Functional vector generation for combinational circuits based on data path coverage metric and mixed integer linear programming

4 authors, including:

Javier Sosa
Universidad de Las Palmas de Gran Canaria
63 PUBLICATIONS   **103** CITATIONS

SEE PROFILE

Juan A Montiel-Nelson
Universidad de Las Palmas de Gran Canaria
178 PUBLICATIONS   **463** CITATIONS

SEE PROFILE

J.C. Garcia
Universidad de Las Palmas de Gran Canaria
55 PUBLICATIONS   **193** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project   I am working about level shifters using 65nm CMOS technology. View project

Functional Vector Generation for Combinational Circuits
Based on Data Path Coverage Metric and Mixed Integer Linear Programming

J. Sosa, Juan A. Montiel–Nelson, H. Navarro and José C. García

Institute for Applied Microelectronics, Department of Electronic Engineering and Automation,
University of Las Palmas de Gran Canaria, E-35017 Las Palmas de Gran Canaria, Spain
{jsosa,montiel,hnavarro,jcgarcia}@iuma.ulpgc.es

Abstract

In this paper, a functional vector generation method to maximize the data path coverage of a combinational circuit is introduced. We present a new gate model based on sensitization requirements for transition propagation, and introduce a new methodology to obtain functional vectors of maximum coverage based on Mixed Integer Linear Programming (MILP). Performance comparison and results based on a large set of MCNC'91 [1] benchmark circuits are given. Experimental results show significant speedups over a greedy SAT method.

1 Introduction

The goal of a functional verification or test is to assure compliance with the specification [2]. An important key is the coverage metric. It measures the degree of confidence in the verification process. In this sense, several coverage metrics have been proposed in previous research works [3]. From all of them, the coverage metrics based on the circuit structure are in general more intuitive and easy to measure. For example, the toggle coverage consists in measuring the total number of circuit nodes or gates that changes its output values with input stimuli [4]. Most of the logic simulators support the toggle coverage metrics. Another coverage metric widely used is the *data path coverage* [5]. It indicates the percent of paths that have been exercised during the verification/test stage. Against, recent logic simulators do not yet support the *data path coverage* as the toggle coverage.

Since the *data path coverage* was formulated in [5], it is used to check a small set of all possible paths. Sometimes, it is not possible to check all paths. That is, there exist circuit structures that may not be logically possible to exercise. Despite of this disadvantage, the *data path coverage* metric has demonstrated efficient to look for bugs that, otherwise they may continue as uncovered bugs [3]. Also, the *data path coverage* metric can be applied successfully to small circuits using exhaustive simulations. However, as the number of paths to verify grows exponentially with the number of circuit gates, the verification based on *data path coverage* is more difficult.

There are several advances in the satisfiability field [6, 7]. In particular, these advances are used to determine the input vector to sensitize a given path, as required in *data path coverage*. In addition, these recent methodologies have proven to be more efficient than other SAT methodologies based on exhaustive simulation or backtracking search strategies [8]. In addition the Mixed Integer Linear Programming (MILP) technique permits to define and solve the SAT problem in an efficient way, as is demonstrated in [7].

In this sense, this paper introduces a novel methodology to generate functional vectors that maximize the *data path coverage* of combinational circuits using MILP. Comparisons between our methodology and a greedy SAT checking method are done over a large set of MCNC'91 [1] benchmark circuits. The greedy approach consists in the explicit enumeration of path sets. For each set, a SAT problem is solved, and therefore, an input vector is obtained — if the solution is feasible.

The proposed methodology generates input vectors of maximum coverage. The resulting vectors sensitize more than one path, simultaneously. This approach is oriented to reduce the total number of vectors needed to check a design. Transitions of input vectors are propagated from primary inputs to primary outputs through *propagation paths*. In a *propagation path*, it is not allowed that more than one transition arrives to a gate. To characterize properly the *propagation path* in a circuit, a gate model is provided, such that, each logic gate is codified using two bits, namely, a data bit and a propagation bit.

The organization of the paper is as follows. Section 2 introduces several graph definitions and concepts. Also, the *data path coverage* metric and satisfiability are explained in detail. In Section 3, the propagation of a transition and its satisfiability conditions are discussed. To propagate transitions on circuit paths, a gate model is introduced. A func-

tional vector generation method based on explicit enumeration is presented in Section 4. Our solution based on MILP is exposed in Section 5. To sum up, experimental results and main conclusions are given in Sections 6 and 7, respectively. The obtained results show a great improvement in computation effort of the proposed methodology versus a SAT greedy solution.

