direction were taken in [1, 12] and more recently improved by [3]. Subsequently more general approaches utilizing dominators in flow graphs [10], stem regions and dominating sets [13, 6] were proposed.

This paper presents a radically different approach to solving the problem which we call graph compaction. The new approach, coupled with some of the above results, leads to improved algorithms in many situations. A complete analysis of the performance of a variant of this algorithm in the case of bounded degree series parallel graphs and matrix multiplication decoders is beyond the scope of this paper and can be found in [9].

In this paper we concentrate on the following problem. Given a combinational circuit C and a test vector, find for each gate g (logical gate or a fanout stem) the set of all primary outputs of C at which the stuck-at fault at the gate will be detected. We call this set the *detection set* of g. The reader is cautioned that the above problem is different from what is usually referred to as fault simulation. Detection set computation is closely related to single vector fault simulation, and in fact subsumes it.

We note that fault simulation is closely related to data flow analysis problems which arise in the context of code optimization [2] and most known fault simulation algorithms have a counterpart in that domain. The graph compaction algorithm presented here is reminiscent of the Hecht and Ullman graph transformation based flow graph manipulation algorithms in code optimization [11].

The ideas in this paper shed some light on the reconvergence structure of directed acyclic graphs and its relationship to fault simulation. In particular a version of the algorithm [9] outperforms all previously published algorithms on the matrix multiplication decoders of [7].

In section 2 the basic algorithms are presented informally. Section 3 contains pseudo code for a version of the algorithm and briefly discusses reconvergence precomputation. Section 4 contains an application for built in self test (BIST) and section 5 contains preliminary experimental results of a Smalltalk implementation.

2. Graph Transformations

The algorithm is motivated by the following observation: Given the detection sets for all the reconvergent stems in a combinational circuit, one can compute the detection sets everywhere in a linear time backwards walk of the circuit. We therefore concentrate on the reconvergent stems and gates by deleting all non-reconvergent vertices from the circuit. After that phase some previously reconvergent vertices might not reconverge any more (see example in Figure 1). If so, the previous process can be repeated. A more precise exposition follows.

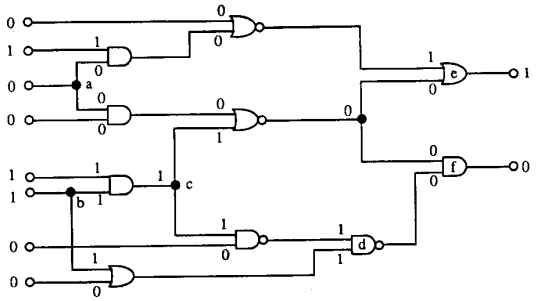
Let G be a *dag* (directed acyclic graph); G^* denotes the transitive closure of G. We call a vertex v in G *forward reconvergent* if v has two immediate successors $x \neq y$, x and y reach a common vertex u in G. We call a vertex v in G *backward reconvergent* if v has two predecessors $x \neq y$, x and y reached by a common vertex u in G. Finally v is a *reconvergent* vertex of G if v is either a forward or a backward reconvergent vertex of G.

Let $G = G_0$ be a dag corresponding to a combinational circuit. Given a test vector x we define a sequence $G_1 G_2, \ldots, G_k$ of subgraphs of G^*, $G_i = (V_i, E_i)$ as follows. Define V_1 to be the set of reconvergent vertices of G_0, and $E_1 = \{ (x, y) \mid x, y \in V_1$ and there is a sensitized path $x = x_0, x_1, \cdots, x_n = y$ in G_0 such that the effects of a single stuck-at fault at the output of x propagate through the path to the respective inputs of y, and such that for $0 < i < n, x_i$ is not reconvergent in $G_0 \}$. In other words, G_1 is obtained from G_0 by removing the non-reconvergent vertices of G_0, and adding in edges which designate single path sensitization in G_0. In this case the effect of a single fault can effect at most one input of the target gate, since all the intermediate vertices on the propagation path are non-reconvergent.

