Accurate Computation of Longest Sensitizable Paths using Answer Set Programming

Benjamin Andres[1], Matthias Sauer[2], Martin Gebser[1], Tobias Schubert[2], Bernd Becker[2], Torsten Schaub[1]

[1] University of Potsdam
August-Bebel-Strasse 89, 14482 Potsdam, Germany
{ bandres | gebser | torsten }@cs.uni-potsdam.de

[2] Albert-Ludwigs-University Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
{ sauerm | schubert | becker }@informatik.uni-freiburg.de

Kurzfassung / Abstract

Precise knowledge of the longest sensitizable paths in a circuit is crucial for various tasks in computer-aided design, including timing analysis, performance optimization, delay testing, and speed binning. As delays in today's nanoscale technologies are increasingly affected by statistical parameter variations, there is significant interest in obtaining sets of paths that are within a length range. We present an ASP-based method for computing well-defined sets of sensitizable paths within a length range. Unlike previous approaches, the method is accurate and does not rely on a priori relaxations. Experimental results demonstrate the applicability and scalability of our method.

1 Introduction

Precise knowledge of the longest sensitizable paths in a circuit is crucial for various tasks in computer-aided design, including timing analysis, performance optimization, delay testing, and speed binning. However, the delays of individual gates in today's nanoscale technologies are increasingly affected by statistical parameter variations [1]. As a consequence, the longest sensitizable paths in a circuit depend on the random distribution of circuit features [2] and are thus subject to change in different circuit instances. For this reason, there is significant interest in obtaining sets of sensitizable paths that are within a length range, in contrast to only the longest nominal path as in classical small delay testing [3]. Among other applications, such path sets can be used in the emerging areas of *Post-silicon validation and characterization* [4] and *Adaptive Test* [5].

While structural paths can be easily extracted from a circuit architecture, many of them are not sensitizable and therefore present *false paths* [6]. The usage of such false paths leads to overly pessimistic and inaccurate results. Therefore, determination of path sensitization is required for high-quality results, although it constitutes a challenging task that requires complex path propagation and sensitization rules.

In order to reduce the algorithmic overhead, various methods for the computation of sensitizable paths make use of relaxations [7], making trade-offs between complexity and accuracy. Methods based on the sensitization of structural paths [8], [9] restrict the number of paths they consider for accelerating the computation and to limit memory usage. Due to these restrictions, however, they may miss long paths. Recent methods [10], [11] based on Boolean Satisfiability (SAT; [12]) have shown good performance results but are limited in the precision of the encoded delay values. As their scaling critically depends on delay resolution, such methods are hardly applicable when high accuracy is required.

We present an exact method for obtaining longest sensitizable paths, using Answer Set Programming (ASP; [13]) to encode the problem. ASP has become a popular approach to declarative problem solving in the field of Knowledge Representation and Reasoning (KRR). Unlike SAT, ASP provides a rich modeling language as well as a stringent semantics, which allows for succinct representations of encodings.

The remainder of the paper is structured as follows. Section 2 provides our ASP encoding of sensitizable paths. The experimental results of our encoding are presented in Section 3 and Section 4 concludes the paper.

2 ASP Encoding

The basic idea of ASP is to represent a given problem by a logic program [1] such that particular models, called answer sets, correspond to solutions, and then to use an ASP solver for finding answer sets. While ASP's input language is inspired by Logic Programming and thus allows for specifying first-order logical rules, the computation of answer sets relies on instantiation (or grounding) followed by Boolean Constraint Solving. The model-oriented approach of ASP shares similarities with PB/SAT-based problem solving; for instance, Kautz and Selman [14] pioneered SAT planning by devising propositional theories such that models (not proofs) describe solutions, and logic programs whose answer sets represent plans were provided by Lifschitz [15]. An important advantage of ASP in comparison to PB/SAT lies in its more stringent notation of modelhood, requiring any true atom to be "constructible" by applying the rules of a logic program. This constructive flavor allows for more succinct representations of inductive concepts like closures, fixpoints, and reachability than in PB/SAT. [2]

[1] In view of ASP's quest for declarativeness, the term *program* is of course a misnomer but historically too well established to be dropped.

[2] Under common assumptions in complexity theory, any vocabulary-preserving translation from ASP to SAT must be worst-case exponential [16], while there are linear-size as well as modular translations from SAT to ASP.

