Nonlinear codes for test patterns compression: the old school way

Jan Schmidt[0000−0003−3221−7370] and Petr Fišer[0000−0001−5306−6343]

Czech Technical University in Prague
{schmidt,fiserp}@fit.cvut.cz
ddd.fit.cvut.cz

Abstract. The problem is to design a nonlinear code for test vectors expander, with the requirement of all r-tuples possible in the output. We formulated the problem as a clique cover problem. The instances of the problem have a high degree of symmetry, which offers the possibility of analytical solution or better heuristic construction. To benefit from the degrees of freedom in the problem, assigning expander inputs to the produced vectors has been identified as an multi-valued (MV) variable encoding problem.
Experimental evaluation shows that good MV encoding is important for small r. Instances up to $n = 32$ and $r = 6$ were solved, with the resulting expander widths i mostly equal to or better than existing solutions. For the synthesis of the expanders, both the classical minimization-decomposition and resynthesis approaches can be used. The produced circuits were larger than corresponding linear expanders.

Keywords: Test compression · Nonlinear code.

1 Introduction

When a digital device is tested, one of the problems is to deliver test stimuli economically. Suppose the device is equipped with n scan chains. Then, the n-bit test vectors must be delivered from outside (from a tester), or generated internally.

A substantial help comes from the fact that test stimuli have large redundancy [1], and can be efficiently compressed. Two major test delivery architectures employ this fact.

The first approach uses a vector stream with the same number of vectors but with minimum redundancy (and therefore with minimum width). The stream is then processed by a combinational circuit often called *combinational expander* or *combinational decompressor*. Generally, there are no additional requirements on the Boolean functions the expander performs, except to deliver the required vectors.

Another approach is to let an FSM generate the required vectors, or their superset. For this to work, the set of vectors must have certain properties, e.g., to be a subset of a linear space. Many FSM classes have been used for this

purpose, mostly an LSFR, but also Cellular Automata (CA) [10], or Registers with Non-Linear Update (RNLUs) [6]. Such methods are characteristic for Built-In Self-Test (BIST) applications. In all these cases, the properties of the FSM transition function are of concern.

These two approaches are just extremes of a broad spectrum; combined methods such as reseeding [5], bit-flipping [16], bit-fixing [15, 14], Embedded Deterministic Test [12], etc., are numerous. As they have state, they are commonly called *sequential decoders*.

The compression scheme can be either application-dependent, or universal for a class of applications with the same number of scan chains and (approximately) the same redundancy in their test sets. The former is undesirable, as the construction of a good compression scheme can be demanding. A class of applications can be specified by

- the number n of scan chains, and
- the requirement that r bits can be set to arbitrary values in each expanded vector.

In the contribution, we will limit ourselves to combinational expanders specified by the above requirements. An example test scenario is shown in Figure 1. Here compressed test patterns are stored in the tester device (ATE) with i channels. These patterns are then decompressed on-chip to be fed to n ($n > i$) scan chains of the Circuit under Test (CUT). Responses to these patterns are then evaluated (typically also on-chip).

Kim and Mitra [8] brought the idea that introducing redundancy corresponds to encoding a symbol in an error-correcting code. They showed that linear codes, such as BCH codes, are effective for this purpose, and that the r-bit requirement can be translated to a Hamming distance requirement for the dual code. This way, expanders can be easily constructed using existing code tables. They list compressed vector lengths i that can satisfy the r-bit requirement in n scan chains, using an undisclosed linear code for $r = 3$ or BCH codes for $4 \leq r \leq 8$, or Reed-Solomon codes for large n in the case of identified clusters in the test vectors.

An extension to codes other than linear is obvious. It gives much more freedom to choose the expander function, but such freedom also translates to much bigger search space. Moreover, there is not such a wealth of existing knowledge as in the case of linear codes. And, last but not least, nonlinear codes that are efficient for error corrections are not guaranteed to be efficient for test vector expansion.

Many efforts come from the BIST domain. Dutta and Touba [3] limit the search space by considering only a limited class of circuits. Novák [9] extends linear codes by non-linear expander outputs. The stochastic search over the complete search space in [11] brought functions that are remarkably efficient. The authors generate a truth table randomly under certain stochastic requirements, and then check the r-bit requirement. Functions for larger r are then composed.