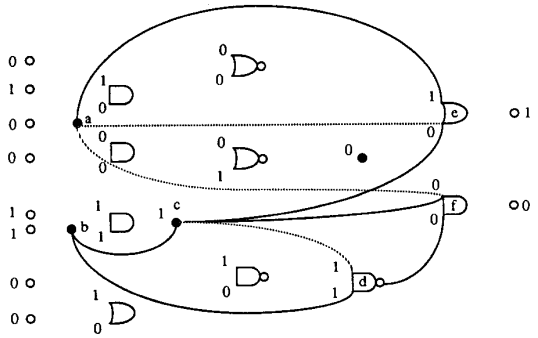
Note that a vertex might be forward (backward) reconvergent in G_0 and not in G_1. Thus it is entirely possible that G_1 further reduces to G_2, and so on. We do not consider multiple edges in G_1 in the sense that if there are paths from a node x to different inputs of a node y, they are considered to constitute a single edge, although the effect of the edge is registered with the respective input slots of y. In general a node x in G_1 (G_i) carries with it in addition to the logical values of its input slots, the set of inputs which have changed due to effects propagating through predecessors that by now have been removed from the graph. An edge in G_1 is assumed to carry all single fault effects propagating from its tail to its head.

In general E_{i+1} is obtained from G_i in the same way E_1 is obtained from G_0 as follows. V_{i+1} is all non-reconvergent vertices of V_i, and $E_{i+1} = \{ (x, y) \mid x, y \in V_{i+1}$ and there is a path from x to y in G_i such that for every node z on the path (except for the endpoints),

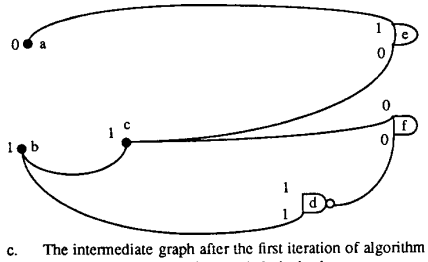
z is non-reconvergent and the fault effect at x propagates through z}. Henceforth we will refer to the above graph compaction procedure as algorithm A. Figure 1 contains a detailed example of executing algorithm A on a simple circuit.



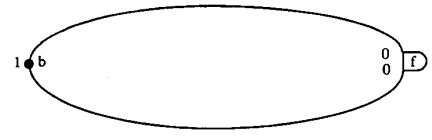
a. A circuit and a test vector after logical simulation.



b. After the first iteration in algorithm A all non-reconvergent gates and stems are removed (shown here isolated). Solid lines represent fault propagating (sensitized) paths through non-reconvergent nodes of the circuit, dashed lines represent non-propagating such paths. Note that although there are paths from stem a to respective inputs of gates e and f, both paths go through an immediate successor of a which is an *AND* gate with a negative (0) output value, and thus the a stuck-at-1 fault does not propagate beyond that successor.



c. The intermediate graph after the first iteration of algorithm A. Note that isolated vertices and dashed edges were removed from the previous diagram.



d. The intermediate graph after the second (and next to last) iteration of algorithm A.

Figure 1. Execution of algorithm A on a simple circuit.

Note that it is possible that $G_{i+1} = G_i$ and G_i may or may not be the empty graph $\emptyset$. If at some point i in the sequence of reductions we have $G_{i+1} = G_i \neq \emptyset$ we say that the original graph G is *irreducible* under the given test vector. In such situations we could use a variation of any known fault simulation algorithms to compute detectability in G_i, and then in linear time complete the computation for the original circuit G. An alternative approach which we will call algorithm A1 is outlined below.

While computing G_{i+1} from G_i we restrict our attention to paths in G_i all whose internal nodes are non-reconvergent in G_i. The intuitive reason is that if there are multiple paths from the head of the path into some gate g on the path, then it may seem as if the effect propagates through g while in fact it will not propagate through g due to multiple path desensitization. This can not happen, however, if g is a fanout stem and so one can propagate a fault through a fanout stem whether or not it is reconvergent! In other words, one can allow forward reconvergent nodes on the path from x to y in the definition of E_{i+1}.

More precisely, we modify the definition, computing the sequence $G = G'_0 G'_1 \cdots G'_k$, V'_{i+1} is the set of non-reconvergent vertices in V_i, and $E_{i+1} = \{ (x, y) \mid x, y \in V_{i+1}$ and there is a path from x to y in G_i such that for every node z on the path (except for the endpoints), z is not *backward-reconvergent* and the fault effect at x propagates through z}. For the new sequence we can prove that the original graph G will always collapses. That is, as long as G'_i is not empty, $G'_i \neq G'_{i+1}$. In fact we can show that the reduction chain $G = G'_0 G'_1 \cdots G'_k$ terminates, and its length is bounded by the depth of the circuit. This follows from the following lemma whose proof is omitted from this version.

