Paper 1.3 at
1st IEEE International Board Test Workshop (BTW'02)
Baltimore, MD, USA – October 2002 (NL-R 22.055)

Incorporating A Ground-Bounce Preventing Constraint into Wiring Interconnect Test Pattern Generation Algorithms

Erik Jan Marinissen[1] Ben Bennetts[2] Bart Vermeulen[1]

[1] Philips Research Laboratories
IC Design – Digital Design & Test
Prof. Holstlaan 4, M/S WAY-41
5656 AA Eindhoven, The Netherlands
{erik.jan.marinissen, bart.vermeulen}@philips.com

[2] Bennetts Associates
Burridge Farm
Burridge, Southampton
SO31 1BY, United Kingdom
ben@dft.co.uk

Abstract

Ground bounce can seriously upset the test operation of a board, especially on boards with many interconnect wires. Ground bounce can be prevented by introducing a so-called *Simultaneous Switching Output Limit* (SSOL) in the test set. In this paper, we describe the problem of adding a "no-SSOL-violations" constraint to existing wiring interconnect test generation algorithms, without negatively impacting their detection and diagnostic properties. We show that the problem is $\mathcal{NP}$-hard. We present an algorithmic approach for this problem, in which we exploit the degrees of freedom to modify an existing test set. Our approach provides a trade-off between compute time and test time.

1 Introduction

In the literature, there is a multitude of test generation algorithms for wiring interconnects [1]. The objective of these test generation algorithms is to generate the smallest possible set of test patterns, with certain minimum detection and diagnostic properties. Fault models that are typically addressed by such test generation algorithms are single-net shorts and multiple-net bridges.

Recently, preventing *ground bounce* has become a new constraint for test generation algorithms. Ground bounce refers to the phenomenon of shifting ground and power voltage levels, for example between the IC-internal ground and power levels and those at the board. Ground bounce can be prevented by introducing an upper limit on the amount of switching activity between consecutive test patterns, known as the *Simultaneous Switching Output Limit* (SSOL).

This paper presents a generic solution that adds the "no-SSOL-violations" constraint to a broad range of interconnect test generation algorithms. The resulting test sets preserve their original detection and diagnostic properties, but in addition also prevent ground bounce from occurring. Our solution offers a trade-off between test time and compute time. At the expense of a possibly lengthy compute job, we can guarantee the minimal set of interconnect test patterns with the abovementioned features. Alternatively, we can efficiently compute a near-minimal test set. The typical problem setting we have in mind is the testing of board-level interconnect wires between 1149.1-compliant Boundary Scan ICs [2] on a Printed Circuit Board (PCB). However, most of the theory described in this paper is also applicable to other situations of testing interconnect wiring, such as with interconnect wires between modules within one IC, multi-conductor cable, etc.

The sequel of this paper is organized as follows. Section 2 describes the typical fault models that are used in this domain, as well as several previously published test generation algorithms with different detection and diagnostic properties. Section 3 introduces the ground bounce phenomenon, why it should be prevented, and how that can be done by means of test sets without SSOL violations. In Section 4, we formally define the problem of adding a "no-SSOL-violations" constraint to an existing test generation algorithm without affecting its detection and diagnostic properties. Subsequently, we explore the three degrees of freedom that we have in modifying existing test sets. Section 5 describes the details of our algorithmic approach to the problem, which consists of two procedures, that both can be solved either exhaustively, or by means of an heuristic. Our approach is illustrated by means of simple examples. As this paper is a preliminary report on this work, we cannot yet publish experimental results on real industrial devices. Section 6 concludes this paper.

2 Prior Work

We consider the situation in which a number of interconnect wires (*nets*) between two or more digital components need to be tested. In order to test these nets, combinations of digital test stimuli are applied to the net inputs, and the responses are observed at the net outputs and compared to expected responses. We assume full controllability over the inputs of these nets, i.e., at the outputs of the components. Likewise, we assume full observability over the outputs of these nets, i.e., at the inputs of the components.

Throughout this paper, we use the following terms for the test stimuli.

- *Test Pattern*. A test pattern is one set of test stimuli that are simultaneously applied in parallel on all nets of the interconnect network. In Figure 1, test patterns are the columns in the set of stimuli. Others [3, 1] refer to test patterns as *Parallel Test Vectors* (PTVs).
- *Code Word*. A code word is the list of test stimuli that are subsequently applied on one individual net. In Figure 1, code words are the rows in the set of stimuli. Others [3, 1] refer to code words as *Sequential Test Vectors* (STVs).