These researches seem to state that "with i tester channels and the r-bit requirement, my code can accommodate up to n scan chains". We believe that the

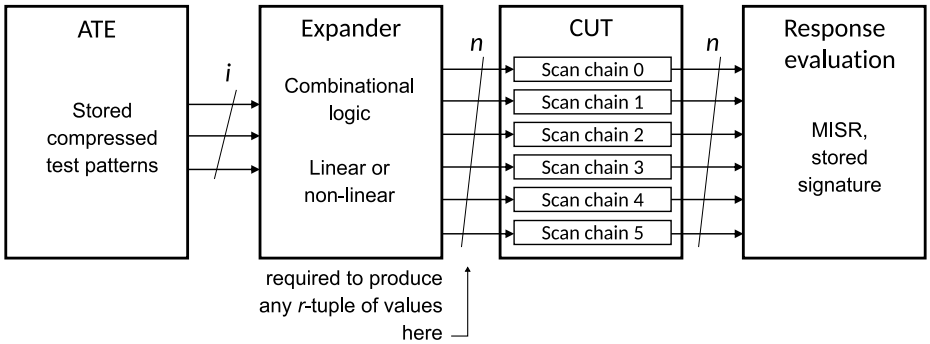


Fig. 1: An example test scenario for three ATE channels and six scan-chains

question in design time is rather "with n scan chains and the r-bit requirement, what is the minimum number i of tester channels?". While the optimization task is the same, the difference must be regarded when comparing.

Definition 1 (Expander Function). *Given integers i, n, r, where $r < n$ and $i < n$, an* expander function *is a function $f_{i,n,r} : \{0,1\}^i \to \{0,1\}^n$, such that, for all ordered r-tuples P of positions from $[0, n]$ and for all valuations $V \in \{0,1\}^r$ of these positions, there exist a vector $\mathbf{x} \in \{0,1\}^i$ such that the vector $f_{i,n,r}(\mathbf{x})$ has the values V at positions P.*

Definition 2 (Expander minimization). *Given integers n, r, $r < n$, find an integer i such that there exists an expander function $f_{i,n,r}$.*

In the proposed approach, we formulate all requirements to the expander function first. Any function that satisfies the requirements is therefore a correct solution. Then, we use synthesis tools to get an optimized implementation of the expander.

The paper is organized as follows. In Section 2 we formulate the problem as a Clique Cover Problem, followed by multivalued variable encoding, and logic synthesis. We analyze instance properties in Section 2.2 and outline a simple heuristic in Section 2.3. The concrete methods used and their results are described in Section 3. We outline results we hope for in Section 4.

2 Proposed approach

We specify all outputs the expander must produce as the output part of its (incompletely specified) function. Then, we are free to construct any input part to optimize the circuit, without affecting the correctness of function. Last, we synthesize the circuit.

2.1 Expander outputs as a clique cover problem

We construct requirements (or constraints) on the expander first. The r-bit requirement tells us that, for every r-tuple of the n expander outputs, all 2^r combinations of binary values must be present for at least one input value. Such constraint can be expressed as 2^r cubes of dimension $n-r$, called *requirement cubes*. The collection of all such cubes completely specifies the output of the expander as the set of *output cubes* of its expander function. Formally,

Definition 3 (Requirements Cube). *Given integers n, r, where $r < n$, an ordered r-tuple P of positions from $[0, n]$, a valuations $V \in \{0,1\}^r$ of these positions, a* requirement cube $\rho_{P,V}$ *for P and V is a subset $\rho_{P,V} \subset \{0,1\}^n$, obtained from $\{0,1\}^n$ by setting values from V in dimensions given by P. It is therefore a cube of dimension $n-r$.*

During the operation of the expander, we do not need to distinguish between all requirement cubes. For every valuation of every r-tuple, there must be an input to the expander that produces those values at the output. Therefore, cubes that intersect can be replaced by their intersection. The reduced number of distinguished outputs saves the resulting expander width.

Understanding the intersection as compatibility, we can construct a compatibility graph and then treat it as a covering problem. All compatible cubes form a clique. As the intersection of two cubes is a cube, each clique is also characterized by a cube. Any clique cover of the graph is a valid set of all output vectors of the expander function. The input width i is the logarithm of the number of output vectors, and does not depend on further optimizations. Therefore, we seek a minimum cover. Figure 2 shows a rather small example.

To obtain the Boolean function of the expander, it is sufficient to number the cubes, and encode the numbers in any way. Then, the encoded numbers will form the input part of incomplete function specification, and the cubes the output part. Technically, this is a PLA-format specification, as defined by two-level minimizers [13].

2.2 Compatibility graph properties

By construction, the compatibility graph has

$$N_R = \binom{n}{r}.2^r \tag{1}$$

nodes.

Let c be any cube of dimension 0, that is, completely specified. It is a clique, because we can find requirement cubes whose intersection is exactly c. Therefore, we have $N_C = 2^n$ such cliques. The number of cubes that have the specified variables the same as the clique c, and hence the clique size, is $S_C = \binom{n}{r}$.

