A System to Detect Timing Problems in Digital Circuits

Esteban Pelaez, Mario Berón, Carlos Salgado, Mario Peralta, Lorena Baigorria, Ana Garis, Germán Montejano, Daniel Riesco, and Pedro Henriques

Universidad Nacional de San Luis, Informatic Department, Ejército de los Andes 950
San Luis, Argentina
{epelaez,mberon,csalgado,mperalta,flbaigor,agaris,gmonte,driesco}@unsl.edu.ar
Universidade do Minho, Informatic Department,Campus de Gualtar 4710-057, Braga, Portugal
pedrorangelhenriques@gmail.com
http://www.unsl.edu.ar

Abstract. Nowadays, the digital circuit production is carried out specifying the circuit functionality using a hardware description language. Then, this specification is synthesized down to a structural netlist suitable for use by the target technologys place-and-route applications. Many synthesis tools make this task introducing some unnecessary gates and wires in the final circuit. As a consequence, it can appear a circuit containing one or more paths that do not influence the circuit output. This kind of non-relevant paths is known as *False Path*.
The problem with false paths is that if they are not considered, the circuit delay may be overestimated during design analysis and optimization. For this reason, the digital circuit industry is looking for effective methods and tools to overcome the mentioned drawbacks.
This paper presents a system to detect *False Paths* based on the analysis of the circuit intermediate specification. The tool analyzes the specification using compilation techniques and then applies some special purpose algorithms for detecting false paths. Furthermore, it shows the gates and wires that are not necessary for the circuit final version.

Keywords: False Path, Syntactic Analysis, Semantic Analysis, Hardware Description Languages, Intermediate Language

1 Introduction

Static time analysis establishes an industry standard in the design methodology to calibrate the speed of high performance microprocessors [jLKWtC02]. Unfortunately, not all paths identified by this type of analysis can be sensitive to them. This leads to a pessimistic estimation of the processor speed, and the dedicated engineering efforts to optimize such paths can not improve chip performance [ZAB^{+}01,KB99].

A false path is a path whose time delay can not affect a change in the output signal [DYG89,CCW$^+$07]. The false path information can be used to reduce the time required for the tests generation and application, and the circuits area, whereas also it reduces to the minimum on-test (Over-testing) [PCDM89]. This information is valuable in the design and circuits testing.

From the design point of view, the constraints on the false paths can be ignored, so designers can replace the gates in false paths by smaller gates with a larger delay. In addition, the paths optimization (For those paths larger than the critical path) can be skipped if paths are identified as paths that does not have to meet design constraints [IOF]. Therefore, the circuit area and the time required for logic sensitized can be decreased by using the false paths information.

A test pattern for delay failures over false path can not be generated, so the early identification of such false path may reduce significantly the ATPG (Automatic Test Pattern Generation) time. Moreover, the false path identification can alleviate the fact that some delay failures over false path may become testable due to the design testability application.

In a circuit with delay faults, however, the false path delay may interfere with the test of other paths. Some paths contain false redundant logic. Since the clock speed is often determined in the static time analysis, the presence of false paths can take to a pessimistic operation (Slow). For these reasons, the false paths removal in combinational logic is advantageous [NP02]. This means that false paths detection and elimination in a circuit contributes to the performance improvement and circuit size. This will result in a cost reduction for the developer, since to detect these false paths in series avoid the construction of thousands of faulty circuits.

In this sense this paper presents a tool that, starting from an HDL code generates an intermediate code document on which is easier to detect and correct the errors introduced by false paths. This will lead, during circuit fabrication, to get more efficient and less expensive circuits.

The paper is organized as follow. Section 2 explains the false path problem. Section 3 presents the system for detecting false path. Section 4 explains some false path algorithms. Section 5 describes the partial results. Finally, Section 6 expounds the conclusion and future work.

2 The Problem

Nowadays, the industry produces digital circuit using hardware description languages. In this context, the circuits are specified and then this specification is sensitized inside a FPGA. Generally, this task produces circuits with many false paths.