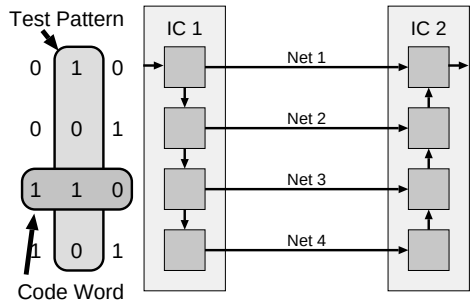

Figure 1: Terminology: *test patterns* and *code words*.

Jarwala and Yau [3] described the following deterministic fault model for wiring interconnects.

- **Multi-net faults**. Bridging faults that create shorts between two or more nets.
  - *Wired-OR*: In case of dominant '1' drivers, the short behaves like a logical 'wired OR' between the shorted nets.
  - *Wired-AND*: In case of dominant '0' drivers, the short behaves like a logical 'wired AND' between the shorted nets.
  - *Strong-Driver*: A specific driver dominates the shorts, and hence the shorted nets follow the dominant driver.
- **Single-net faults**. Stuck-at faults and stuck-open faults on one net. We assume the following possible deterministic behavior for such faults.
  - *Stuck-at-0*: The downstream sensor is such that an open leads to a captured logical '0'.
  - *Stuck-at-1*: The downstream sensor is such that an open leads to a captured logical '1'.

The requirement for testing shorts is that every net must get a unique code word. If the nets are fault-free, each response is unique. In case of a short, the nets involved in that short have the same response. Therefore, these responses are no longer unique and the short is detected. William Kautz [4] proposed a test for shorts which has become known as the *Counting Sequence Algorithm*. The code words are generated by a simple binary counting algorithm. For k nets, $\lceil^2\log k\rceil$ test patterns are required. The Counting Sequence Algorithm guarantees the detection of all shorts with the minimum number of test patterns. In the example of Figure 2(a), there are four nets, i.e., $k = 4$. Hence, we need $\lceil^2\log 4\rceil = 2$ test patterns only. The code words are 00, 01, 10, and 11, which can be arbitrarily assigned to the four nets.

In order to guarantee that every net is tested for stuck-at-0 opens, every code word needs to contain at least one '1'. Likewise, for stuck-at-1 faults, every code word needs to contain at least one '0'. This is not guaranteed by the Counting Sequence Algorithm. Therefore, Goel and McMahon [5] proposed a test generation algorithm that has become known as the *Modified Counting Sequence Algorithm*. The main idea is that open faults can be detected if we forbid the all-zeros and all-ones code words. Hence, for k nets, $\lceil^2\log(k+2)\rceil$ test patterns are needed, which can be generated again by a simple binary counting algorithm, which starts at one, instead of at zero. The Modified Counting Sequence Algorithm guarantees the detection of all shorts and opens with the minimum number of test patterns. In the example of Figure 2(b), with $k = 4$, we need $\lceil^2\log(4+2)\rceil = 3$ test patterns. This yields six possible code words (001, 010, 011, 100, 101, and 110), from which we can arbitrarily choose four and assign them to the four nets.

Another interconnect test generation algorithm, published by Eerenstein and Muris [6], is known as the *LaMa Algorithm*. It is based on the Modified Counting Sequence Algorithm, but increments by three rather than by one. This yields code words with a Hamming distance of at least two, instead of Hamming distance one as is the case for the code words generated by the Modified Counting Sequence Algorithm. This can be proven as follows. All code words generated by the LaMa Algorithm are binary representations of $1 + 3n$ with $n \in \mathbb{N}^+$. Suppose that two code words, say $1 + 3x$ and

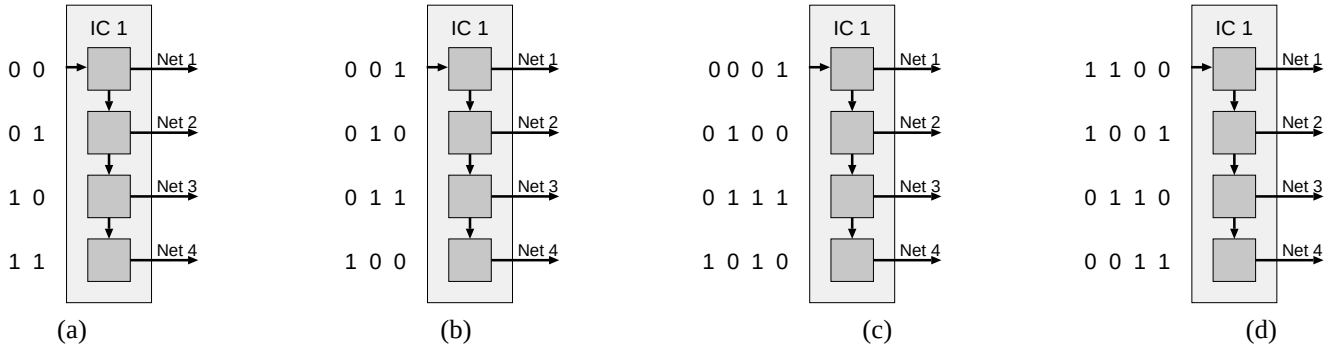

Figure 2: Test stimuli for four nets for (a) Counting Sequence Algorithm [4], (b) Modified Counting Sequence Algorithm [5], (c) LaMa Algorithm [6], and (d) True/Complement Test Algorithm [7].

