PathFinder

(Beta version 0.9)

• *Introduction:*

PathFinder is a tool used to find the all the paths exercised in the circuit, the key idea is that can we get these paths exercised by the signal without having to simulate the circuit with the given pattern. ***PathFinder*** is a static analysis tool finds all paths in the circuit between any two pin's in the circuit by default, but we can restrict ***PathFinder*** to find all the paths through the circuit lines which we are interested, this makes ***PathFinder*** (referred as PF) flexible to be adapted to any situation of interest.

In this work we describe how we can make use of PF can be used for ATPG (Automatic Test Pattern Generation) path delay analysis, since in any circuit we have exponential number of paths, modeling paths using ***PDF*** (path delay fault model) does not scale that well for huge designs, we use PF in such situation to qualify the patterns on size of path the pattern can exercised thus we can prefer patterns which can test long paths with the patterns. In the next section we describe the implementation details of the PF.

• *Algorithmic Details*

The algorithm PF uses is a variation of the standard BFS(Breadth First Search) the only change is that PF algorithm enumerates all the paths between two nodes in the graph, where as BFS can only find if the two nodes in the graph are connected or not. The following are some of the definitions before we go further into the algorithm. We define the ***path*** as a sequence of pins ($p_1,p_2,p_3....p_n$) in which a pin pair (p_i, p_{i+1}) when *'i'* is odd belong to the same ***net*** **(N)** if *'i'* is even then the pin pair (p_i, p_{i+1}) belong input and output pins of a ***block(B).*** And we are interested in only such paths in which each of these pins p_i belong to the transition fault list reported by the ATPG. We describe two algorithms ***ALGO1*** and ***ALGO2*** former is a non-recursive version and the latter is a recursive algorithm.

Problem Definition: Given a sequence of pins ($p_1,p_2,p_3....p_n$) given by the ATPG as transition fault list for a given pattern report path(s) P_1, P_2..... P_n (path as defined above) such that each of these paths begin and end either at a primary input ***(PI)*** or memory block or at a primary output ***(PO).***

ALGO1: A very simple algorithm such as a Breadth First traversal of the Graph (Hyper Graph) in our case the Hyper Graph is a structural verilog netlist, the only change in the BFS algorithm is that we should expand a node (in this case a pin and sub-circuit connected to that pin) only if the pin is in the transition fault list reported by the ATPG, thus at the end of the BFS we are left with all the possible walks in the graph which contain the pins (transition fault list pins) reported by the ATPG report. One important observation is that we should expand the nodes in the ***Topological Order*** , this step is essential because we don't to miss any paths between two nodes. Below is the ***ALGO1*** briefly described.

ALGO1:

```
FindAllPaths(Graph G, Node n1, Node n2){
              List expand_list = List(n1);
              Node n;
              while(expand_list){
                    n = RemoveFirstElement(expand_list);
                    UpdatePathsToAllIncidentNodes(n,G);
                    AppedToListInTopologicalOrder(n,expand_list);
              }
/*At this point n2 contains all the paths from n1 to n2*/
}
```

You can find the details of the algorithm in the **path_algorithm.c** file in the PF source code. The variation of BFS are the functions **UpdatePathsToAllIncidentNodes** and **AppedToListInTopologicalOrder** , for every edge (u,v_i) incident on u **UpdatePathsToAllIncidentNodes** add's to v_i all the computed paths to u with edge (u,v_i) appended to each of these paths, in simple words if there is a edge between (u,v) all the ways to reach v is all the ways to reach u and for each of these ways add the edge (u,v). Since we don't want to miss any of the paths we need to be careful on the order of the nodes which we process **AppendToListInTopologicalOder** takes care of this issue, see the following example on the need of **Topological Ordering.**