A *false path* is a path that never be activated by any input vector. Observe Figure 1 and suppose that the goal is to verify if the path $< B, 1, 2, 3, 4, Y >$ is a false path. In order to carry out this task, the signals C and A must be initialized with 1 and 0 respectively. In this situation, the value of signal B is propagated to the gate input 4. To propagate the value of signal X2 to the output Y, the signal

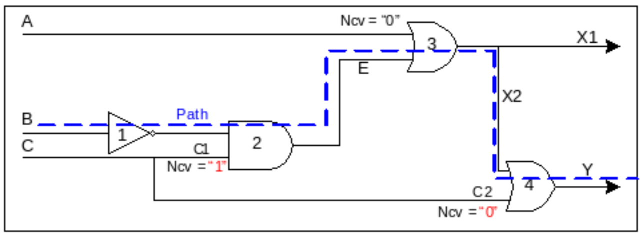

Fig. 1. A simple circuit with a false path

C must be initialized with 0. It is possible to observe that this configuration is invalid because the signal C can not be initialized with both 1 and 0 at the same time. For this reason, the path $< B, 1, 2, 3, 4, Y >$ is a false path.

3 The Solution: A System for detecting False Path

The problem mentioned in section 2 can be solved building a system with the characteristic described below.

The system is composed by:

- A syntactic/semantic analyzer [BDRG$^+$74];
- Several auxiliary routines, for the internal data structures administration and verification processes;
- A plug-in area, for inserting false path algorithms.

The input is a circuit specification generated by a synthesis tool. This specification is done using a standard intermediate language.

As output, the tool produces:

- *True or False*, when the goal is just to verify if some path is a false path,
- *A set of false paths*, when the goal is to identify all false paths.
- *True* if the internal circuit representation (Graph) is correct. *False* in other case.

The system architecture is shown in Figure 2. Next items explain each system component.

Syntactic and Semantic Analyzer: The syntactic/semantic analyzer was difficult to develop because the grammar of intermediate language (CDL. **C**ircuit **D**escription **L**anguage) is not available (The synthesis tools are commercial and they do not give access to the source code) [MHS05,DKV00]. In order to overcome this problem, the synthesizer output was manually analyzed and the following grammar was created.

1. Design ::= Subckts
2. Subckts ::= Subckt | Subckts Subckt

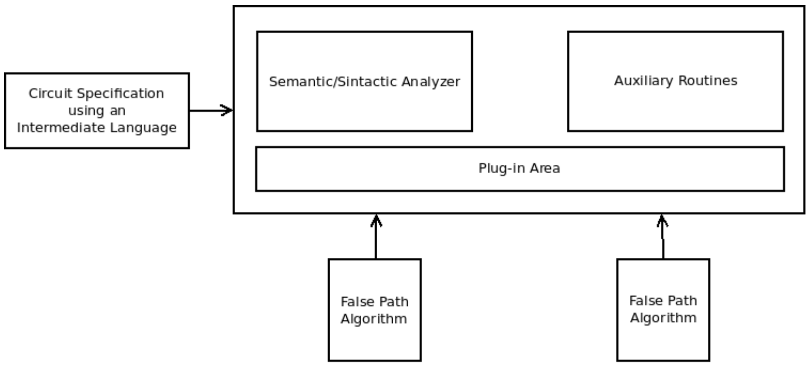
Circuit Specification using an Intermediate Language
Semantic/Sintactic Analyzer
Auxiliary Routines
Plug-in Area
False Path Algorithm
False Path Algorithm

Fig. 2. System Architecture

3. Subckt ::= **SUBCKT** Idlist **END** Id
4. Subckt ::= **SUBCKT** Idlist InstanceList **END** Id
5. Subckt ::= **SUBCKT** Idlist Params **END** Id
6. Subckt ::= **SUBCKT** Idlist InstanceList Params **END** Id
7. InstanceList ::= Instance | InstanceList Instance
8. Instance ::= **X**Id Idlist
9. Params ::= Param | Params Param
10. Param ::= **EQUATION** Id | **EQUATION STRING** | **STRING**
11. IdList ::= Id | IdList Id

The grammar defines a design as a sub-circuit set (See rules 1 and 2). Each sub-circuit can be specified as follow:

– *Describing its interface.* In this case, the component is specified in other file or it is a standard component (See rule 3).
– *Specifying its interface and its instances.* In this situation a new component is created (See rule 4).
– *Drawing its interface and equations.* This equation indicates how the gates are connected (See rule 5).
– *Giving a complete specification.* In this case, all the alternatives previously described are used (See rule 6).

The circuit instances (Rule 7) begin with a special symbol and after that an identifier list is expected. The last identifier of each instance is a gate and the others are signals. Figure 3 shows the specification of the circuit depicted in Figure 1.