$1 + 3y$, have Hamming distance one. Then $(1 + 3x) - (1 + 3y) = 3(x - y)$ should be a power of 2, which leads to a contradiction, and hence to the conclusion that the code words generated by the LaMa Algorithm have a Hamming distance of at least two. Eerenstein and Muris claim that a larger Hamming distance between code words increases the chances of detection of a short, especially in the case that neither the '0' nor '1' driver is dominant. The LaMa algorithm requires $\lceil^2\log(3k + 2)\rceil$ test patterns for k nets. In the example of Figure 2(c), with $k = 4$, we need $\lceil^2\log(3 \cdot 4 + 2)\rceil = 4$ test patterns. This yields five possible code words (0001, 0100, 0111, 1010, and 1100), from which we can arbitrarily choose four and assign them to the four nets.

The abovementioned test generation algorithms guarantee detection of faults. However, next to detection of faults, *diagnostic* resolution is often another important property of a test. Jarwala and Yau [3] described the situation of *aliasing*, which obscures the diagnostic resolution. Aliasing occurs if the faulty response of a faulty net is equal to the fault-free response of another, fault-free net. In this case, we cannot determine whether the fault-free net also suffers from the fault at the faulty net. Wagner [7] presented the *True/Complement Test Algorithm*, that enhances the (Modified) Counting Sequence Algorithm in order to prevent from aliasing. It applies all test patterns of the original Counting Sequence Algorithm, followed by the same test patterns with inverted values. Due to the inverted test patterns, the all-zeros and all-ones code words will not be generated. Therefore, it is not necessary to exclude the numbers 0 and $k-1$ in the counting sequence. Hence, the True/Complement Test Algorithm requires $2 \cdot \lceil^2\log k\rceil$ test patterns. The True/Complement Test Algorithm guarantees that every code word contains an equal number of zeros and ones, due to its inversion. In case of a 'Wired-OR' fault with another net, in the response word the number of ones is increased and the number of zeros is decreased. Likewise, in case of a 'Wired-AND' fault with another net, the number of zeros is increased and the number of ones is decreased. In case of a stuck-at open fault, all response values will be either all-zeros or all-ones. Hence, in every fault case the numbers of zeros and ones changes, such that the response word does not contain an equal number of zeros and ones any more. Therefore, a faulty response will never be equal to a fault-free response of another net, and hence, the True/Complement Test Algorithm is aliasing-free. In the example of Figure 2(d), with $k = 4$, we need $2 \cdot \lceil^2\log 4\rceil = 4$ test patterns. The code words are 1100, 1001, 0110, and 0011, and they can be arbitrarily assigned to the four nets.

3 Ground Bounce and its Prevention

During the regular operation of an IC, there is much internal switching activity amongst the transistors. Too much activity can cause surges in current demand from the power-supply and cause the IC-internal power or ground levels to fluctuate considerably. This in turn, can cause spurious pulses to be generated on critical signals, such as system clocks or master resets, thereby causing spurious behavior in the IC. *Ground bounce* is the term used for the phenomenon of varying ground and power levels. In the context of ICs on a PCB, ground bounce typically refers to a shift from the IC's internal ground or power voltage level compared to the PCB's ground or power level [8, 9].

Ground bounce can have a negative impact on the operation of the IC. If the IC-internal ground voltage level rises to the threshold voltage of the IC-internal logic, there exists the likelihood of invalid logic behavior. In the case of Boundary-Scan-equipped ICs [2], ground bounce can result in spurious test clocks (TCKs), which, in turn, can cause on-board Boundary-Scan devices to go out of synchronization.