2 Data path coverage problem

In order to understand the next sections, some graph definitions and concepts are following. After this introduction, the path coverage problem is discussed in detail.

2.1 Graph–theoretic definitions

Given a graph $\overrightarrow{G}$, composed by vertices v and edges e between vertices, it represents a combinational logic circuit [9] where: the input and output vertices of the graph represent the combinational logic circuit inputs and outputs, respectively. The graph internal vertices model the logic functions of the combinational logic circuit. The edges represent the data connections between logic functions, inputs and outputs.

Formally, given a Boolean network $\overrightarrow{G}$ corresponding to a combinational logic circuit obtained by technology independent synthesis procedures — $\overrightarrow{G}$ is a structural description using 2–input AND, 2–input OR and INV gates — a set of design constrains and a logic gate library L, a mapping M is a transformation of $\overrightarrow{G}$ into a net–list of logic gates.

A walk of $\overrightarrow{G}$ is a sequence of vertices v_i and edges e_j, that it is expressed as $v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$ such that e_i connects nodes v_i and v_{i+1}. A tour is a walk where all edges are different. A path is a tour with distinct vertices. A path connecting nodes v_0 and v_k is defined as $path_{v_0,v_k} = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$; where the path tail $t(path_{v_0,v_k})$ is v_0, i.e. the tail of its first edge e_0; the path head $h(path_{v_0,v_k})$ is v_k, i.e. the head of its last edge e_{k-1}; and the set of vertices is $v(path_{v_0,v_k}) = \{v_0, v_1, \dots, v_k\}$. Moreover, we are interested in those paths where the path tail v_0 and head v_k are circuit primary inputs and outputs, respectively.

2.2 Data path coverage

A verification quality measurement is the *data path coverage* [5]. The data path coverage metric is used to measure how thoroughly the paths are exercised over some specified set of values. The data path coverage value is defined as:

$$data\ path\ coverage = \frac{number\ of\ exercised\ paths}{cover\ set},$$

where *cover set* is the total number of paths in the circuit. For example, given a combinational logic circuit, as shown

in Figure 1(a), its graph $\overrightarrow{G}$ is illustrated in Figure 1(b). Table 1 enumerates all those paths that should be exercised to obtain the circuit full verification coverage, i.e. the *cover set*. The first column is the label of each path. The second column shows the ordered set of nodes — primary input node, internal nodes and primary output nodes — for each path. Last column is the output logic value of some nodes to ensure the exercising of each path, i.e. the satisfiability conditions of a path. For example, in order to check the
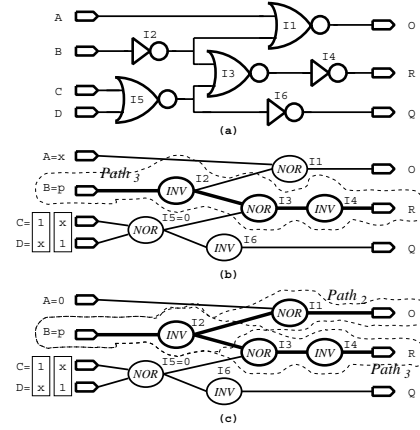

Figure 1. A data path coverage example: (a) combinational logic circuit, (b) single path exercising, (c) multiple path exercising

correctness of the path labelled as $Path_3$, the node I_5 should be fixed to logic zero, otherwise, $Path_3$ can not be checked. In this condition, a transition from low to high logic value or high to low logic value is propagated through $Path_3$, and its functionality is checked. In other words, in those conditions, a transition is propagated from a primary input to a primary output. Figure 1(b) illustrates a set of primary input logic values (i.e. the input patterns) used to exercise $Path_3$. The input transition stimulus is represented by the 'p' value. The total circuit coverage achieved in this example with $Path_3$ exercising is 14.286%. An important note

Table 1. Paths and satisfiability conditions of the data path coverage example

name	path	conditions
$Path_1$	$A–I_1–O$	$I_2 = 0$
$Path_2$	$B–I_2–I_1–O$	$A = 0$
$Path_3$	$B–I_2–I_3–I_4–R$	$I_5 = 0$
$Path_4$	$C–I_5–I_3–I_4–R$	$I_2 = 0$
$Path_5$	$C–I_5–I_6–Q$	–
$Path_6$	$D–I_5–I_3–I_4–R$	$I_2 = 0$
$Path_7$	$D–I_5–I_6–Q$	–

that must be remarked is that one input pattern may exercise more than one path at the same time. In this sense, paths $Path_2$ and $Path_3$, are simultaneously checked with the input

pattern `0p1x` or with the pattern `0px1`, as shown in Figure 1(c). Therefore, the data path coverage of this pattern is 28.57%.