Let R be the set of all requirement cubes and let $\rho \in R$ be a requirement cube. Now let us count other compatible request cubes. By construction, ρ contains r

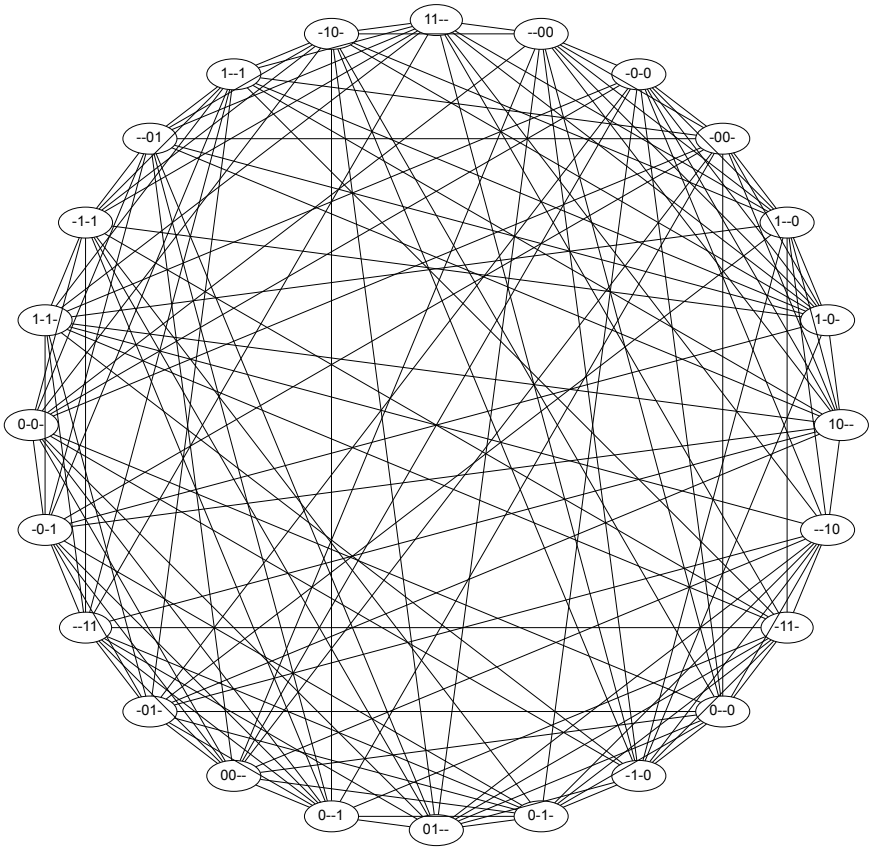


Fig. 2: The compatibility graph for $n = 4$ and $r = 2$

care bits. To be compatible, a cube must contain either the same values or don't cares at those places. Let us choose j places of ρ. We have $\binom{n}{r}$ possibilities. As the compatible cube must have r care places, there are $\binom{n-r}{r-j}$ possibilities for the positions of the remaining $r - j$ care bits. As each of them may be 0 or 1, we finally get the number of cubes compatible with any given cube as

$$N_{COMPAT} = \sum_{j=0}^{r-1} \binom{r}{j} \binom{n-r}{r-j} 2^{r-j} \tag{2}$$

Given an r-tuple of positions in the cube and their values, by construction there is another cube with the same positions but inverted values. This leads to the following:

Definition 4. *Let c be a cube. Then $inv(c)$ is a cube obtained from c by inverting all care values.*

Definition 5. *Let c be a cube in B^n and let P be a permutation of the sequence $1 \ldots n$. Then $perm(c, P)$ is a cube obtained from c by permuting all positions of c according to P.*

Definition 6. *Let C be a set of cubes. Then $inv(C)$ is a set of cubes $\{\forall c \in C : inv(c)\}$.*

Definition 7. *Let C be a set of cubes in $\{0,1\}^n$ and let P be a permutation of the sequence $1 \ldots n$. Then $C^p erm(C, P)$ is a set of cubes $\{\forall c \in C : perm(c, P)\}$.*

Theorem 1. *Let R be a set of all requirement cubes for given n and r. If a set of cubes C covers (resp. does not cover) R, then so does $inv(C)$ and $perm(C, P)$, $\forall P$.*

The proof follows from the facts that 0 and 1 are treated the same (their ordering is immaterial) and also, that the numbering of positions is immaterial. Furthermore,

Theorem 2. $\forall C, P : inv(perm(C, P)) = perm(inv(C), P)$.

Therefore,

Theorem 3. *Let R be a set of all requirement cubes for given n and r. Then, for any set of cubes C, $\{C, inv(C), \forall P : perm(C, P), \forall P : perm(inv(C), P)\}$ is an equivalence class with respect to covering R.*

Equation 2 and Theorem 3 state that the problem is highly regular and symmetric, which gives some hope to find either the minimum clique cover size N_{OPT} analytically in the future, or to use Theorem 3 to prune a search efficiently.