Taking the context free grammar (CFG) listed above, a translation grammar (TG) was created adding attributes to the grammar symbols and semantic actions to the productions. The TG is intended to extract information from the source text in order to build the circuit internal representation that will be used in the next phase by to the auxiliary routines. For example, semantic actions detect: i) Gates; ii) Circuit Inputs; iii) Circuit outputs; iv) Gate delay; v) Wire delay, etc.

```
subckt Circuit A B C X1 Y
 Xi1 B D NEG
 Xi2 D C E AND2
 Xi3 A E X2 OR2
 Xi4 C X2 Y OR2
end Circuit
```

Fig. 3. Circuit Specification

Auxiliary Routines: The auxiliary routines have three main goals. The first one is to build a circuit representation based in graph. The second one is to implement several traversals on the graph. The thrid is to verify that the graph representation is equivalent to the CDL specification received as input.

Concerning the first goal, the circuits were modeled as a graph G=(P,E) where P is a node set and E is an arc set (A binary relation on P). Each element of P represents a circuit gate and each element of E represents the wires that connect the gates. The elements of P have attributes useful to detect the false path and other circuit properties. For example, each node has associate a: i) Delay; ii) Number of input; and iii) Number of output. Furthermore of the attributes previously described, each node has an identifier to gather it quickly.

Like nodes, the arcs also have attributes. In this case, furthermore of the arc identifier, a delay is associated with each arc. This attribute is used for computing the path delay.

The traversals (Second goal) are used to do complex connection between graph nodes (This kind of connections appear in the specification of big designs), and to implement false path algorithms.

The verification process (Third goal) is carried out through the application of several traversal on the graph for generating a CDL specification. After that, this CDL specification is compared with the input CDL file. If both files are equal then the verification process result is correct. In other case, some errors in the graph representation have been found.

Plug-in Area: This area allows to add the false path algorithms used for verifying designs. The algorithms work using: i) The circuit representation created by the Analyzer and the Auxiliary Routines; ii) The node source; and iii) The target node. One variant can be used when the goal is to detect all the circuit false path. In this case, the algorithms only needs to use the graph representation.

Currently, there are many algorithms for detecting false paths. However, they have two main problems:

- Their complexity is exponential;
- They can not be directly applied because are based on data structures not provided by the synthesis tools.

The first one is complex to solve because is not easy to reduce the algorithms computational complexity. However, for tackling the problem many heuristics[1] useful for detecting false paths can be used. The second one is successfully solved building a syntactic/semantic analyzer for creating the data structures needed by the algorithms. The analysis is carried out on a standard domain specific language as previously defined.

4 Algorithms to detect False Path

Before applying any false path checking algorithm, is necessary to discover and save all existent paths between every input and output of the circuit being scanned. After such analysis, a file containing a list of paths is created. Every path is represented by a node sequence from the beginning (Input port) to the end (Output port). This file and the graph representation are then utilized to feed whichever of project. As explained before, there are many algorithms for false path detection. Among different approaches, there is a particular one who deals with dynamic false path (This algorithm takes into account gate delays [DYG89]). Besides dynamic false path detection, there is a static false path detection which is in fact our project target. Regarding Static Analysis, we must say that a complete analysis can not be achieved applying only one algorithm. In our research, at least three different procedures must be performed in order to get a complete analysis. Those three actions are described below.

Backward Unequivocal Propagation (BUP): Given a known value V (0 or 1) on a signal S coming from a gate G (AND, OR, NAND or NOR). V is an unequivocal value only if it is generated by a unique set of values at G inputs (i.e. for a OR gate, the output equal to zero can only be obtained if all gate inputs are also zero). Therefore, the algorithm works moving backwards only on these cases. It looks for an inconsistency and if it finds one the path is declared *false*. The problem with this approach is that it is not able to set signals backwards when they are not unequivocal.

Forward Unequivocal Propagation (FUP): Given a control value V as input of a gate G, this gate output can be determined unequivocally despite other inputs. Once again, this approach move forward only for control values and is unable to set signals forward when they are not unequivocal.

Backward SEG Propagation (BSEGP): This approach was proposed by us with the intention of solving the previous issues. This algorithm performs a more complex and deep analysis on signals that are not unequivocal but ambiguous; trying with different values and including new types like strong and weak values. Drawbacks found in this solution are: The computational complexity increase hugely when the circuit under analysis has many gates.

5 Partial Results

The partial results are related with:

[1] Our research group did define several heuristics and they were applied to various designs. The heuristics have not been published.

– The implementation of some flase path algorithms.
– Improvements done on the algorithms in order to reduce the computational complexity and to increment the number of false path detected.
– Apply the false path algorithms to some test cases.

Next subsection describe each one of the items mentioned above.

5.1 False Path Algorithms

The false path algorithms implemented are described below.