3 Path propagation satisfiability

Given a graph $\overrightarrow{G}$ representing a Boolean network, $path_{v_0,v_k}$ is a *propagation path* between a primary input vertex v_0 and a primary output vertex v_k, if incoming transitions at vertex v_0 are propagated through the set of vertices $v(path_{v_0,v_k}) = \{v_0, v_1, ..., v_k\}$, and transitions are available at the primary output vertex v_k. In order to establish the *propagation path*, several primary inputs should be set to specific logic values, that is, an input condition must be satisfied. Given a path to propagate a transition, its input condition depends on the propagation conditions of each gate of the path.

To characterize properly the *propagation path* in a Boolean logic circuit, we need to introduce a new gate model. It consists in to codify each gate with two bits, a data bit and a propagation bit (i.e. a_d and a_p for the signal a, respectively). If propagation bit is active, there is a propagation. Otherwise, the data bit contains a valid data value.

Intuitively, a 2–input AND gate — with inputs a, b and output y — propagates a transition when a transition arrives to one of its inputs and the other input is set to logic one. If the transition arrives to one input and the other input is set to logic zero, the transition is killed and the output is set to logic zero. Finally, if more than one transition arrives simultaneously — this happens when more than one transition converge in a gate — its output could not have a controllable value. Therefore, it must be avoided. When there is not input transition, the gate behaviour is the normal AND logic function. Formally, the gate model is:

$$y_p = ((a_p \cap b_d \neq \phi) \vee (a_d \cap b_p \neq \phi)) \wedge (a_p \cap b_p = \phi) \quad (1)$$

$$y_d = ((a_d \cap b_d \neq \phi) \wedge (a_p \cup b_p = \phi)) \quad (2)$$

In a similar manner, a 2–input OR gate propagates a transition, if a transition arrives to one of its inputs and the other input is set to logic zero. But, if the other input is set to logic one, the gate sets its output to logic one and kills the propagation. When there is not input transition, the gate behaviour is the common OR logic function. Therefore, the new OR gate model is described as:

$$y_p = ((a_p \cap \bar{b_d} \neq \phi) \vee (\bar{a_d} \cap b_p \neq \phi)) \wedge (a_p \cap b_p = \phi) \quad (3)$$

$$y_d = ((a_d \cup b_d \neq \phi) \wedge (a_p \cup b_p = \phi)) \quad (4)$$

And finally, the INV always propagates a transition when it arrives to its single input. Otherwise, the logic inversion is performed. The INV description is:

$$y_p = a_p \quad (5)$$

$$y_p = \bar{a_d} \quad (6)$$

Given a path or a set of paths, the *propagation satisfiability* means to find an input vector to guarantee the propagation conditions for each gate of the given paths.

4 Functional vector generation based on explicit enumeration

Figure 2 illustrates the algorithm to sensitize a set of paths. Given a set of paths, the sensitization requirements for each path of the set are annotated in the circuit. At this stage two different errors can arise, namely, *transition convergence* and *sensitization contradiction*. A *transition convergence* error is established when several transitions arrives to any gate. The propagation conditions of the paths can derive different values at the same intermediate signals, that is, a *sensitization contradiction*. At last stage, the algorithm solves the SAT propagation conditions problem. The

```
PCFVector(path_set)
 foreach path in path_set
   write sensitization requirement on
     intermediate signal values of current path;
 end foreach
 solve the SAT problem;
end PCFVector
```

Figure 2. Algorithm to obtain a functional vector that sensitizes a set of paths

algorithm of Figure 2 only provides an input vector that sensitizes a set of paths. However, our main concern is finding an input vector that maximizes the *data path coverage* of the circuit. A greedy strategy is based on searching by explicit enumeration an input vector that sensitizes the maximum number of paths. In general, such approach is unpractical when the circuit complexity — in terms of number of inputs, outputs and gates — is high. Therefore, we need cut down the number of paths on each set. This bound is exactly the maximum path coverage. As will be illustrated in Table 2, higher complexity — measured in gate count — requires longer computation time.