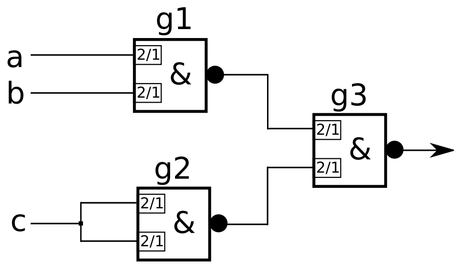

Figure 1 Example circuit for computation of longest sensitizable paths.

```
in(a). nand(g1). wire(a,g1,2,1).  out(g3).
in(b). nand(g2). wire(b,g1,2,1).  test(g3).
in(c). nand(g3). wire(c,g2,2,1).
                 wire(g1,g3,2,1).
                 wire(g2,g3,2,1).
```

Figure 2 ASP instance describing the circuit in Figure 1 by facts.

For a pragmatic introduction to ASP, we outline its application for finding the longest sensitizable path through the gate `g3` in the circuit presented by Figure 1. The numbers on the gate inputs represent the delay caused by a rising / falling edge, respectively. The first step in solving this problem consists of describing the circuit in terms of facts, yielding the ASP *instance* (i.e., a logic program consisting solely of facts) in Figure 2. Observe that the representation of the circuit by an associated ASP instance is straightforward:

`in(g).`	for each primary input `g`.
`nand(g).`	for each nand gate `g`.
`out(g).`	for each output gate `g`.
`test(g).`	for the test gate `g`.
`wire(g1,g2,r,f).`	for wireing between `g1` and `g2`.

Thus, the facts `nand(g1)`, `nand(g3)`, and `wire(g1,g3,2,1)` stands for the wiring from the nand-gate `g1` to the gate `g3` with a delay of 2 (1) in the case of a rising (falling) edge on the corresponding input pin of gate `g3`. The second and more sophisticated step is to specify logical rules such that answer sets satisfying them match problem solutions. Our rules describing the problem of finding the longest satisfiable path (for arbitrary instances) are shown in in Figure 3; such an instance-independent logic program part is called an ASP *encoding*.

Given that gate delays are only required for path length maximization, but not for the actual path calculation, the rule in Line 1 projects instances of `wire(G1,G2,R,F)` (given by facts) to wired gates `G1` and `G2`. The calculation of a path through the test gate `G` (in our example `g3`) is implemented by the Lines 3 to 5. It starts in Line 3 by choosing exactly one output gate, represented by an instance of `path(G2)`. In Line 4 the path is continued backwards including exactly one predecessor gate for every non-input gate already on the path. Given this, the so-called integrity constraint in Line 5 (the omitted left-hand side of the implication expressed by ":-" refers to an implicitly false consequence) denies answer sets, such that the path does not include the test gate `G` (here `g3`). Also note that, although path calculation is logically encoded backwards, ASP solving engines are not obliged to proceed in any such order upon searching for answer sets.

The truth assignments needed for checking whether a path at hand is sensitizable are generated by the rules in Line 7 to 10. To this this end, for each input gate `G1` of the circuit (`in(G1)` hold), choice rules allow for guessing truth values. For example, the atoms `one(a)` and `two(a)` express whether the input `a` is true in the first and the second time frame respectively. Given the values guessed for the inputs (`a`, `b` and `c`), the NAND gates (`g1`, `g2` and `g3`) are evaluated accordingly. The rules in Line 12 and 13 check whether the gate `G` has a `rising` (`falling`) flank, i.e. is sensitizable. Finaly, the integrity constraint in Line 14 stipulates that each gate on the calculated path must be either falling or rising, thus denying truth assignments whose transition does not propagate along the whole path. In order to calculate the longest sensitizable paths, the rules in Line 16 and 17 derives the delay incurred by two gates `G1` and `G2` connected along the path in respect of their flank (falling or rising). The main objective of calculating the longest paths is expressed by the `#maximize` statement in Line 18, which instructs ASP solving engines to compute answer sets such that the sum of associated gate delays is as large as possible.

One longest path through the toy example of Figure 1 found by our ASP model is `path(c)`, `path(g2)`, `path(g3)` with `falling(c)`, `rising(g2)` and `falling(g3)`.

3 Experimental Results

We evaluate our method on ISCAS85 and the combinatorial cores of ISCAS89 benchmark circuits, given as gate-level net lists. Path lengths are based on a pin-to-pin delay model with support for different rising-falling delays. The individual values have been derived from the Nangate 45nm Open Cell Library [17]. Below, we report sequential runtimes of the ASP solver *clasp* (version 2.0.4) on a Linux machine equipped with 3.07GHz Intel i7 CPUs and 16GB RAM.