Ground bounce can be prevented by designing ICs (or boards) such that they can handle large amounts of switching activity within the IC (or between ICs on the board). The switching activity during test is typically higher than during normal (non-test) operation. By its very nature, testing sets up sensitive paths through a circuit thereby causing many

outputs to change value. In order to save test time, it is often an objective of test generation algorithms to obtain the highest possible fault coverage with a minimum number of test patterns. This increases still further the amount of internal switching activity per test pattern. Designers do design their ICs and boards to handle ground bounce during normal operation and during test operation. In fact, the IEEE 1149.1 Boundary-Scan Standard [2] mandates that there should be no ground-bounce inside the device during all test operations based on the use of the boundary-scan register. The problem is that at board level, with all on-board boundary-scan devices in EXTEST test mode for interconnect testing, there is no guarantee that board-level ground bounce will not occur, even though each device is within its ground bounce specification. In fact, there is a high probability that ground bounce will occur if there are more than a few thousand Boundary-Scan to Boundary-Scan interconnects.

Richter and Münch [8] reported on ground bounce problems during EXTEST testing between 1149.1-compliant ASICs on an industrial telecommunications board. All the ASICs met their electrical ground bounce specification and simulation confirmed correct operation of the ASICs under normal operation with at most 50% of the ASIC outputs switching simultaneously. The ASIC designers did not consider the possibility that many (or even all) boundary-scan outputs might switch simultaneously (worst-case EXTEST), even though that was what happened during board testing.

Ground bounce can be prevented at board level by controlling the degree of switching activity in the test patterns generated and applied across the interconnects. This can be done by introducing an upper limit on how many Boundary-Scan outputs can be simultaneously switched in the transition from one test pattern to the next. In this paper, we refer to this upper limit as the *Simultaneously-Switching Outputs Limit* (SSOL). In general, the SSOL should be derived and defined by the board designer responsible for the electrical design of the board. Many vendors of board-level interconnect ATPG tools have added SSOL constraints into their products [8].

The SSOL constraint restricts the maximum Hamming distance between two consecutive test patterns. However, a certain minimum amount of switching is necessary in order to achieve the desired detection and diagnostic levels. Therefore, incorporating an SSOL constraint into test generation can lead to an increase in the number of test patterns generated, and hence to an increase in test time. The objective of this paper is to show how a user-defined SSOL constraint can be added to any interconnect test generation algorithm without jeopardizing the ability to meet other test constraints, e.g., the ability to detect and unambiguously locate opens and shorts, and minimum or near-minimum number of test patterns.

4 Problem Definition

A central notion in our problem definition is formed by SSOL violations, which are defined as follows.

Definition 1 [SSOL Violation].
Let p_1 and p_2 be test patterns, and let $ssol \in \mathbb{N}$. If p_1 and p_2 are applied consecutively, they have a SSOL violation $sv(p_1, p_2)$, defined by $sv(p_1, p_2) = \max(h(p_1, p_2) - ssol, 0)$, where $h(p_1, p_2)$ represents the Hamming distance of test patterns p_1 and p_2. □

The problem addressed in this paper is to incorporate the constraint of SSOL-violation-free consecutive test patterns into any given test generation algorithm. It can be described as follows.

Problem 1 [SSOL-Constrained Test Generation].
Given are (1) a test generation algorithm that generates interconnect test patterns for a user-defined number of k nets, with certain detection and diagnosis properties, and (2) a user-defined integer *ssol*. Find a test that (1) has the same detection and diagnostic properties as the tests generated by the original algorithm, (2) for which any pair of consecutive test patterns p_1 and p_2 holds $sv(p_1, p_2) = 0$, and (3) has a minimal number of test patterns. □

We solve this problem by analyzing and exploiting the degrees of freedom that we have for modifying an interconnect test set without altering the detection and diagnostic properties of such a test. We distinguish the following three degrees of freedom.

1. *Code Word Subset Selection*.
   For k nets, a test generation algorithms requires $p(k)$ test patterns. With $p(k)$ test patterns, the algorithm generates $c(k)$ unique code words, such that $k \leq c(k)$. The functions $p(k)$ and $c(k)$ depend on the test generation algorithm. The expressions for k, $p(k)$, and $c(k)$ for the four test generation algorithms described in Section 2 are as follows.

   - Counting Sequence Algorithm [4]:
     $p(k) = \lceil ^2\log k \rceil$
     $c(k) = 2^{p(k)} = 2^{\lceil ^2\log k \rceil}$
   - Modified Counting Sequence Algorithm [5]:
     $p(k) = \lceil ^2\log(k+2) \rceil$
     $c(k) = 2^{p(k)} - 2 = 2^{\lceil ^2\log(k+2) \rceil}$
   - LaMa Algorithm [6]:
     $p(k) = \lceil ^2\log(3k+2) \rceil$
     $c(k) = \lceil (2^{p(k)} - 2)/3 \rceil = \lceil (2^{\lceil ^2\log(3k+2) \rceil} - 2)/3 \rceil$
   - True/Complement Test Algorithm [7]:
     $p(k) = 2 \cdot \lceil ^2\log k \rceil$
     $c(k) = 2^{p(k)/2} = 2^{\lceil ^2\log k \rceil}$