5 MILP functional vector generation

The functional vector generation that maximizes the *data path coverage* can be modelled using MILP as follows. First, the circuit must be translated to MILP equations. Second, it is necessary to provide an objective function.

5.1 Circuit description in MILP

The work presented in [7] describes a method to model binary logic gates in MILP. The models work correctly when the input variables are kept in the integer range $[0,1]$. Therefore, the output variables will be kept in the integer range $[0,1]$. It implies that circuit primary input variables must be defined as binary variables in the MILP problem.

The behavioural of a 2–input AND gate in MILP following our model (see Equation 1 and 2) is expressed as follows:

$$y_d < a_d \qquad (7)$$
$$y_d < b_d \qquad (8)$$
$$y_d > a_d + b_d - 1 \qquad (9)$$

$$y_p' < b_p \quad (10) \qquad y_p'' < a_p \quad (13)$$
$$y_p' < a_d \quad (11) \qquad y_p'' < b_d \quad (14)$$
$$y_p' > a_d + b_p - 1 \quad (12) \qquad y_p'' > b_d + a_p - 1 \quad (15)$$
$$y_p = y_p' + y_p'' \quad (16) \qquad a_p + b_p < 1 \quad (17)$$

Equations 7, 8 and 9 model data bit behavioural (see Equation 2). Signal y_p' models a transition that arrives to the input terminal b of the 2–input AND gate. For example, the AND gate does not propagate any transition from terminal b, while a transition has not arrived to the input terminal b or the input terminal a is fixed to logic zero. That is implemented with Equations 10 and 11, respectively. Equation 12 express the condition to propagate a transition from input terminal b. Signal y_p'' models the propagation of a transition from input terminal a (see Equations 13, 14 and 15). Finally, the gate propagation bit is composed by signals y_p' and y_p'' as is illustrated in Equation 16. Equation 17 is used to avoid the *transition convergence* errors.

The behavioural of a 2–input OR gate in MILP following our model (see Equations 3 and 4) is expressed as follows:

$$y_d > a_d \qquad (18)$$
$$y_d > b_d \qquad (19)$$
$$y_d < a_d + b_d \qquad (20)$$

$$y_p' < b_p \quad (21) \qquad y_p'' < a_p \quad (24)$$
$$y_p' < a_d \quad (22) \qquad y_p'' < b_d \quad (25)$$
$$y_p' > a_d + b_p - 1 \quad (23) \qquad y_p'' > b_d + a_p - 1 \quad (26)$$
$$y_p = y_p' + y_p'' \quad (27) \qquad a_p + b_p < 1 \quad (28)$$

Equations 18, 19 and 20 implement the OR data bit. Equations 21, 22 and 23 model the propagation of a transition from the input b. Similarly, Equations 24, 25 and 26 conform the propagation of a transition from the input a. Finally, the gate propagation bit is composed by signals y_p' and y_p'' as is illustrated in Equation 27. Equation 28 is used to avoid the *transition convergence* error.

The behavioural of an INV gate in MILP following our model (see Equation 5 and 6) is expressed as follows:

$$y_d = 1 - a_d \quad (29)$$
$$y_d = a_p \quad (30)$$

Equation 29 implements the data inversion, and Equation 30 models the propagation bit.

5.2 MILP optimization

Given a Boolean network $\vec{G}$ corresponding to a combinational logic circuit obtained by technology independent synthesis procedures — $\vec{G}$ is a structural description using 2–input AND, 2–input OR and INV gates — it is translated to MILP equations using our gate models (see Equations 1–6). The objective function to be maximized represents the number of sensitized paths.

As result, for each gate of the sensitized paths, their propagation bits are activated. Because our model avoids *transition convergence* at each gate, if a propagation bit is activated at the primary output, then there exists a single path between a primary input and this primary output. Therefore, *maximize the number of sensitized paths is equivalent to maximize the number of propagation bits at the primary outputs of the circuit*.

6 Experimental results

We processed a large set of two–level and multi–level examples of the MCNC'91 [1] benchmark suite to determine a functional vector that maximizes the *data path coverage* of a combinational circuit.