The ASP instance describing the circuit and our generic encoding are grounded by *gringo*. The grounding serves as input for *clasp*, which in its first run performs optimization to identify a longest sensitizable path with maximum delay d_g. With d_g at hand, we further proceed to compute all paths with a delay equal or greater than $r = 0.95 * d_g$. This is accomplished by reinvoking *clasp* with the command-line parameters `--opt-all=`r and `--project` to enumerate all sensitizable paths within the range $[r, d_g]$. While the first parameter informs *clasp* about the threshold r for sensitizable paths to enumerate, the second is used to omit repetitions of the same path with different truth assignments. As a consequence, *clasp* enumerates distinct sensitizable paths, whose delay are at least r, without repetitions. An overlaying python program reuses the information of d_g and paths found in previous iterations to decide whether subsequent gates need to be analysed and ensures that *clasp* does not need to calculate the same paths for different gates.

Table 2 displays the runtimes of our method using a length-preserving mapping (avoiding rounding errors) of real-valued gate delays to integers. "Circuit" and "Gates" indicate a particular benchmark circuit along with its number of gates to be tested. The next three columns give statistics for the search for longest sensitizable paths, displaying the average runtime per solver call, the sum of runtimes for all gates in seconds and the number of solver calls needed to calculate d_g for all gates. The three columns below "Path set" provide statistics for the enumeration of distinct sensitizable paths with a length

```
wire(G1,G2) :- wire(G1,G2,R,F).

1 { path(G2) : out(G2) } 1.
1 { path(G1) : wire(G1,G2) } 1 :- path(G2), not in(G2).
:- test(G), not path(G).

{ one(G1) } :- in(G1).
one(G2) :- nand(G2), wire(G1,G2), not one(G1).
{ two(G1) } :- in(G1).
two(G2) :- nand(G2), wire(G1,G2), not two(G1).

falling(G) :- one(G), not two(G).
rising(G)  :- two(G), not one(G).
:- path(G), not 1[falling(G), rising(G)].

delay(G1,G2,D) :- path(G1), path(G2), wire(G1,G2,R,D), falling(G1).
delay(G1,G2,D) :- path(G1), path(G2), wire(G1,G2,D,F), rising(G1).
#maximize[ delay(G1,G2,D) = D].
```

Figure 3 ASP encoding of the longest sensitizable paths problem.

of at least r. Here, we show the average runtime for enumerating 1000 paths, the sum of runtimes for all gates, and finally the total number of different paths found. The columns below "Total" summarize both computation phases of *clasp*, optimization and enumeration. The first column present the total number *clasp* was called. Finally, the last two columns provide the total solving time of *clasp* for both computation passes and the total runtime needed for the benchmark.

As can be seen in Table 2, the scaling of our method is primarily dominated by the number of gates in circuits. Over all circuits, the average runtime for processing one test gate is rather low and often within fractions of a second. In addition, our method allows for enumerating the complete set of sensitizable paths within a given range in a single solver call, thus avoiding any expenses due to rerunning our solver. This allows us to enumerate thousands of sensitizable paths and test pattern pairs sensitizing them very efficiently. In fact, the overhead of path set computation compared to optimization in the first phase is relatively small, even for complex circuits. E.g., for the c3540 circuit, 2.26 seconds are on average required for optimization, and 10.42 seconds on average per 1000 enumerated paths. The rather large discrepancy between solving and total runtime for large, computational easy circuits, e.g. cs13207, is explained by the fact that *clasp* currently needs to read the grounded file from the disc for every call. To overcome this bottleneck we hope to utilize *oclingo*, an incremental ASP system implemented on top of *gringo* and *clasp*, in future work as soon as *oclingo* supports #`maximize` statements. This would allow us to analyze all gates of a circuit within a single solver call, thus drastically reducing the disc access. In addition, the *oclingo* could reuse information gained from previously processed gates for solving successive gates, efficiently.