In Figure 2, there are $N_R = 24$ requirement cubes. Each of the cubes is compatible with $N_{COMPAT} = 12$ other cubes. There are 16 distinct minimum covers with $N_{OPT} = 5$ and hence $i = 3$. These solutions form 3 classes of equivalence.

2.3 Techniques for Minimum Clique Cover

Very small instances can be solved by brute force, which can give some insight into the problem. Because we know that the covers are not large for small instances, we constructed Algorithm 1 which tries to construct a cover of a given size s. Furthermore, it uses only cubes of size n (that is, completely defined) for the cover.

As Eq. 1 shows, compatibility graphs tend to be large even for small n and r. Therefore, we sought a way to get the clique cover without *storing* requirement cubes explicitly, in an on-line fashion. This of course does not guarantee optimality, as it is a sort of greedy technique. The procedure can be outlined as Algorithm 2. Notice that this is still a kind of meta-algorithm: the order of tuple selection in Line 2 and values selection in Line 3 is not defined. Also, the algorithm is satisfied with the first candidate clique that covers a given cube (Line 7).

Algorithm 1 Cube generation and exact clique cover

Input: n, r, s ▷ s is the tried cover size
Output: A set $\{C\}$ of all clique covers C

Let R be an empty set of requirement cubes.
for all ordered r-tuples P_R from $1 \dots n$ **do**
    for all $V \in B^n$ **do**
        construct a cube q having values from V at places P_R and dont'cares otherwise.
        insert q into R.
    end for
end for
Let $\{C\}$ be an empty set of covers.
for all ordered s-tuples P_C from $1 \dots 2^n$ **do**
    let C be an empty cover.
    for all members p of P_C **do**
        construct a cube c from the binary representation of p
        insert c into C
    end for
    if C covers R **then**
        insert C into $\{C\}$.
    end if
end for

Algorithm 2 Cube generation and greedy clique cover

Input: n, r
Output: A clique cover C.

Let C be empty.
for all ordered r-tuples P from $1 \dots n$ **do** ▷ in some order
    for all $V \in B^n$ **do** ▷ in some order
        construct a cube q having values from V at places P and dont'cares otherwise
        for all cubes $c \in C$ **do**
            if c is compatible with q **then**
                replace c with $c \cap q$
                break
            end if
        end for
        if q still uncovered **then**
            insert q into C
        end if
    end for
end for

2.4 Expander input as an MV-encoding problem

Once we have the output part of the two-level specification of the expander function, we are free to choose a distinct input pattern for each of the output cubes. From the possible 2^i values, only N_{OPT} are used. By the construction of i, $2^{i-1} < N_{OPT} \leq 2^i$. It means that up to a half of the values are unused.

We can treat the expander inputs as a multi-valued (MV), symbolic variable. Then the problem is how to encode it to produce minimum circuit. At a first glance, this is similar to the opcode encoding or state encoding problem [13]. The difference, especially from the state encoding problem, is that here the variable is an external input variable. Available encoding algorithms cannot benefit from the degree of freedom. Nevertheless, the constraints from such encoders can be used in a specialized algorithm.

2.5 Method summary

Given the encoder specification, the expander can be synthesized. Starting the synthesis by a two-level description, which does not suggest circuit structure, is unusual in contemporary practice (cf. [4]). In this situation, the classical minimization-decomposition approach, e.g, using BDS [17], seems worth consideration. However, any contemporary logic optimization tool accepting such description at its input can be used, e.g. ABC [2].

Our method can be outlined as Algorithm 3. An example description obtained in Steps 10 and 11 is in Figure 3.

Algorithm 3 Method overview

1: For every value of every r-tuple in an n bit vector, construct a *requirement cube* R_i stating that those values are output, and nothing else.
2: Construct a *compatibility graph* $G(R, E)$, such that R is the set of all requirement cubes and there is an edge between requirements cubes R_1 and R_2 iff they intersect.
3: Solve Minimum Clique Cover on G. Let the number of cliques be N_{OPT}.
4: The necessary input width is $i = \lceil \log_2 N_{OPT} \rceil$.
5: Collect all clique cubes as the output part of the two-level description.
6: **if** MV optimization is available: **then**
7: Choose N_{OPT} distinct symbolic values for the left-hand side, providing an MV description.
8: Perform MV minimization and encoding on the MV description.
9: **else**
10: Choose N_{OPT} distinct binary combinations for the input part, providing a two-level description.
11: Perform minimization on the two-level.
12: **end if**
13: Synthesize the resulting two-level description by any method, possibly technology-dependent.

```
0000 0000000000
0001 0000111110
0010 1111000001
0011 1111111111
0100 0110000111
0101 1001111000
0110 0101010011
0111 1010101100
1000 -0--0-1--1
1001 -1--1-0--0
```