The results were computed in a Sun–Fire 280R server, powered by two UltraSPARC III at 900 MHz, with 4 GByte of RAM memory. We developed a logic simulator in C programming language to solve the satisfiability problem with the proposed gate model and algorithms (see Section 3 and 4). In addition, we use GLPK [10] software as MILP solver.

Each circuit of the MCNC benchmark suite was preprocessed by MISII [11] logic synthesis system and mapped to a library of logic gates — 2–inputs NAND, 2–input NOR and INV — with minimum area and power consumption. A functional vector exercising the maximum number of observable paths was determined by both a greedy algorithm and the proposed methodology. The greedy approach consists in the explicit enumeration of sets of paths to find a functional vector that exercise them. In addition, we limit the number of paths on each set of the greedy strategy to the maximum coverage number obtained by the proposed approach.

Table 2 shows comparisons in terms of CPU time to compute the first functional vector that maximizes the *data path coverage* of the combinational circuit. The first column gives the name of the circuit according to the MCNC benchmark suite.

The complexity of the circuit is measured in terms of number of gates (column two), number of inputs (column three), number of outputs (column four) and number of paths (column five). Column six gives the number of variables used in the MILP problem, and column seven presents the number of Boolean variables. Column eight presents the number of paths for the best obtained coverage. Column nine presents the total time required to solve the MILP problem.

Column ten gives the CPU time needed to obtain an input vector that sensitizes a set of paths. Next column provides