In many practical cases, the situation arises that the algorithm generates more code words than strictly necessary, i.e., $k < c(k)$. k, $p(k)$, and $c(k)$ are all integers, and hence the expressions above have ceil operators that round off any non-integer values to the nearest higher integer value. These ceil operators cause that in many cases $k < c(k)$. An example is the Counting Sequence Algorithm for $k = 5$. In that example, $p(k) = 3$, $c(k) = 8$, and hence $k < c(k)$.
The degree of freedom is that we can freely choose a subset of k unique code words out of the available $c(k)$ code words generated by the test generation algorithm, without affecting either the detection and diagnostic properties of a test, or its test time.

2. *Re-Ordering of Test Patterns*.
Nets do not contain memory elements. Therefore, neither the detection and diagnostic properties of a test set, nor its test time, are affected by modifying the order in which the test patterns of a test set are applied.

3. *Additional Test Patterns*.
The detection and diagnostic properties of a test set depend on the test patterns in that test set, but are not affected by adding other, new test patterns, nor by repeating some of the existing test patterns. Adding test patterns *does* obviously increase the test time.

These three degrees of freedom are utilized in the next section to solve our problem of incorporating the SSOL constraint into an existing test pattern generation algorithm.

5 Algorithmic Solution Approach

Our solution approach to the problem defined in the previous section consists of two procedures.

1. *Test Pattern Ordering and Insertion*.
In this procedure, we assume a given subset of code words and test patterns. The test patterns are ordered in order to minimize the number of SSOL violations. Remaining violations, if any, are solved by inserting additional test patterns. This procedure exploits Degrees of Freedom 2 and 3 regarding re-ordering and addition of test patterns.

2. *Code Word Subset Selection*.
This procedure searches for the subset of code words that leads to the smallest test pattern set without SSOL violations. The procedure exploits Degree of Freedom 1 regarding code word subset selection, and uses the procedure for test pattern ordering and insertion as a subroutine.

Both procedures are described in more detail in the sequel of this section.

5.1 Test Pattern Ordering and Insertion

In this procedure, we assume given a set with $p \times k$ stimuli, i.e., p test patterns and k code words, necessary for testing k interconnect wires. The procedure consists of two steps. In Step 1, we exploit Degree of Freedom 2, and we try to re-order the p test patterns such that the number of SSOL violations is as small as possible. In Step 2, we resolve the remaining SSOL violations, if any, by exploiting Degree of Freedom 3 and inserting a minimal number of additional test patterns.

Initially, we construct a so-called *SSOL Violations Graph*. This is a weighted, fully-connected, and undirected graph. The nodes of the graph correspond to the p test patterns. The edge between nodes p_1 and p_2 has a weight $sv(p_1, p_2)$, where function sv is defined according to Definition 1. The idea is that the weight of an edge represents the number of SSOL violations in case the two test patterns connected by that edge are applied after one another.

In Step 1, the problem is to find an ordering of the test patterns, such that the number of SSOL violations is minimized. In terms of our SSOL Violations Graph, this means that we need to find a tour through all nodes of the graph, such that the summed weight of the edges encountered along the tour is minimal.

This problem is equal to the optimization variant of the well-known *Traveling Salesman Problem* [10].

Problem 2 [Traveling Salesman Problem (TSP)].
Given a set C of m cities, with distances $d(c_i, c_j) \in Z^+$ for each pair of cities $c_i, c_j \in C$, and a positive integer B. Is there a tour of C having length B or less? □

TSP is known to be $\mathcal{NP}$-complete, and hence its optimization variant is $\mathcal{NP}$-hard [10]. In practical terms, this means that the time needed to compute an optimal solution increases exponentially with the problem instance size. The problem instance size is determined by the number of test patterns. Fortunately, the problem instance size for board-level interconnect testing is not very large. The number of test patterns is in the order of $^2 \log k$, and hence, even for boards with thousands of nets, we have in between 10 to 20 test patterns. Therefore, it seems feasible for most practical problem instances to solve this problem exhaustively, i.e., by enumerating all possible tours in the graph. Alternatively, there are effective and efficient heuristic algorithms for TSP available in the literature, that are able to solve this problem with close-to-optimal solutions in short compute times [11].