In order to demonstrate the scaling of our approach wrt delay accuracy, we also used different mappings of real-valued delays to integers, and corresponding runtime results for the ISCAS85 benchmark set as shown in Table 1. In addition to the exact mode used in the previous experiment, we employed a rounding method to five delay values, shown in the columns labeled with "5". Likewise, we applied rounding to 1000 delay values. As before, we report average runtimes per call in seconds for the two phases of optimizing sensitizable path length and of performing enumeration. Considering the results, we observe that runtimes of *clasp* are almost uninfluenced by the precision of gate delays. This is explained

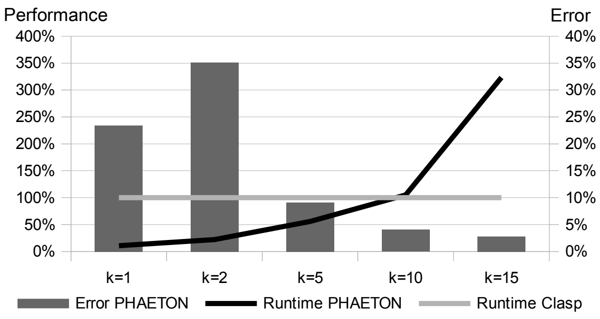

Figure 4 Comparison with PHAETON [10] using ISCAS85 circuits

by the fact that weights used in #`minimize` or #`maximize` statements do influence the space of answer sets wrt to which optimization and enumeration are applied. In the ISCAS89 benchmark set the solving time per call was almost universally less than $0.01s$.

We compared our method with an SAT-based approach called "PHAETON" proposed in [10]. The results are shown in Figure 4. The Figure shows the runtime needed by PHAETON to compute 1000 paths for ISCAS85 benchmark circuits with different levels of accuracy indicated by the number of delay steps k. In order to compare the results of the proposed method with PHAETON, the runtime is given as percent on the primary x-axis, with 100% being our method. The secondary x-axis gives the discretization error of PHAETON. As can be seen, for low accuracy levels which result in an average discretization error of around 5%, PHAETON scales better than our optimal approach. However, for increased accuracy levels, the proposed method outperforms PHAETON and is therefore better suited for precise computation of longest sensitizable paths.

4 Conclusions

We presented a method for the accurate computation of sensitizable paths based on a flexible and compact encoding in ASP. Unlike previous methods, our approach does not rely on a priori relaxations and is therefore exact. We demonstrated the applicability and scalability of our method by extensive experiments on ISCAS85 and ISCAS89 benchmark circuits.

Future work includes further efforts to optimize the ASP encoding by incorporating additional rules, with the goal of

Circuit	Gates	Longest path (d_g)			Path set (95%)			Total		
		Time in s per call	Time in s	Calls	Time in s per 1000 paths	Time in s	Paths	Solver calls	Solving Time in s	Total time in s
*	< 220	< 0.01	< 0.01	< 70	< 0.01	< 0.01	< 310	< 210	< 0.01	< 3.20
c0432	160	0.05	2.46	53	0.24	4.67	19356	112	7.13	11.67
c0499	202	0.01	0.64	64	0.49	0.94	1928	160	1.58	5.54
c0880	383	< 0.01	0.33	82	0.21	0.77	3617	212	1.10	7.78
c1355	546	0.29	18.87	64	1.36	32.54	23936	160	51.41	63.60
c1908	880	0.25	34.37	137	2.00	64.33	32174	378	98.70	131.22
c2670	1269	0.01	5.30	440	1.41	8.05	5700	1023	13.35	101.79
c3540	1669	2.26	544.32	241	10.42	1125.60	107994	697	1669.92	1799.69
c5315	2307	0.05	25.43	485	2.02	39.65	19603	1206	65.08	266.83
c7552	3513	0.04	24.59	576	1.97	40.77	20745	1622	65.36	444.07
cs00526	194	< 0.01	< 0.01	74	< 0.01	< 0.01	247	190	< 0.01	2.17
cs00641	379	< 0.01	0.01	68	0.06	0.02	326	236	0.03	5.74
cs00713	393	< 0.01	0.01	84	0.06	0.02	309	276	0.03	5.20
cs00820	289	< 0.01	0.02	71	< 0.01	< 0.01	361	273	0.02	5.36
cs00832	287	< 0.01	0.02	73	< 0.01	< 0.01	372	269	0.02	5.02
cs00838	446	< 0.01	0.05	140	0.18	0.15	853	444	0.20	12.31
cs00953	418	< 0.01	0.01	111	0.03	0.01	342	354	0.02	7.48
cs01196	530	< 0.01	0.39	145	0.62	0.34	550	417	0.73	14.90
cs01238	509	< 0.01	0.54	144	0.72	0.42	586	400	0.96	13.39
cs01423	657	< 0.01	1.51	184	0.87	1.94	2236	529	3.45	25.05
cs01488	653	< 0.01	0.02	155	0.02	0.01	517	676	0.03	22.10
cs01494	647	< 0.01	0.02	157	0.02	0.01	521	654	0.03	21.69
cs05378	2779	< 0.01	0.65	506	0.32	1.71	5334	1759	2.36	320.37
cs09234	5597	< 0.01	6.82	795	0.69	13.54	19703	3483	20.36	1091.54
cs13207	8027	0.02	27.15	1332	2.97	53.41	18011	5864	80.56	2833.38
cs15850	9786	0.66	973.07	1480	9.56	3172.45	331964	6322	4145.52	9825.64
cs35932	16353	< 0.01	0.06	5321	0.41	5.9	14321	13463	5.96	16437.94
cs38584	19407	< 0.01	33.23	7266	2.35	65.18	27722	20227	98.41	42700.61