Table 2. Experimental results and comparisons using MILP and data path coverage

circuit name	# gates	# inputs	# outputs	# paths	# used vars	# bin vars	coverage	MILP CPU time	search CPU time	# comb.	estimated CPU time	CPU time ratio
5xp1	122	7	10	172	234	14	9	1.44	6.00E-05	1.37E+19	8.19E+14	5.69E+14
9sym	239	9	1	297	434	18	1	3.31	1.40E-04	9.70E+01	1.36E-02	4.10E-03
9symml	235	9	1	286	416	18	1	3.03	1.20E-04	1.34E+02	1.61E-02	5.31E-03
C17	776	5	2	9	28	10	2	0.02	1.30E-04	2.00E+01	2.60E-03	1.30E-01
C2670	1053	233	140	47295	2328	466	11	790.20	6.70E-04	7.13E+47	4.78E+44	6.04E+41
C7552	2697	207	108	565784	4270	414	9	1105.75	1.06E-03	2.32E+22	2.46E+19	2.23E+16
C880	497	60	26	22151	868	120	25	504.91	2.70E-04	1.15E+51	3.12E+47	6.17E+44
Z5xp1	635	7	10	2866	846	14	9	121.79	1.90E-04	1.81E+30	3.44E+26	2.83E+24
Z9sym	232	9	1	292	414	18	1	3.06	9.00E-05	5.40E+01	4.86E-03	1.59E-03
alu2	559	10	6	39874	808	20	6	6.10	2.00E-04	2.23E+24	4.46E+20	7.30E+19
alu4	1110	14	8	402207	1514	28	8	245.63	3.10E-04	4.92E+42	1.53E+39	6.21E+36
apex1	1312	45	45	8377	1822	90	16	844.62	4.70E-04	9.06E+57	4.26E+54	5.04E+51
apex2	482	39	3	914	818	78	3	13.76	2.00E-04	1.01E+08	2.01E+04	1.46E+03
apex3	1762	54	50	7893	2450	108	10	1494.39	5.00E-04	1.96E+38	9.79E+34	6.55E+31
apex4	2718	9	19	15171	3416	18	15	2004.89	7.40E-04	1.22E+51	9.06E+47	4.52E+44
apex5	1067	117	88	3362	1900	234	84	797.05	5.10E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
apex6	974	135	99	1783	1824	270	96	220.64	5.70E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
apex7	299	49	37	198	604	98	35	11.44	2.10E-04	9.43E+88	1.98E+85	1.73E+84
b12	105	15	9	126	220	30	7	1.69	1.20E-04	1.18E+10	1.42E+06	8.40E+05
b1	11	3	4	12	30	6	3	0.02	1.30E-04	7.20E+02	9.36E-02	4.68E+00
b9	142	41	21	214	324	82	18	1.62	1.10E-04	7.94E+33	8.74E+29	5.39E+29
bw	211	5	28	388	366	10	15	3.51	1.40E-04	1.52E+31	2.13E+27	6.06E+26
c8	149	28	18	189	300	56	18	0.43	1.10E-04	2.33E+33	2.56E+29	5.95E+29
cc	82	21	20	109	188	42	17	0.31	1.70E-04	2.64E+31	4.49E+27	1.45E+28
cht	192	47	36	235	408	94	36	0.93	2.20E-04	1.36E+57	2.99E+53	3.21E+53
clip	130	9	5	234	228	18	5	0.75	1.20E-04	5.80E+09	6.96E+05	9.28E+05
cm138a	32	6	8	48	68	12	2	0.07	8.00E-05	5.00E+00	4.00E-04	5.71E-03
cm150a	47	21	1	55	128	42	1	0.05	1.00E-04	5.40E+01	5.40E-03	1.08E-01
cm151a	27	12	2	46	72	24	2	0.03	1.30E-04	1.98E+03	2.57E-01	8.58E+00
cm152a	24	11	1	22	66	22	1	0.03	8.00E-05	6.40E+04	6.40E-04	2.13E+02
cm162a	56	14	5	83	110	28	5	0.06	7.00E-05	4.53E+08	3.17E+04	5.29E+05
cm163a	47	16	5	57	112	32	5	0.05	1.00E-04	3.68E+07	3.68E+03	7.37E+04
cm42a	34	4	10	44	72	8	2	0.09	1.00E-04	4.00E+00	4.00E-04	4.44E-03
cm82a	29	5	3	45	56	10	3	0.03	1.20E-04	3.15E+04	3.78E+00	1.26E+02
cm85a	56	11	3	91	108	22	3	0.08	1.00E-04	1.22E+03	1.22E-01	1.52E+00
cmb	56	16	4	112	122	32	2	0.23	1.20E-04	2.50E+03	3.00E-01	1.30E+00
comp	212	32	3	1536	356	64	2	7.11	1.40E-04	1.01E+06	1.42E+02	1.99E+01
con1	23	7	2	19	52	14	2	0.04	9.00E-05	1.26E+02	1.13E-02	2.84E-01
cordic	23	23	2	140	164	46	2	0.08	1.00E-04	7.02E+02	7.02E-02	8.78E-01
count	77	35	16	368	354	70	16	0.56	1.60E-04	1.15E+31	1.85E+27	3.30E+27
cu	59	14	11	96	136	28	3	0.38	9.00E-05	2.90E+10	2.61E+06	6.86E+06
dalu	1462	75	16	1157487	2112	150	16	311.41	4.60E-04	1.34E+90	6.18E+86	1.98E+84
decod	32	5	16	80	98	10	2	0.22	1.10E-04	1.76E+03	1.94E-01	8.80E-01
duke2	4704	22	29	2051	776	44	13	394.80	2.20E-04	6.85E+38	1.51E+35	3.82E+32
ex4	459	128	28	577	1126	168	14	12.04	3.20E-04	1.05E+32	3.35E+28	2.78E+27
example2	373	85	66	1327	836	170	51	370.66	2.80E-04	8.89E+92	2.49E+89	6.72E+86