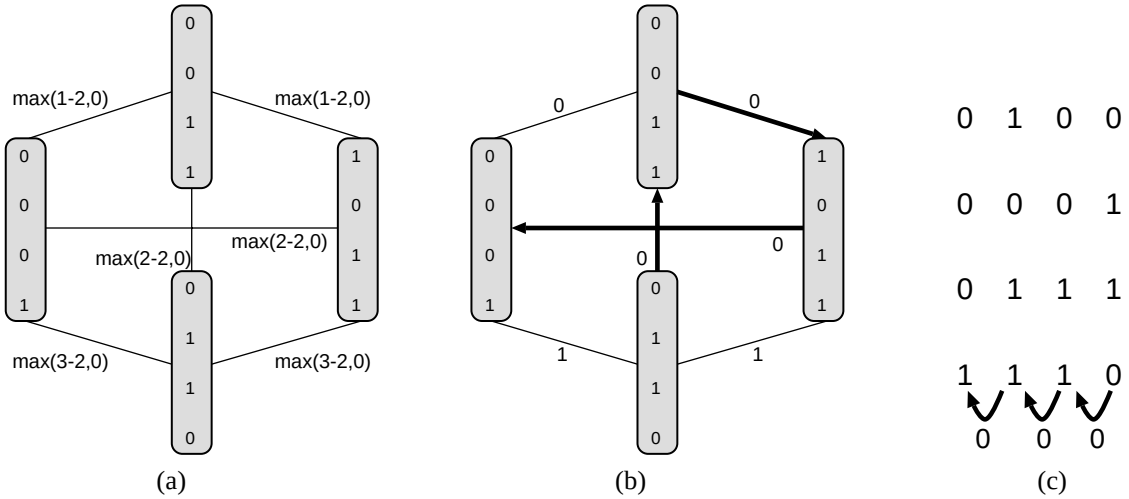

Figure 3: Example of an SSOL Violations Graph with *ssol* $= 2$ (a), the test pattern ordering in this graph (b), and the corresponding resultant test set (c).

Figure 3(a) shows an example of an SSOL Violations Graph. In this example, *ssol* $= 2$. The test set contains four test patterns of four bits each: 0001, 0011, 0110, and 1011. The nodes in the graph are the four test patterns. The edges between two nodes have weights that equal the number of SSOL violations in case the corresponding test patterns are applied consecutively. We can see in this graph that only two out of the six edges have an SSOL violation. Figure 3(b) shows the same graph, in which by means of bold arrows the SSOL-violation-free tour through this graph is indicated. Figure 3(c) shows the corresponding resultant test set, in which the four test patterns are ordered according to the SSOL-violation-free tour in the graph.

After running a TSP Solver, there are two possible outcomes: (1) we have obtained a tour with summed weight zero, or (2) we have obtained a tour with summed weight larger than zero. In the first case, we have right away found an ordering of the test patterns that meets the SSOL constraint. In the latter case, we have found an ordering of the test patterns that still has one or more SSOL violations. This might be due to the fact that there is no tour with summed weight zero, or there is such a tour, but the heuristic TSP solver we used was not capable of finding it. In any case, it is expected that the number of SSOL violations is relatively low, as minimizing the summed weight of the tour was the objective function of the TSP Solver.

Remaining SSOL violations can be resolved by inserting one or more additional test patterns in between two consecutive test patterns that have an SSOL violation. Per inserted test pattern, we can resolve *ssol* remaining SSOL violations. Hence, if consecutive test patterns p_1 and p_2 have $sv(p_1, p_2)$ SSOL violations (with $sv(p_1, p_2) > 0$), we need to insert $\lceil \frac{sv(p_1,p_2)}{ssol} \rceil$ additional test patterns in order to resolve all SSOL violations between these two patterns.

Figure 4(a) shows an example test pattern set, generated by the LaMa Algorithm for $k = 4$. The SSOL violation values are depicted as weights of the arrows at the bottom of the figure. For *ssol* $= 2$, there is only one SSOL violation, between the third and fourth test pattern. This violation is resolved in Figure 4(b) by inserting one additional test pattern in between test pattern three and four.

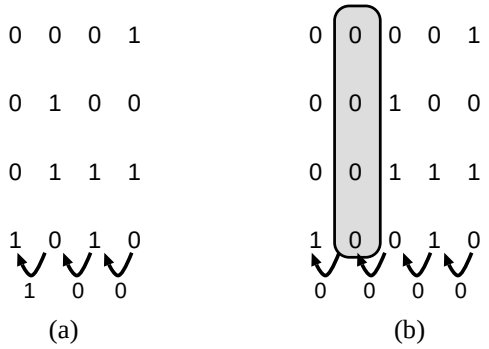

Figure 4: Test pattern set with violation against *ssol* $= 2$ (a) and the violation resolved by inserting one additional test pattern.