Lemma 1: The depth of G'_{i+1} is at least one smaller than that of G'_i.

3. The Algorithm

An outline of algorithm A1 is given below. The procedure *propagateForward* below computes G_{i+1} from G_i. The procedure *graphCompaction* repeatedly calls *propagateForward* until either the current graph is empty or there is no change. Finally *compute_detection_sets* recovers the detection sets of g from the detection sets of its successors in drop(g) in a single backwards walk (in topological order) of the graph. Recall that g's successors at the time g dropped out of G_i.

Note that an implementation of algorithm A can easily be obtained from the above pseudo code as follows. In the procedure *propagateForward* do not treat fanout stems of the original circuit as a special case (i.e. replace the condition **if** s is a backward reconvergent node of G_i **then** by **if** s is a reconvergent node of G_i **then**). In addition it is now necessary to check in *graphCompaction* for the case that the graph is not reducible that is: $G_i = G_{i+1} \neq \emptyset$, in which case it is necessary to perform fault simulation on the irreducible graph (with respect to the test vector) G_i before applying *compute_detection_sets*.

We have programmed several variations of the graph compaction algorithm. Some of the variations allow different kinds of intermediate vertices on the paths represented by edges in the compacted graph. Some take into consideration independent propagation through some gates [4]. By far the most effective speed-up was gained by avoiding recomputation of reconvergence by preprocessing the graph as follows. Perform a preprocessing compaction of the graph making worst case assumptions about fault propagation through gates, that is: whenever a fault effect may propagate through a gate we assume that it does. The variation hence referred to as algorithm B performs this precomputation and

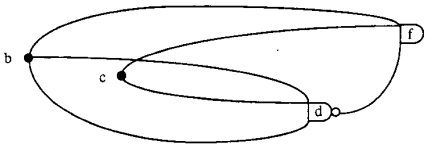
```
procedure propagateForward(i, V)
"backwards walk of G_i modify edges creating G_{i+1}"
    for j := |V_i| down to 1 do
        g := the j-th topological order gate of G_i
        if g is a non-reconvergent node of G_i then
            copy the successor set of g into the set drop(g);
            add g to V_{i+1}; fi
        new_successors := the empty set;
        for each successor s of g do
            if s is a backward reconvergent node of G_i then
                add s to new_successors;
            else if effect propagates from g through s then
                for each successor t of s do
                    add a copy of t (including effects
                        on t's input) to new_successors;
                od fi fi
        od od
    V := V_{i+1};
end {propagateForward}

procedure graphCompaction
"compact the graph, return number of iterations"
    i := 1; V' := V;
    while V' is not empty do
        propagateForward(i, V');
        comp_forward_reconvergence(G_i);
        comp_backward_reconvergence(G_i);
        i := i+1; od
    compute_detection_sets;
    return i;
end {graphCompaction}

procedure compute_detection_sets
for i := n down to 1 do
    v := the i-th vertex in topological order;
    detection_set(v) := union { detection_set(u) | u in drop(v)
        and the fault effect at v propagates through u };
od
end {compute_detection_sets}
```

remembers the reconvergence values computed during each iteration. The graph in Figure 1.b is the result of the first iteration of the preprocessing phase on the circuit of Figure 1.a. Here all lines (dashed and solid) are the edges of the graph. Note that this graph is independent of the chosen test vector. The result of the second and third iterations are given in Figure 2.

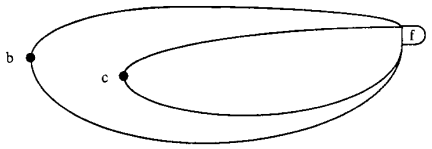
It can be shown that it is safe to use these precomputed reconvergence values in the test vector dependent computation. Another variation which gave a slight improvement, and henceforth will be referred to as algorithm B1, also records which vertices during the backwards walk of the graphs could be ignored since all their immediate successors were backward reconvergent during a particular iteration. Experimental results obtained by execution of algorithms B and B1 on the benchmark circuits of [5] are given in table 2.