(a) as generated by Algorithm 2

```
1--0 0000001001
1--1 0100100000
00-1 0000111110
-100 0110000111
-101 1001111000
-110 0101010011
-01- 1111000001
--11 1010101100
```

(b) after minimization

Fig. 3: two-level description of an $n = 10$, $r = 2$ expander

3 Results

With the techniques described above, we solved instances up to $n = 32$ and $r = 6$. Besides the comparison of the resulting widths, we also investigated the importance of good MV encoding, synthesis approaches and expander circuits properties.

3.1 Implementation

Algorithms 2 and 1 have been implemented as a sequential C++ program and run on an office machine (Intel i7, 8 cores at 3GHz, SUSE Linux). The ordering in Line 2 of Algorithm 2 has been implemented as ordered random sampling ([7], p. 166). The ordering in Line 3 is systematic. A variant of the algorithm, which found a cube c maximally intersecting p (Line 7) has been tested with no improvement.

To illustrate the influence of randomization, Algorithm 2 has been run 500 times for $n = 20$ and $r = 4$. The resulting histogram is in Figure 4. Random selection does have an influence on the obtained suboptimum cover size N_{SUB}, however, it is unlikely that the differences cause a difference in i.

3.2 Resulting codes

Table 1 comapres Algorithms 1 and 2. We can see that the heuristic operates within 150% of the optimum cover size. The number of solutions and classes differ wildly, indicating that the characters of the instances also differ. It seems that the gap between greedy and exact solutions closes with larger instances.

Table 2 summarizes instances and results obtained from Algorithm 2. Although instance parameters show that the element counts grow rapidly, clique cover sizes remain relatively small.

Table 3 contains expander widths i reported by various authors in comparison to results of Algorithm 2. The table is limited to comparable values of n and r,

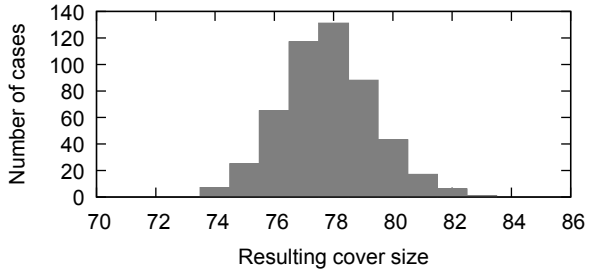


Fig. 4: N_{SUB} frequencies in 500 runs of Algorithm 2 for $n = 20$ and $r = 4$

Table 1: Exact and heuristic solutions of small instances

		Exact				Heuristic	
n	r	i	N_{OPT}	number of solutions	number of classes	i	N_{SUB}
4	2	3	5	16	3	3	6
4	3	3	8	2	2	4	12
5	2	3	6	2896	36	3	8
5	3	4	10	16	3	4	15
6	2	3	7	2080192	1958	3	8
7	2	3	8	2845462	–	3	8

Table 2: Instance properties, resulting widths i and achieved cover sizes N_{SUB} from Algorithm 2

n	r	N_R	S_C	i	N_{SUB}	n	r	N_R	S_C	i	N_{SUB}
10	2	180	45	4	10	28	2	1512	378	4	14
10	3	960	120	5	21	28	3	26208	3276	6	35
10	4	3360	210	6	52	28	4	327600	20475	7	92
10	5	8064	252	7	118	28	5	3144960	98280	8	231
10	6	13440	210	8	215	28	6	24111360	376740	10	569
16	2	480	120	4	12	30	2	1740	435	4	14
16	3	4480	560	5	27	30	3	32480	4060	6	37
16	4	29120	1820	7	69	30	4	438480	27405	7	97
16	5	139776	4368	8	165	30	5	4560192	142506	8	244
16	6	512512	8008	9	365	30	6	38001600	593775	10	596
20	2	760	190	4	12	32	2	1984	496	4	14
20	3	9120	1140	5	30	32	3	39680	4960	6	37
20	4	77520	4845	7	74	32	4	575360	35960	7	98
20	5	496128	15504	8	194	32	5	6444032	201376	8	248
20	6	2480640	38760	9	443						
24	2	1104	276	4	14						
24	3	16192	2024	6	34						
24	4	170016	10626	7	84						
24	5	1360128	42504	8	211						
24	6	8614144	134596	10	513						

so that results in the practical range of n in the hundreds are not included. The presented algorithms, due to combinatorial explosion, will never be able to work in that range.