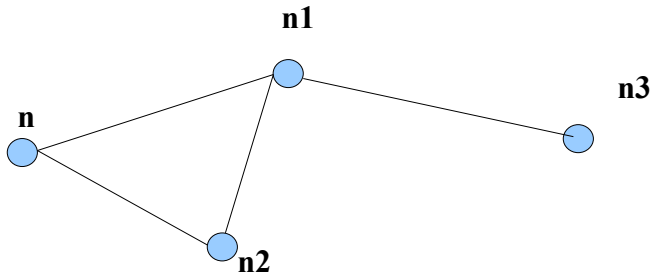

Fig2: Need of Topological Sort

As we can see in the above situation if we are expanding node **n** during the BFS processing step we can process **n1** or **n2** since both are in the same level with respect to **n** but see the important fact if we expand **n1** first rather than **n2** we are going to miss out path through **n2** at the end of BFS when we reach **n3.** This is exactly **Topological Sort** comes into our rescue, since out circuit is a DAG (Directed Acyclic Graph), ordering the nodes topologically forces us to process **n2** before **n1** this is taken by **AppendToListInTopologicalOrder.**

ALGO2: Another simple algorithm which saves the space in terms of the intermediate paths we need to keep track, is a simple recursive algorithm which marks a node and goes into recursion and unmarks when it comes back of the recursion. Please see path_algorithm.c for more details.

- ***Architecture of the System***

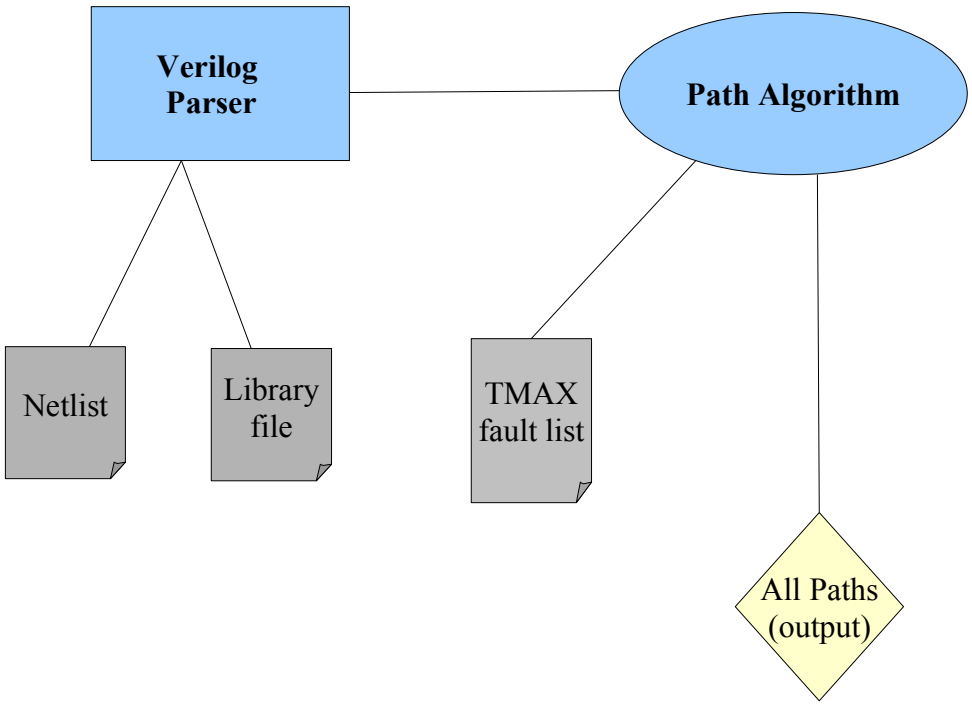

Fig3: Overview of Pathfinder system.

- ***Results***

Design	PAT:1	PAT:2	PAT:3	PAT:4	PAT:5	PAT:6	PAT:7	PAT:8	PAT:9
s298	5	5	5	0	4	7	6	5	7
s1196	6	7	10	0	4	7	6	10	10
s9234	10	6	12	14	7	11	10	12	10
s13207	19	14	18	14	11	14	22	9	20
s15850	14	16	11	12	13	14	10	13	9
s38417	22	22	15	17	19	22	22	14	22
s38584	18	19	21	26	17	17	20	22	22

Table 1: longest paths and patterns