f51m	137	8	8	188	262	16	8	1.19	1.20E-04	4.19E+15	5.03E+11	4.23E+11
frg1	142	28	3	148	286	56	3	0.25	2.10E-04	1.32E+02	2.77E-02	1.11E-01
frg2	1144	143	139	6075	2050	286	13	42.76	6.00E-04	4.83E+39	2.90E+36	6.77E+34
i1	55	25	16	72	156	50	13	0.13	1.80E-04	8.07E+21	1.45E+18	1.12E+19
i2	217	201	1	228	822	402	1	1.55	4.50E-04	2.27E+02	1.02E-01	6.59E-02
i3	152	132	6	132	528	264	6	0.67	3.60E-04	4.56E+06	1.64E+03	2.45E+03
i5	198	133	66	672	662	266	66	0.80	4.00E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
i6	559	138	67	778	1182	276	67	9.50	3.60E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
i7	755	199	67	1003	1558	398	67	11.79	5.90E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
i8	1331	133	81	9811	2262	266	81	51.15	5.40E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
i9	792	88	63	19919	1382	176	63	23.32	3.80E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
inc	208	7	9	520	310	14	6	8.08	1.00E-04	1.00E+15	1.00E+11	1.24E+10
k2	1313	45	45	9127	1796	90	16	330.13	3.90E-04	3.44E+58	1.34E+55	4.07E+52
lal	122	26	19	197	240	52	17	0.25	1.10E-04	2.32E+33	2.55E+29	1.02E+30
majority	13	5	1	10	30	10	1	0.01	9.00E-05	3.00E+00	2.70E-04	2.70E-02
misex1	70	8	7	119	128	16	5	0.45	1.00E-04	3.27E+09	3.27E+05	7.28E+05
misex2	122	25	18	174	254	50	5	21.94	1.30E-04	1.70E+09	2.21E+05	1.01E+04
misex3	864	14	14	2792	1380	28	11	451.35	2.80E-04	2.68E+31	7.51E+27	1.66E+25
misex3c	622	14	14	1163	1056	28	10	227.94	2.20E-04	1.40E+30	3.08E+26	1.35E+24
mux	55	21	1	55	130	42	1	0.06	1.40E-04	5.40E+01	7.56E-03	1.26E-01
my_adder	268	33	17	1310505	420	66	17	1.72	2.20E-04	2.10E+75	4.62E+71	2.69E+71
o64	131	130	1	130	520	260	1	0.61	2.70E-04	6.90E+01	1.86E-02	3.05E-02
pair	1991	173	137	15513	3412	346	45	794.10	7.80E-04	> 1.00E+99	> 1.00E+99	> 1.00E+99
parity	69	16	1	352	124	32	1	0.09	1.20E-04	2.04E+02	2.45E-02	2.72E-01
pcle	84	19	9	122	166	38	9	0.13	1.70E-04	2.37E+11	4.03E+07	3.10E+08
pcler8	126	27	17	234	230	54	16	0.16	1.80E-04	4.72E+25	8.49E+21	5.31E+22
pm1	55	16	13	70	134	32	9	0.06	9.00E-05	8.00E+00	7.20E-04	1.20E-02
rd53	58	5	3	71	116	10	3	0.14	1.00E-04	2.10E+01	2.10E-03	1.50E-02
rd73	147	7	3	235	250	14	3	0.63	1.10E-04	2.01E+06	2.21E+02	3.51E+02
rd84	285	8	4	949	424	16	4	1.56	1.90E-04	1.78E+11	3.39E+07	2.17E+07
rot	808	135	107	8695	1634	270	29	52.46E-04	5.80E-04	2.83E+95	1.64E+92	3.13E+90
sao2	164	10	4	255	322	20	4	1.99	1.50E-04	2.98E+08	4.47E+04	2.24E+04
sct	90	19	15	150	184	38	13	0.20	8.00E-05	2.63E+19	2.10E+15	1.05E+16
seq	1952	41	35	4947	2882	82	17	1120.72	5.30E-04	6.81E+45	3.61E+42	3.22E+39
squar5	180	5	8	409	284	10	6	3.38	1.10E-04	3.59E+13	3.95E+09	1.17E+09
t481	563	16	1	1077	850	32	1	17.12	1.90E-04	7.49E+02	1.42E-01	8.31E-03
table3	1019	14	14	6747	1540	28	11	1504.04	3.60E-04	1.95E+41	7.00E+37	4.66E+34
table5	1007	17	14	6180	1570	34	12	278.25	3.00E-04	1.46E+37	4.38E+33	1.58E+31
tcon	40	17	16	40	114	34	16	0.04	1.10E-04	7.89E+23	8.68E+19	2.17E+21
term1	299	34	10	601	516	68	10	2.57	2.90E-04	5.98E+26	1.73E+23	6.75E+22
too_large	482	38	3	941	82	76	3	0.02	2.10E-04	8.00E+07	1.68E+04	8.40E+05
ttt2	252	24	21	348	446	48	19	6.59	1.80E-04	2.50E+41	4.49E+37	6.82E+36
unreg	119	36	16	160	272	72	16	0.40	1.60E-04	4.68E+33	7.49E+29	1.87E+30
vda	817	17	39	4768	1344	34	19	854.95	2.80E-04	2.04E+61	5.72E+57	6.69E+54
vg2	113	25	8	246	232	50	7	1.32	1.00E-04	2.02E+11	2.02E+07	1.53E+07
x1	412	51	35	438	748	102	26	158.78	2.20E-04	2.95E+02	6.49E-02	4.09E-04
x2	78	10	7	81	142	20	4	0.42	1.00E-04	3.98E+05	3.98E+01	9.49E+01
xor5	18	5	1	46	36	10	1	0.02	8.00E-05	4.10E+01	3.28E-03	1.64E-01