The results in [11] seem to form two groups: one with n above 40, and the other group below. The proposed algorithm is able to match the second group. It also outperforms linear BCH codes, and optimum linear codes for larger n.

Table 3: Expander widths comparison

		width for code							width for code				
n	r	linear [11] [8]	BCH [8]	NBC1 [11]	NBC2 [11]	proposed	n	r	linear [11] [8]	BCH [8]	NBC1 [11]	NBC2 [11]	proposed
8	3	4				5	12	5	8				8
12	3	4				5	13	5				7	8
16	3	5				5	18	5			8		8
32	3	6				6	24	5	10			8	8
8	4	6				5	11	6	9				8
10	4	7				6	13	6			9		9
13	4	8				6	16	6			9		9
14	4	6		6		6	17	6	10				9
18	4	6			6	7	32	4		12			7
9	5	7				7	32	5		18			8
10	5			7		7	32	6		18			15

3.3 Is optimum MV encoding important?

We have found no way to use MV optimizers to design compressed test encoding, that is, the input part of the two-level description. For the optimizers, the symbolic input part is an external, given information. To estimate how important the encoding is, we arranged the following experiment.

For every code obtained by Algorithm 2, 500 random input parts were generated randomly. Each two-level description was optimized by Espresso [13]. The resulting minimized descriptions were characterized by the total number of literals. Statistical characterization of the result is in Table 4.

The expected result is that the size of the encoder depends on input encoding. This happens, at least in some cases – in the case of $n = 10$ and $r = 2$, the difference between the worst and the best result spans 66% of the median. The surprising fact is that this influence rapidly diminishes with increasing r. There is a strong correlation of -0.9. No other such correlation has been found. It is true that the cover does not use all 2^i values, and that the number of unused values differs from instance to instance. Yet the spread does not correlate with that at all (less than 0.1).

Table 4: Number of literals after minimization, random encodings. m is the median of the distribution, and Δ is the difference between max and min

n	r	min	m	max	σ/m	Δ/m	n	r	min	m	max	σ/m	Δ/m
10	2	35	57	73	0.07	0.667	28	2	134	182	216	0.047	0.451
10	3	114	153	178	0.052	0.418	28	3	508	564	606	0.013	0.174
10	4	339	412	501	0.041	0.393	28	4	1637	1738	1773	0.009	0.078
10	5	946	1063	1272	0.026	0.307	28	5	4747	4851	4892	0.003	0.03
10	6	1870	2040	2408	0.017	0.264	28	6	12169	12293	12390	0.002	0.018
16	2	68	100	119	0.05	0.51	30	2	132	193	228	0.028	0.497
16	3	247	298	325	0.023	0.262	30	3	579	662	697	0.021	0.178
16	4	723	845	889	0.017	0.196	30	4	1834	1951	2002	0.009	0.086
16	5	1987	2251	2323	0.009	0.149	30	5	5244	5398	5465	0.004	0.041
16	6	5333	5474	5594	0.005	0.048	30	6	13331	13474	13574	0.002	0.018
20	3	329	385	413	0.019	0.218	32	3	620	704	738	0.015	0.168
20	4	1018	1090	1112	0.008	0.086	32	4	1988	2090	2128	0.007	0.067
20	5	3043	3176	3242	0.006	0.063	32	5	5666	5787	5850	0.003	0.032
20	6	7748	7882	7970	0.003	0.028							
24	3	428	497	529	0.025	0.203							
24	4	1314	1412	1452	0.009	0.098							
24	5	3852	3954	4002	0.004	0.038							
24	6	9791	9929	10004	0.002	0.021							

3.4 Expander synthesis

The starting point of the synthesis is a two-level description, which is unusual. We therefore chose two alternative approaches to synthesis. The first one is classical; minimization with Espresso [13] followed by decomposition with BDS [17]. The other approach uses only ABC [2] with a rather high effort. The script in Algorithm 4 is iterative as recommended by the authors of ABC. Nevertheless, we tried to combine both approaches and use ABC with the same script as a post-optimizer for Espresso/BDS.