4. Application for Built-In Self Testing

The graph compaction approach is particularly attractive in situations where it is desirable to compute the detection set of faults, rather than just whether or not a single stuck-fault is detectable. One example of such a situation occurs is when creating fault dictionaries. Another application is for built-in self testing. Assume we have some random number generator, typically an LFSR, implemented on the same chip to facilitate self testing of the chip. Each random input vector is plugged as input into the circuit under test, and the output vector is "mixed into" a signature producer, typically an LFSR again. The usual way a BIST scheme like the one described above works is by feeding a single many test vectors produced in succession by the input LFSR. The signature thus obtained consists of as many bits as there are primary outputs.



a. The intermediate graph after the second preprocessing iteration of algorithm B on the circuit of Figure 1.a. The intermediate graph after the first preprocessing iteration is given in Figure 1.b.



b. The intermediate graph after the third preprocessing iteration of algorithm B on the circuit of Figure 1.a. The intermediate graph after the fourth iteration is identical to the graph of Figure 1.d.

Figure 2. Execution of the preprocessing phase of algorithm B on a simple circuit.

Computation of the fault coverage of the above scheme, and even its estimation is hard in general. One way to compute the coverage exactly is to compute the detection set of every fault at every phase, eventually producing a signature for every single stuck at fault. Note that our method has advantages in this case for the following two reasons. First, it computes the detection sets rather than just detecting faults. In addition, it takes advantage of the topology of the circuit rather than utilizing fault dropping which is not permissible under the above model. Finally, the large number of test vectors simulated seems to warrant the extra cost in the complexity of the algorithm.

5. Experimental Results

We have implemented a prototype of the algorithms described above in Smalltalk 80 on a Tektronix 4406 work station. We have also implemented single fault propagation algorithms not using dominators [10] in the same environment. The most effective algorithms we tested were algorithms B and B1 which precompute reconvergence. A table comparing execution times of the algorithms appears in the complete version of the paper (see [8]) and is omitted from this version. That table indicates that algorithm B1 outperforms single fault propagation by a factor ranging from 1.5 to 3.2. However, since similar speed-up of single fault propagation is possible using dominators and other techniques [3, 10]. these results are inconclusive.

Table 2 below gives the number of iterations required to collapse the circuits, as well as statistics on the sizes of the first two graphs collapsed (and their relevant subsets), and the total sizes of all subgraphs over all iterations. The things to notice are the relative low number of iterations and the dramatic drop in the number of vertices after the first iteration. This suggests that the first iteration alone might be useful in practice for any fault simulation algorithm. That is the algorithm would be employed on the graph

obtained by the first iteration of algorithm B, and then the results for the original graph can be recovered in linear time.

Circuit Sizes				
Circuit	I	O	E	V
alu181	14	8	323	209
c432	36	7	432	249
c499	41	32	499	261
c880	60	26	880	508
c1355	41	32	1355	805
c1908	33	25	1908	1265
c2670	157	64	2594	1647
c3540	50	22	3540	2248
c5315	178	123	5315	3113
c7552	206	107	7551	4812

Table 1. Sizes of benchmark circuits. Here I and O represent the number of primary inputs and outputs of the circuit respectively, while V and E represent the numbers of vertices (gates and fanout stems) and edges (leads) respectively.

Intermediate Subgraph Sizes								
Circuit	K	Size	V1	E1	V2	E2	total_V	total_E
alu181	7	total	231	323	125	347	799	2834
		actual	208	297	45	100	660	2501
c432	11	total	292	432	174	886	1678	13615
		actual	218	292	46	638	1201	12548
c499	6	total	334	499	131	1312	886	6621
		actual	294	459	89	1240	732	6413
c880	15	total	594	880	208	665	2117	10511
		actual	531	791	118	493	1412	8562
c1355	10	total	878	1355	643	1392	3916	17197
		actual	526	867	417	832	2198	14045
c1908	14	total	1323	1907	611	1672	5030	53794
		actual	1208	1597	353	821	3497	46265
c2670	17	total	1868	2593	752	2064	6105	24902
		actual	1649	2312	372	1269	4661	22160
c3540	30	total	2320	3537	1128	4312	21888	201735
		actual	1888	2919	252	1883	12577	177331
c5315	25	total	3414	5315	1438	4104	14006	102830
		actual	2962	4776	515	1862	8699	93199
c7552	26	total	5125	7551	2788	8404	29731	243142
		actual	4017	6189	383	2466	16755	218108

Table 2. Sizes of intermediate subgraphs in algorithms 5 and 6. K represents the number of iteration required to reduce the graphs. V1 and E1 represent the number of vertices and edges after the first iteration respectively. The total field refer to all vertices (edges) in the subgraphs obtained during a run of algorithm B, while the actual field represents the number of relevant vertices (edges) actually scanned by algorithm B1.