* This includes the circuits c0017, c0095, cs00027, cs00208, cs00298, cs00344, cs00349, cs00382, cs00386, cs00400, cs00420, cs00444, cs00510.

Table 2 Application using exact delay values

Circuit	Time (d_g) per call			Time (95%) per call		
	5	1000	exact	5	1000	exact
c0017	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
c0095	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
c0432	0.04	0.05	0.05	0.07	0.08	0.08
c0499	0.01	0.01	0.01	0.01	0.01	0.01
c0880	< 0.01	0.01	< 0.01	0.01	0.01	0.01
c1355	0.13	0.22	0.29	0.15	0.21	0.34
c1908	0.20	0.31	0.25	0.20	0.47	0.27
c2670	0.01	0.01	0.01	0.01	0.02	0.01
c3540	2.24	2.55	2.26	2.50	2.63	2.47
c5315	0.04	0.06	0.05	0.04	0.08	0.05
c7552	0.04	0.05	0.04	0.04	0.07	0.04

Table 1 Delay accuracy comparison

reducing the search space and helping *clasp* to discard unsatisfactory sensitizable paths faster. Another way to improve runtime is to specialize *clasp*'s search strategy to the problem of calculating (longest) sensitizable paths.

5 Acknowledgements

Parts of this work are supported by the German Research Foundation under grant GRK 1103.

6 Literatur

[1] "International Technology Roadmap For Semiconductors," Available at `http://www.itrs.net`.

[2] K. Killpack, C. Kashyap, and E. Chiprout, "Silicon speedpath measurement and feedback into eda flows," in *DAC '07*, june 2007, pp. 390 –395.

[3] M. Kumar and S. Tragoudas, "High-quality transition fault ATPG for small delay defects," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 5, pp. 983 –989, 2007.

[4] P. Das and S. Gupta, "On generating vectors for accurate post-silicon delay characterization," in *ATS'11*, 2011, pp. 251 –260.

[5] P. Maxwell, "Adaptive test directions," in *ETS'10*, 2010, pp. 12 –16.

[6] O. Coudert, "An efficient algorithm to verify generalized false paths," in *DAC'10*, 2010, pp. 188 –193.

[7] J. Jiang, M. Sauer, A. Czutro, B. Becker, and I. Polian, "On the optimality of k longest path generation algorithm under memory constraints," in *DATE*, 2012.

[8] W. Qiu and D. Walker, "An efficient algorithm for finding the k longest testable paths through each gate in a combinational circuit," in *Test Conference, 2003. Proceedings. ITC 2003. International*, vol. 1, 30-oct. 2, 2003, pp. 592 – 601.

[9] J. Chung, J. Xiong, V. Zolotov, and J. Abraham, "Testability driven statistical path selection," in *DAC'11*, 2011, pp. 417 – 422.

[10] M. Sauer, J. Jiang, A. Czutro, I. Polian, and B. Becker, "Efficient SAT-based search for longest sensitisable paths," in *ATS'11*, 2011, pp. 108 –113.

[11] M. Sauer, A. Czutro, T. Schubert, S. Hillebrecht, I. Polian, and B. Becker, "SAT-based analysis of sensitisable paths," in *2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2011, pp. 93 –98.

[12] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185.

[13] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[14] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, B. Neumann, Ed., 1992, pp. 359–363.

[15] V. Lifschitz, "Answer set programming and plan generation," *Artificial Intelligence*, vol. 138, no. 1-2, pp. 39–54, 2002.

[16] V. Lifschitz and A. Razborov, "Why are there so many loop formulas?" *ACM Transactions on Computational Logic*, vol. 7, no. 2, pp. 261–268, 2006.

[17] "Nangate 45nm open cell library," Available at http://www.nangate.com.