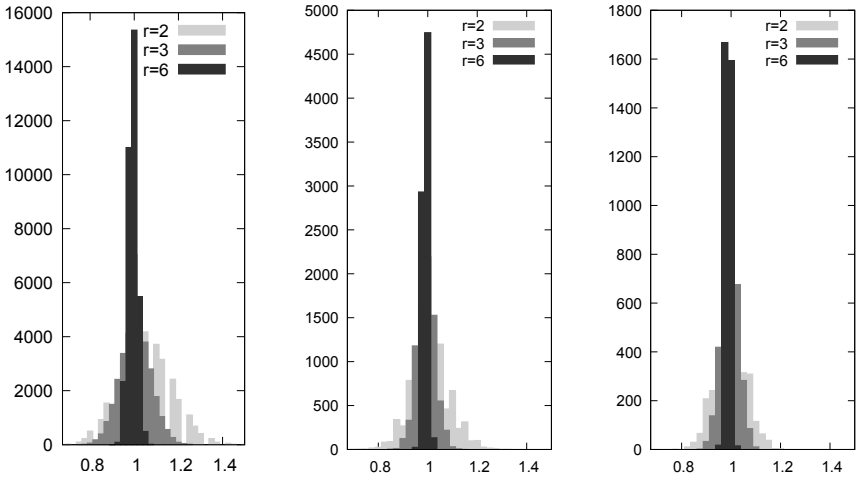
Algorithm 4 Expander synthesis using ABC

Input: in.pla: a two-level description of the expander
Output: out.blif: Optimized BLIF description of the expander

```
1: read_pla in.pla; &get -n                    ▷ convert into the GIA structure
2: for 20 times do
3:     &st; &synch2; &if -m -a -K 2; &mfs -W 10;
4:     &st; &dch; &if -m -a -K 2; &mfs -W 10
5: end for
6: &put; write_blif out.blif
```

The comparison of the two synthesis approaches shows that although the decomposition approach performance is acceptable, iterated synthesis can pro-

vide yet better results. It does not suffer from the absence of structure, which means, that it is able to discover circuit structure independently. ABC as a post-optimizer improves the results of the classical approach considerably, achieving the best results from the three alternatives.



(a) $n = 10$ (b) $n = 20$ (c) $n = 30$

Fig. 5: Gate counts frequencies with random encoding, normalized to median

The synthesis results obtained with random encoding corroborate Section 3.3 in the sense that, the influence of encoding decreases with growing r. As Figure 5 illustrates, this holds also for increasing n to some degree.

The resulting gate counts in Table 5 indicate that, with current methods, the improved expander width is paid for by expander size. To illustrate, a small experiment for $n = 15$ and $r = 4$ is presented in Table 6. A linear expander using the BCH(4,2) code has been synthesized with Espresso and BDS. A nonlinear expander has been generated using Algorithm 2 and binary input encoding. The synthesis flow was the same. Although the BCH(4,2) code is not optimal, the expander is considerably smaller.

4 Future directions

The main hope of this work is to gain knowledge about the problem. As Equation 2 and Theorem 3 indicate high degree of symmetry in all instances, we hope that the problem is not so difficult as the numbers in Table 2 indicate. Any N_{SUB} gives a lower bound for encoder width. To find N_{OPT} analytically would enable us to judge optimality of any code. Another potential benefit of the high symmetry could result in better construction algorithm.

Table 5: Expander implementations, gate counts with random encoding

n	r	ABC			Espresso, BDS			Espresso, BDS, ABC		
		min	median	max	min	median	max	min	median	max
10	2	22	33	49	16	40	69	15	28	48
10	3	69	94	126	74	111	145	59	84	115
10	4	188	228	276	212	260	307	167	207	244
10	5	441	492	552	488	550	610	403	448	495
10	6	889	972	1046	978	1071	1171	792	876	952
16	2	36	50	73	35	69	101	30	48	70
16	3	128	159	194	149	189	232	119	148	177
16	4	498	588	662	643	735	823	497	576	655
16	5	1214	1309	1403	1472	1568	1690	1180	1267	1361
16	6	2625	2770	2907	3057	3201	3374	2471	2592	2726
20	2	39	58	82	45	84	122	36	58	83
20	3	168	198	239	194	234	276	151	183	213
20	4	650	738	817	823	924	1014	654	726	808
20	5	1612	1736	1830	1883	1989	2125	1533	1614	1708
20	6	3572	3700	3854	3888	4044	4226	3168	3276	3419
24	2	52	72	101	56	102	138	47	72	96
24	3	290	355	431	401	492	572	305	360	428
24	4	816	917	1010	1041	1127	1223	815	894	976
24	5	1984	2104	2220	2277	2376	2506	1841	1939	2028
24	6	6427	6645	6848	8065	8360	8702	6335	6570	6866
28	2	58	88	113	77	123	158	59	88	115
28	3	340	408	493	458	567	667	334	414	490
28	4	1001	1092	1195	1234	1323	1420	971	1062	1136
28	5	2387	2512	2615	2659	2772	2904	2154	2256	2372
28	6	7837	8109	8335	9623	9900	10206	7587	7848	8103
30	2	64	92	122	82	132	168	64	92	120
30	3	386	462	541	530	623	717	366	462	549
30	4	1098	1197	1291	1326	1421	1522	1066	1140	1220
30	5	2604	2718	2852	2842	2970	3127	2310	2408	2517
30	6	8602	8848	9074	10404	10700	11003	8136	8480	8729
32	2	65	94	122	88	138	173	61	96	123
32	3	399	486	574	538	656	748	401	486	559
32	4	1172	1274	1368	1412	1520	1625	1136	1212	1306
32	5	2811	2912	3028	3029	3150	3280	2457	2552	2668