5.2 Code Word Subset Selection

A test generation algorithm generates for k nets $c(k)$ unique code words, with $k \leq c(k)$. We only need one unique code word per net. In case $k = c(k)$, we are done, as in that case Degree of Freedom 1 does not provide us any real freedom. However, when $k < c(k)$, we do have actual freedom, as we have to choose which k unique code words we select out of the set of $c(k)$ available code words. This can be done in $\binom{c(k)}{k} = \frac{c(k)!}{k! \cdot (c(k)-k)!}$ ways. In many practical situations, there are many alternative subsets of k code words. As example, consider the True/Complement Test Al-

gorithm for $k = 257$. According to the equation in Section 4, $c(257) = 2^{\lceil 2 \log 257 \rceil} = 2^9 = 512$. Therefore, there are $\binom{512}{257} \approx 4.7 \times 10^{152}$ alternative code word subsets!

There are two alternative ways in which we can address the issue of code word subset selection.

1. Exhaustive Code Word Subset Selection
2. Heuristics-Based Code Word Subset Selection

In the sequel of this section, we describe both alternatives.

The exhaustive code word subset selection is done through exhaustive enumeration through all code word subsets with k elements. This approach is described in Algorithm 1. It consists of an iteration over all code word subsets. For a code word subset, we solve the test pattern ordering problem by means of a TSP Solver as described in Section 5.1. If the TSP Solver yields an ordering of the given test patterns without SSOL violations, we have found a test with a minimal number of test patterns, and hence can abort our exhaustive enumeration algorithm. If the TSP Solver yields a solution with SSOL violations, we calculate how many additional test patterns ap need to be inserted to resolve these SSOL violations. *best_ap* represents the lowest number of additional test patterns needed to resolve SSOL violations for all code word subsets considered so far. If $ap <$ *best_ap*, we make *best_ap* equal to ap, and memorize the current code word subset as the one with the smallest number of test patterns so far.

Algorithm 1 [Exhaustive Enumeration]

```
1   best_ap := ∞;
2   for i = 1 to (c(k) choose k) do {
3       solve 'Test Pattern Ordering' (TSP) for Code Word Subset i;
4       if # SSOL violations = 0 {
5   /*      Hurrah, we have found an optimal solution! */
6           abort
7       }
8       else {
9           ap := # test patterns to be inserted
                  to resolve SSOL violations;
10          if ap < best_ap {
11              best_ap := ap;
12              memorize Code Word Subset i
13          }
14      }
15  }
```

Instead of using the exhaustive enumeration approach, we can also employ an heuristic for code word subset selection. Many different heuristics can be invented. As an example, we describe one such heuristic in the sequel of this section. This heuristic counts the bit transitions in each code word as generated by the test generation algorithm, i.e., before test pattern re-ordering. Out of the available $c(k)$ code words, this heuristic selects k code words such that the sum of their transition counts is as low as possible.

An example that illustrates the operation of this heuristic is depicted in Figure 5. In the example, we used the LaMa Algorithm for $k = 4$ and *ssol* $= 2$. This yields $\lceil 2 \log(3 \cdot 4 + 2) \rceil = 4$ test patterns and $\lceil (2^4 - 2)/3 \rceil = 5$ unique code words. These code words are: 0001, 0100, 0111, 1010, and 1101. We only need four unique code words, and hence the code word subset selection here revolves around the question of *de*selecting one of the five available code words. Counting the number of bit transitions in the five code words shows that code word 1010 has the highest bit transition count, viz. three. Therefore, this code word becomes our candidate for deselection. The remaining code words right away form a test set without SSOL violations, and hence, in this case, no further test pattern re-ordering and/or insertion is necessary.

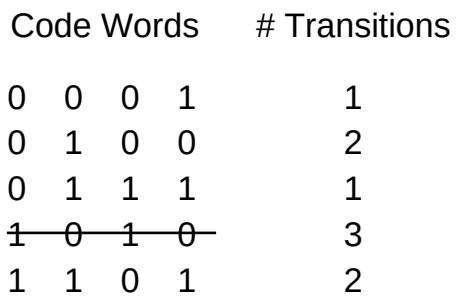

Figure 5: Example of a code word subset selection heuristic, based on the number of bit transitions in the original code words.

Note that the heuristic described above is an example of an heuristic that directly selects one code word subset. Such an algorithm subsequently invokes the procedure for test pattern ordering and insertion only once. One can also think of heuristic approaches that allow for the evaluation of multiple code word subsets. In that case, the general structure of Algorithm 1 can be utilized, i.e., evaluate each code word subset, memorize the best subset so far, and abort the algorithm as soon as a subset is found that does not have any SSOL violations without additional test patterns being inserted. In fact, for code word subset selection, a full range of algorithms is possible, with exhaustive enumeration of all code word subsets as one extreme, and the direct selection of only one single subset as the other extreme.