the number of combinations $\binom{n}{m}$ required by the explicit enumeration, where n is the total number of paths and m the *data path coverage* (see column eight). The estimated CPU time to find a path set of maximum coverage using the greedy search is presented in column twelve. Finally, last column shows the CPU time ratio between the greedy search and the MILP solution. CPU time is expressed in seconds.

The total number of processed circuits was 94. The highest complex circuit — in terms of gate count — is duke2 (4704 gates), and the lowest is b11 (11 gates). In terms of the number of inputs, the highest complex circuit is C2670 (233 primary inputs) and the lowest is b1 (3 primary inputs). The maximum coverage was close to the number of primary outputs. The exhaustive SAT search algorithm is not practical when coverage grows, that is when the number of outputs grows. This is the reason because we limit the number of paths on each set of the greedy strategy. Even in such condition, we obtain a significant improvement in computational effort.

7 Conclusions

The computational complexity of functional vector generation to optimize the *data path coverage* in combinational circuits grows exponentially with the number of gates. We propose an efficient methodology to determine functional vectors that exercises paths and maximizes the *data path coverage* of the verification test. We used Mixed Integer Linear Programming (MILP) to implement the proposed methodology. Comparisons with a greedy search strategy demonstrate that our methodology using MILP obtains functional vectors to optimize the *data path coverage* in a very efficient way (in terms of CPU time). The implementation of the proposed methodology is always better than the greedy search when the number of primary inputs and outputs is greater than eight and three, respectively. The CPU time advantage of our solution based in MILP is several orders of magnitude greater than the greedy solution. This reduction allows verifying combinational logic circuits in a practical CPU time, against the approach based on extensive simulation. This is the first ever reported work on functional vector generation to optimize *data path coverage*.

Our methodology has an obvious limitation; as the circuit grows in terms of number of gates, the vector generation takes more CPU time. In order to reduce the CPU time, several works are in progress related to circuit partition and word–level description of the MILP problem.

8 Acknowledgements

References

[1] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide version 3.0," *Rep. Microelectronics Center of North Carolina*, 1991.

[2] P. Rashihkar, P. Paterson and L. Singh, "System–on–a–Chip Verification – Methodology and Techniques," *Kluwer Academic Publishers*, Feb. 2001.

[3] S. Tasiran and K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs," in *IEEE Design & Test of Computers*, vol. 18, no. 4, pp 36–45, July–August 2001.

[4] M. Kantrowitz and Lisa M. Noack, "I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor," in *Proc. of Design Automation Conference*, pp. 325–330, Jun. 1996.

[5] John R. Wallack and Ramaswami Dandapani, "Coverage Metrics for Functional Test," in *Proc. of the 12th IEEE VLSI Test Symposium*, pp. 176–181, 1994.

[6] F. Fallah, S. Devadas and K. Keutzer, "Functional vector generation for HDL models using linear programming and 3-satisfiability," in *Proc. on Design Automation Conference*, Jun. 1998.

[7] Zeng Zhihong, P. Kalla and M. Ciesielski, "LPSAT: a unified approach to RTL satisfiability," in *Proc. on Design, Automation and Test in Europe*, pp. 398–402, Mar. 2001.

[8] Luís Guerra e Silva, L. Miguel Silveira and João Marques-Silva. "Algorithms for solving boolean satisfiability in combinational circuits," In *Proceeding of Design, Automation and Test in Europe*, pp. 526–530, 1999.

[9] Navaeed A. Sherwani, "Algorithms for VLSI Physical Design Automation," *Kluwer Academic Publishers*, Jan. 1993.

[10] Andrew Makhorin, "GNU Linear Programming Kit, Reference Manual" *Department for Applied Informatics, Moscow Aviation Institute*, Moscow, 2003.

[11] R. K. Brayton, R. Rudell, A. L. Sangiovanni Vicentetelli and A. Wang, "MIS: A Multiple Level Logic Optimization System," in *IEEE trans. on Computer–Aided Design of Integrated Circuits and Systems*, pp. 1062–1081, Nov. 1987.