False Path Checking Algorithms (FPCA) Backward unequivocal Propagation (BUP in advance) and some of Backward SEG Propagation (BSEGP) have been coded and implemented so far. Currently, BUP and BSEGP are carried out together. It means that when a signal is being treated, the backward propagation method applied depends on that signal value. So, if the program finds a non- unequivocal value, then it performs a BSEGP. On the other hand, if the program finds an unequivocal value it performs a BUP.

Deep Parameter It would enable user specify a parameter so that analysis could stop after passing a number of gates in every branch. Such parameter is not implemented.

5.2 Test Cases

In order to apply the system to real cases the process shown in Figure 4 must be used. The CDL file is a input of two process. The first one builts a graph representation and optionally verify the circuit false path. The second one generates a CDL specification from the graph. This specification is compared with the CDL input file. The main goal of this task is to check that the graph exactly represent the input CDL description.

Using the procedure mentioned above, many CDL files have been tested and have succeeded finding false paths and finishing successfully. The results obtained are shown in Table 1.

This table presents the time taken by the algorithm FPCA (See section 5). The analysis was performed using several CDL examples.

Based on the information available it is possible to affirm that: there are no direct relations between the time employed by the algorithm and the features of the file in question. Although the bigger file size the longer time is taken to finish the analysis, it does not necessarily happen for all files. The main parameter that does affect the processing time is the graph connectivity degree. The connectivity is directly related to the number of paths found for every pair source-target (Input port output port).

The results show that some file took to much time (14 hours) to finish. This happens due to the huge amount of paths found in these circuits. This characteristic is important because the designs analyzed are small. Therefore for complex designs the approaches used in the system must be improved.

File	Size	I/O Ports	Paths Found	False Path Found	FPCA Time
des90	11,7 kb	188	12965	1756	32 seg
des79	23,8 kb	639	2068	0	4 seg
des78	36 kb	557	4312	0	20 seg
des94	40 kb	799	10149	111	42 seg
des39	135,3 kb	1454	2886866	1027972	50400 seg
des38	192,6 kb	2908	61902	0	604 seg
des43	191,2 kb	2759	35590	2726	489 seg

Table 1. Results

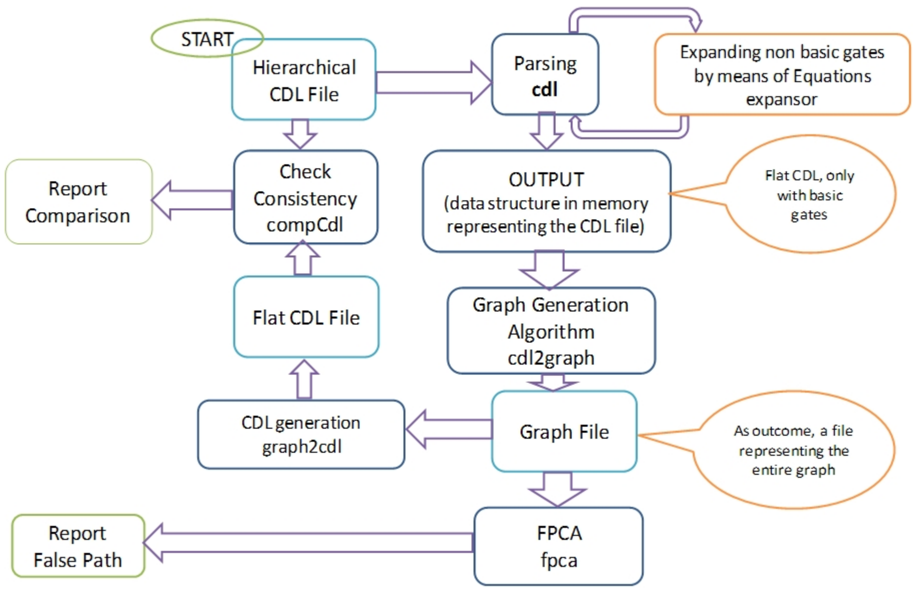
START
Hierarchical CDL File
Parsing cdl
Expanding non basic gates by means of Equations expansor
Report Comparison
Check Consistency compCdl
OUTPUT (data structure in memory representing the CDL file)
Flat CDL, only with basic gates
Flat CDL File
Graph Generation Algorithm cdl2graph
CDL generation graph2cdl
Graph File
As outcome, a file representing the entire graph
Report False Path
FPCA fpca

Fig. 4. Task Flow for detecting false path

5.3 Improvements to Apply

In our analysis, we found that the algorithm implementation order is very important regarding computational time. BUP and FUP are predictable in terms of time due to the unequivocal characteristic. When these algorithms can not continue, they just stop. After being the circuit analyzed and the BUP and FUP possibilities used up, the BSEGP should be applied. It minimizes the number of no valued signals so that BSEGP can complete in less time.