6 Conclusion

Many wiring interconnect test generation algorithms exist. These algorithms have different detection and diagnostic properties, and also vary in the number of code words and test patterns they produce. In this paper we have described four of such algorithms: the Counting Sequence Algorithm,

the Modified Counting Sequence Algorithm, the LaMa Algorith, and the True/Complement Test Algorithm.

Varying levels of power and ground voltage, commonly known as ground bounce, is a growing concern within ICs, as well as between ICs on boards. Especially during board-level interconnect testing for boards with many wires, ground bounce is a likely phenomenon, that might cause malfunctioning tests. Ground bounce during board-level interconnect testing can be prevented by restricting the number of simultaneously switching IC outputs to a limit, named SSOL, determined by the board designer. This requires upgrading of existing interconnect test generation algorithms, such that they meet this additional SSOL constraint.

In this paper we have presented the problem of finding an algorithm that takes any interconnect test generation algorithm as starting point, and generate a test that (1) maintains the detection and diagnostic properties of the original test generation algorithm, (2) meets the SSOL constraint for a user-defined SSOL value, and (3) requires only a minimal or near-minimal number of test patterns. We showed that there are three degrees of freedom which can be exploited to address this problem: (1) the selection of the set of code words, (2) the re-ordering of test patterns, and (3) the insertion of additional test patterns. We showed that this problem is $\mathcal{NP}$-hard, by showing that one of its subproblems is $\mathcal{NP}$-hard. We have presented an algorithmic approach that provides a trade-off between the resulting 'test time' (i.e., the number of test patterns) and the compute time required to obtain a test set.

Code word subset selection can be done exhaustively, or by means of an heuristic. Exhaustive selection guarantees the best solution possible, but is often more expensive with respect to compute time. The number of alternative code word subsets can be very large. When combined with a low, and hence difficult to achieve SSOL value, this might lead to intractable compute times. An heuristic-based approach costs less compute time, but might lead to additional test patterns that were perhaps not strictly necessary.

Test pattern re-ordering can be done by means of an exhaustive or an heuristic TSP Solver. It is important to note that this procedure is invoked for every code word subset evaluation, and hence, the compute time required by the TSP Solver is increasingly important when more code word subsets are evaluated. Fortunately, the size of the TSP problem instances is relatively small, as it scales logarithmically with the number of nets. Therefore, even for large printed circuit boards, exhaustive TSP solving is still feasible.

Acknowledgements

We thank Gerard Beenker, Lars Eerenstein, Sandeep Kumar Goel, Frans de Jong, Peter Meijer, Math Muris, Harald Vranken, and Wim Verhaegh for their help in preparing this paper.

References

[1] José T. de Sousa and Peter Y.K. Cheung. *Boundary-Scan Interconnect Diagnosis*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[2] IEEE Computer Society. *IEEE Standard Test Access Port and Boundary-Scan Architecture - IEEE Std. 1149.1-2001*. IEEE, New York, NY, USA, July 2001.

[3] Najmi Jarwala and Chi W. Jau. A New Framework for Analyzing Test Generation and Diagnosis Algorithm for Wiring Interconnects. In *Proceedings IEEE International Test Conference (ITC)*, pages 63–70, October 1989.

[4] William H. Kautz. Testing of Faults in Wiring Interconnects. *IEEE Transactions on Computers*, Vol. C-23(No. 4):358–363, April 1974.

[5] P. Goel and M.T. McMahon. Electronic Chip-In-Place Test. In *Proceedings IEEE International Test Conference (ITC)*, pages 83–90, October 1982.

[6] Lars Eerenstein and Math Muris. *Method for Generating Test Patterns to Detect an Electric Shortcircuit, a Method for Testing Electric Circuitry While Using Test Patterns So Generated, and A Tester Device for Testing Electric Circuitry with Such Test Patterns*. June 1997. United States Patent 5,636,229.

[7] Paul T. Wagner. Interconnect Testing with Boundary Scan. In *Proceedings IEEE International Test Conference (ITC)*, pages 52–57, October 1987.

[8] Hans Peter Richter and Norbert Münch. Boundary-Scan Test Triumphs Over Ground-Bounce. *Test & Measurement World Europe*, August/September 1997.

[9] Amitava Majumdar, Michio Komoda, and Tim Ayres. Ground Bounce Considerations in DC Parametric Test Generation using Boundary Scan. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 86–91, April 1998.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability - A guide to the theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, CA, USA, 1979.

[11] David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study. In Emile H.L. Aarts and Jan-Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons Ltd., Chichester, England, 1997.