6. Conclusion

We have presented a single vector fault detection algorithm based on graph compaction. A variation of the algorithm collapses a circuit graph in a test vector independent way. The importance of the algorithm is in identifying some subtle properties of circuit graphs which makes them more or less amenable to efficient fault simulation. For example we were able to show that a version of the algorithm coupled with stem regions [13, 6] reduces the time complexity of both hard [7] and easy (bounded degree series parallel) circuits.

In addition table 2 shows that the first iteration is particularly effective in reducing the size of the simulated circuit that it may well be worth considering as a preprocessing stem in other fault simulation methods.

Acknowledgement

The authors wish to thank Mark Friedman for enlightning discussions concerning the material in this paper and for carefully reviewing a previous version of the paper.

References

1. Abramovici, M. , Menon, P.R., and Miller, D.T., "Critical Path Tracing - An Alternative to Fault Simulation," in *Proceedings of the 20th Design Automation Conference* , pp. 214-220, 1983.
2. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers - Principles, Techniques, and Tools*, Addison Wesley, Reading Massachusetts, 1986.
3. Antreich, K.J. and Schulz, M.H., "Fast Fault Simulation in Combinational Circuits," in *Proceedings of the International Conference on Comp. Aided Design*, pp. 330-334, 1986.
4. Bhattacharya, B.B. and Seth, S.C., Fault Simulation in Combinational Circuits, unpublished manuscript, 1986.
5. Brglez, F. and Fujiwara, H., "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran,," in *Proc. of IEEE Int. Symp. on Circuits and Systems, (Special Session on ATPG and Fault Simulation)*, June, 1985.
6. Friedman, M., Harel, D., Maamari, F., and Rajski, J., "A Dominators View of Stem Regions in Combinational Logic and its Application to Fault Simulation", *Technical Report 87-50*, Computer Research Laboratory, Tektronix Laboratories 1987.
7. Harel, D. and Krishnamurthy, B., "Is There Hope for Linear Time Fault Simulation?," in *Proceedings of the 17th FTCS Symposium*, pp. 28-33, July, 1987.
8. Harel, D. and Krishnamurthy, B., "A Graph Compaction Approach to Fault Simulation", *Technical Report 87-46*, Computer Research Laboratory, Tektronix Laboratories, October 1987.
9. Harel, D. and Krishnamurthy, B., "Efficient Graph Transformation Based Fault Simulation", *Technical Report 87-55*, Computer Research Laboratory, Tektronix Laboratories, November 1987.
10. Harel, D., Sheng, R., and Udell, J., "Efficient Single Fault Propagation in Combinational Circuits," in *Proceeding of ICCAD 1987*, pp. 2-5, November 1987.
11. Hecht, M.S. and Ullman, J.D., "Characterizations of Reducible Flow Graphs," *Journal of the Association for Computing Machinery*, vol. 21, pp. 367-375, 1974 .
12. Hong, S.J., "Fault Simulation Strategy for Combinational Logic Networks," in *Proc. of 8th International Symp. on Fault Tolerant Computing*, pp. 96-99, June, 1978.
13. Maamari, F. and Rajski, J., "Reconvergent Fanout Analysis and Fault Simulation Complexity of Combinational Circuits", *Technical Report 87-3R*, VLSI Design Laboratory, McGill University, August, 1987. To appear *FTCS* 1988.