Another issue to be considered is the number of paths involved in the analysis. Finding all paths in a circuit takes much time in big circuits given that they are proportional to input and output ports number. Due to the amount of paths, it is necessary to save in hard disk and that implies the use of low speed resources. Adding other parameters like test-longest-path and if false, test next longest, indeed reduce time compared to analyzing every path in circuit.

A better and more accurate graph representation should be considered. A stronger graph library may improve the computational time as well.

It is possible to achieve better computational time doing two independent tasks at the same time. Starting with path discovering and simultaneously rebuilding back the CDL file for comparison could shrink the whole program execution time.

6 Conclusion and Future Work

In this paper a system to detect false paths was presented. This system was built with the goal of solving several problems found in the industrial circuit design. The system has the following components:

1. The syntactic/semantic Analyzer;
2. The Auxiliary Routines;
3. A Plug-in area.

The first component was difficult to build because the analysis is carried out using an intermediate language. The grammar for this language was not available because the synthesis tools are commercial software. To solve this problem, the grammar was derived through the manual inspection of several circuits specified in that intermediate language.

The second component implements: i) Algorithms for building and traversing graphs and ii) Routines for doing some useful verifications. The first one is used to connect complex circuits and to detect false paths. The second one is needed for checking that the graph exactly represent the input file.

Finally, the third component implements the interface for adding false path algorithms.

The system was used to verify several real designs (Small size) and its performance was good. However, we detected some drawbacks that deserve further research. So as future work, we plan the following tasks:

1. Improve the performance of false path checking algorithms.

2. Parallelize the analysis process.
3. Use a better graph representation.
4. Elaborate some strategies for parsing large specifications.

Another challenging and important is to extend the strategies described in this paper, for the analysis of combinational and sequentials designs [NTKW98].

References

[BDRG$^+$74] F. L. Bauer, F. L. De Remer, M. Griffiths, U. Hill, J. J. Horning, C. H. A. Koster, W. M. McKeeman, P. C. Poole, and W. M. Waite. *Compiler construction: an advanced course.* Springer-Verlag New York, Inc., New York, NY, USA, 1974. Series Editor-Goos, G. and Series Editor-Hartmanis, J.

[CCW$^+$07] Lei Cheng, Deming Chen, Martin D. F. Wong, Mike Hutton, and Jason Govig. Timing constraint-driven technology mapping for fpgas considering false paths and multi-clock domains. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 370–375, Piscataway, NJ, USA, 2007. IEEE Press.

[DKV00] Arie Deursen, Paul Klint, and J. M.W. Visser. Domain-specific languages. Technical report, Amsterdam, The Netherlands, The Netherlands, 2000.

[DYG89] D. H. Du, S. H. Yen, and S. Ghanta. On the general false path problem in timing analysis. pages 555–560, 1989.

[IOF] H. Iwata, S. Ohtake, and H. Fujiwara. An approach to rtl false path mapping using uniqueness of paths. In *Information Science Technical Report.*

[jLKWtC02] Jing jia Liou, Angela Krstic, Li-C. Wang, and Kwang ting Cheng. False-path-aware statistical timing analysis and efficient path selection for delay testing and timing validation, 2002.

[KB99] Yuji Kukimoto and Robert K. Brayton. Timing-safe false path removal for combinational modules. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 544–550, Piscataway, NJ, USA, 1999. IEEE Press.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[NP02] Mehrdad Nourani and Christos A. Papachristou. False path exclusion in delay analysis of rtl structures. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(1):30–43, 2002.

[NTKW98] Kazuhiro Nakamura, Kazuyoshi Takagi, Shinji Kimura, and Katsumasa Watanabe. Waiting false path analysis of sequential logic circuits for performance optimization. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 392–395, New York, NY, USA, 1998. ACM.

[PCDM89] S. Perremans, L. Claesen, and H. De Man. Static timing analysis of dynamically sensitizable paths. In *DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 568–573, New York, NY, USA, 1989. ACM.

[ZAB$^+$01] Jing Zeng, Magdy S. Abadir, Jayanta Bhadra, Jacob A. Abraham, and Jacob A. Abraham eda Tools. Full chip false timing path identification: Applications to the powerpc trade; microprocessors, 2001.