Table 6: A comparison of a linear and a nonlinear expander for $n = 15$ and $r = 4$

	BCH(4,2)	proposed
width i	8	7
literals	108	807
gates	27	329

5 Conclusions

The problem to design a nonlinear code for a test vectors expander, with the requirement of all r-tuples possible in the output, has been formulated as a clique cover problem. The instances of the problem are large but have a high degree of symmetry, which seems to offer the possibility of analytical solution or better heuristic construction. To benefit from the degrees of freedom in the problem, assigning expander inputs to the produced vectors has been identified as an multi-valued (MV) variable encoding problem.

Experimental evaluation shows that good MV encoding is important for small r. For small instances up to $n = 6$ and $r = 2$, the sets of all minimum size covers were obtained using brute force. Instances up to $n = 32$ and $r = 6$ were solved heuristically, with the resulting expander widths i mostly equal to, and sometimes better than existing solutions. For the synthesis of the expanders, both the classical minimization-decomposition and resynthesis approaches can be used. The produced circuits were larger than corresponding linear expanders.

Acknowledgment

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures. The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics".

References

1. Agrawal, V.D., Kime, C.R., Saluja, K.K.: A tutorial on built-in self-test. 2. Applications. IEEE Design Test of Computers 10(2), 69–77 (1993)
2. Berkeley Logic Synthesis and Verification Group, *ABC: A System for Sequential Synthesis and Verification*. Software system. UC Berkeley, 2000. URL: `http://www.eecs.berkeley.edu/~alanmi/abc/`
. 3. Dutta, A., Touba, N.A.: Using Limited Dependence Sequential Expansion for Decompressing Test Vectors. In: 2006 IEEE International Test Conference, pp. 1–9 (2006)
4. Fišer, P., Schmidt, J.: A Difficult Example Or a Badly Represented One? In: Proc. of 10th International Workshop on Boolean Problems, pp. 115–122. Technische Universität Bergakademie, Freiberg, DE (2012)
5. Hellebrand, S., Rajski, J., Tarnick, S., Venkataraman, S., Courtois, B.: Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers. IEEE Transactions on Computers 44(2), 223–233 (1995)
6. Li, N., Dubrova, E.: Area-efficient high-coverage LBIST. Microprocessors and Microsystems 38(5), 368–374 (2014)
7. McGeoch, C.C.: *A Guide to Experimental Algorithmics*. Cambridge University Press (2012)
8. Mitra, S., Kim, K.S.: XPAND: an efficient test stimulus compression technique. IEEE Transactions on Computers 55(2), 163–173 (2006)

9. Novák, O.: Extended binary nonlinear codes and their application in testing and compression. In: 2017 22nd IEEE European Test Symposium (ETS), pp. 1–2 (2017)
10. Novák, O. "Pseudorandom, Weighted Random and Pseudoexhaustive Test Patterns Generated in Universal Cellular Automata". In: Lecture Notes in Computer Science 1667 – Dependable Computing – EDCC-3. Berlin Heidelberg New York: Springer Verlag, 1999, pp. 303–320
. 11. Novák, O., Rozkovec, M., Plíva, J.: Decompressors using nonlinear codes. Microprocessors and Microsystems 76, 103076 (2020)
12. Rajski, J., Tyszer, J., Kassab, M., Mukherjee, N.: Embedded deterministic test. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23(5), 776–792 (2004)
13. Rudell, R.L. *Multiple-Valued Logic Minimization for PLA Synthesis*. Tech. rep. M86/65. University of California in Berkeley, ERL, June 1986
. 14. Touba, N.A., McCluskey, E.J.: Bit-fixing in pseudorandom sequences for scan BIST. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20(4), 545–555 (2001)
15. Touba, N.A., McCluskey, E.J.: Synthesis of mapping logic for generating transformed pseudo-random patterns for BIST. In: Proceedings of 1995 IEEE International Test Conference (ITC), pp. 674–682 (1995)
16. Wunderlich, H.-J., Kiefer, G.: Bit-flipping BIST. In: Proceedings of International Conference on Computer Aided Design, pp. 337–343 (1996)
17. Yang, C., Ciesielski, M.: BDS: A BDD-Based Logic Optimization System. IEEE Transactions on Computer-Aided Design of Integrated Circuits And Systems 21(7), 866–876 (2002)