Automatic Functional Test Program Generation for Microprocessors

Chen-Shang Lin and Hong-Fa Ho

Department of Electrical Engineering
National Taiwan University
Taipei 10764, Taiwan

ABSTRACT

A new algorithm, O-algorithm, for automatic test program generation of microprocessors in a user environment is presented. Specifically, to eliminate the redundant tests, a weighted-digraph model is used to model the signal flow of the general microprocessors. Improved functional fault models of microprocessors are derived from Turing machine model. The O-algorithm is then constructed based on the signal flow model and functional fault models. The complexity of our algorithm is better than [6]. Moreover, the simulation had shown that the fault coverage is better than 97%.

1.INTRODUCTION

Microprocessors are extremely versatile and are hence widely used in many complex systems. However, the test of microprocessors is a nontrivial problem. Especially in the user environment, the test of microprocessors is complicated by the fact that the detail circuit information is not available. As a result, the classical gate-level test generation methods simply can not be applied in this situation. A more feasible approach is to generate test program based on the functional-level information which is available to the users.

Several algorithms [2-6] had been developed to generate test program for microprocessors in the functional level. Thatte, Brahme and Abraham [2,6] proposed a graph model for microprocessors at the register transfer level. However, the fault models were restricted to data path and its associated control function.

In this paper, a new algorithm, O-algorithm, for automatic test program generation of microprocessors is proposed. Specifically, to eliminate the redundant tests, a weighted-digraph model is used to model the signal flow of the general microprocessors. Improved functional fault models of microprocessors, such as those of register decoding function, data register function, and I/O pin function are derived from Turing machine model. These fault models cover more faults than [2,6]. The O-algorithm is then constructed based on the signal flow model and functional fault models. The complexity of our algorithm is better than [6].

In the next section, the signal flow model of microprocessors is described. The improved functional fault models are discussed in Section 3. Based on the signal flow model and fault models, a new O-algorithm for generating test program is then developed. The O-algorithm is described in Section 4 and its complexity is briefly discussed in Section 5. Then the configuration of the automatic test program system based on O-algorithm is described and the applications of the system on MCS8048 and a subset of Intel 8086 are shown in Section 6.

2.SIGNAL FLOW MODEL FOR MICROPROCESSORS

In order to test microprocessors systematically, the signal flow model for microprocessors must first be established. Let RS = {R1,R2,...,Rn} denote the set of distinct registers in a microprocessor. RS does not include on-chip memory such as RAM, cache memory or program ROM. Let IS = {I1,I2,...,Ip} denote the set of distinct instructions of a microprocessor. The signal flow of a microprocessor can be modeled as a weighted digraph G = (V,E,g), where V is the set of vertices (or nodes) comprised by the set RS , the vertice IN for input ports , and the vertice OUT for output ports; E is the set of edges in G and E = {e | e is an information flow of instruction Ii between nodes, for all Ii in IS }; and the function g in G is a weighted function from V to the set of pairs of integers which will be defined latter. In other words, the vertices in G are registers in a microprocessor and the edges (or links) stand for the information flow including data flows and/or address flow of instructions among registers.

An instruction Ii in IS is a set of paths in G. S(Ii) denotes the set of source vertice(s) of instruction Ii, and D(Ii) denotes the set of sink (destination) vertice(s) of instruction Ii. All instructions in IS are classified into three types: type -T for data Transfer instructions, type-B for Branch instructions, and type-M for data Manipulation instructions[2].

Let P(u,v) be the set of directed paths from node u to node v in G, and p(u,v) be a path in P(u,v).

DEFINITION 1: Let Ii be of type-T or type-B. The function NI(p(w,v)) is the number of distinct Ii in which at least one connected path of Ii are in p(w,v).

Notation < I1,I2,...,Ii,Ij,...,Ia > denotes that the instructions will be executed sequentially. Notation [S1|S2|...|Sm] denotes that microinstructions S1, S2,...,Sm are executed concurrently.

DEFINITION 2: Controllability of Vi is CY(Vi) = min[NI(p(IN,Vi))], for all p(IN,Vi) in P(IN,Vi).

DEFINITION 3: Observability of Vi is OY(Vi) = min[NI(p(Vi,OUT))], for all p(Vi,OUT) in P(Vi,OUT).

For example, if there exist three instructions, Ix:"MOV Rx<-Ry" , Iy:"MOV Ry<-#d", and Iz:"PUSH Rx" only, then < Iy,Ix > is the only way to control the state (or value) of register Rx. Hence P(IN,Rx) = { <Iy,Ix> }, and <Iy,Ix> is the only element in