Application of Formal Methods (SAT/SMT) to the Design of Constrained Codes

1st Sunil Sudhakaran
Department of Electrical Engineering
Stanford University

2nd Clark Barrett
Department of Electrical Engineering
Stanford University

3rd Mark Horowitz
Department of Electrical Engineering
Stanford University

***Abstract*—Constrained coding plays a crucial role in high-speed communication links by restricting bit sequences to reduce the adverse effects imposed by the characteristics of the channel. This technique trades off some bit efficiency for higher transmission rates, thereby boosting overall data throughput. We show how the design of hardware-efficient translation logic to and from the restricted code space can be formulated as a Satisfiability Modulo Theories (SMT) problem. Using SMT, we can not only try to minimize the complexity of this logic and limit the effect of transmission errors on the final decoded output, but also significantly reduce development time—from weeks to just hours. Our initial results demonstrate the efficiency and effectiveness of this approach.**

***Index Terms*—constrained codes, high-speed links, wireline, RTL, SMT, SAT, signal integrity**

I. Introduction

Since the invention of the transistor, constrained codes have increased the reliability of digital data transmission [1]. Run-Length Limited (RLL) codes, such as 8b10b, have been pivotal in preventing signal degradation by limiting long sequences of identical bits, a technique widely adopted in storage devices and modern high-speed communication systems like PCIe (PCI-Express) and Ethernet links [2]. Beyond RLL codes, constrained codes play a significant role in memory interfaces. Memory standards like HBM, LPDDR, and DDR utilize Data-Bus Inversion (DBI) to reduce the number of lanes which simultaneously switch, effectively limiting power noise and jitter [3].

To increase data rates, links are moving from 2-level signals (PAM-2) to using 3 (PAM-3) or 4 levels (PAM-4). Maximum Transition Avoidance (MTA) coding is used in the PAM-4 signaling used in GDDR6x memory to mitigate inter-symbol interference and crosstalk by preventing transitions between the extremal signals [4]. As shown in Figure 1, MTA coding improves link margin by enlarging the eye diagram, a critical metric in channel performance evaluation. Finally, for PAM-3 codes used in Ethernet, GDDR7, and USB4-v2 one needs to convert binary data to ternary symbols to scale data rates [5], [6]. For all the codes mentioned, a key challenge in designing these codes lies in developing *efficient* schemes that minimize **digital logic complexity**, with a focus on reducing area and latency, key synthesis Quality of Results (QoR) metrics.

Beyond their primary role in managing bit sequences, constrained codes must work in tandem with error detection mechanisms like Cyclic Redundancy Check (CRC) to ensure reliability. The interaction between constrained codes and CRC is not trivial; the order in which these blocks are implemented have an impact on the system's ability to detect errors. Specifically, if the encode/decode precedes the CRC check, the decoder can amplify errors which occur while transmitting the coded data, complicating error detection. In this paper, we focus on these two critical aspects of constrained code design: mitigating the adverse effects on error detection and optimizing the total gate count and physical area. We demonstrate how these design challenges can be approached using Satisfiability Modulo Theories (SMT) formulations, a method that has also proven effective for similar design space exploration tasks [7]. We then discuss the strengths and limitations of this approach.

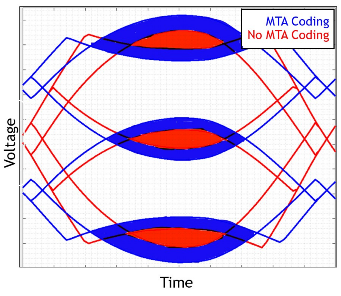


Fig. 1: Constrained Code Motivation: larger eye diagram area with GDDR6x PAM-4 Maximum Transition Avoidance Coding (blue) than without (red)

II. Background & Preliminaries

Let us formally define a constrained code as taking a sequence of n bits and generating a sequence of $n+k$ bits that obeys some sequence constraints. In order for the encoding to be possible, the number of $n+k$-bit sequences obeying these constraints (which we denote as ν) must be at least 2^n, ensuring that every possible input sequence has a valid output mapping within the subset of allowed codewords. As shown in Eq. 1, an encoder f maps a bit string of width n to the encoded output,and a decoder g performs the inverse mapping.

$$\begin{gathered} f : \{0,1\}^n \to \{0,1\}^{n+k} \\ g : \{0,1\}^{n+k} \to \{0,1\}^n \\ \forall\, x \in \{0,1\}^n,\ g(f(x)) = x \end{gathered} \tag{1}$$

A. Background on Constrained Code Design and Related Work

Constrained coding methodologies involve defining the specific constraints that an encoded sequence must satisfy, determining the parameters n and k, and defining the functions f and g. A *block* refers to a fixed-length segment of input or output symbols processed during encoding or decoding. There are two types of constrained codes. In state-dependent codes, the encoding of a block depends on both the current input and the encoder's state, while in memoryless codes, each block is processed independently.

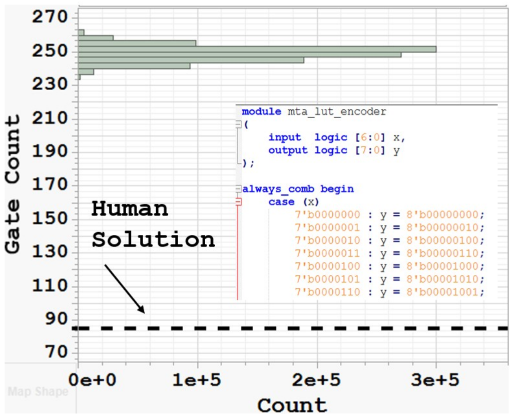
Memoryless Codes: The encoding of each block of symbols is independent of previous blocks. For example, mapping binary sequences into PAM-3 symbols, $(-1, 0, 1)$, uses a memoryless code. Each PAM-3 symbol needs 2 bits to represent the three possible values, so n bits of binary data are converted into s ternary symbols ($2s = n + k$). This process can also be reversed: when dealing with ternary weights or numbers, s ternary symbols can be compressed into n bits. Throughout this paper, we will refer to specific instances of these codes, including 3b2s (where $n = 3, k = 1, s = 2$) and 3s5b (where 3 ternary symbols are compressed to 5 bits).

State-Dependent Codes: These codes can be modeled by a finite-state machine (FSM), where the output at any given time depends not only on the current input but also on the state, which reflects some compressed information about previous inputs. Several of the codes mentioned in Section I are state-dependent, including MTA, RLL codes, and DBI. In the 1980's, Adler et al. developed a methodology to tackle the design of state-dependent codes using a "sliding block algorithm," where a sliding window of input symbols determines the output sequence [8]. When the number of states is small, two in MTA and 8b10b, a simpler approach can be used [4], dynamically selecting between two different code books.

To illustrate the challenge in finding hardware-efficient schemes, consider an example using PAM-4 with MTA, where $n = 7, k = 1, s = 4$ (4 PAM-4 symbols). As discussed in the introduction, MTA coding eliminates transitions like $-3 \leftrightarrow 3$ to reduce inter-symbol interference, improving the eye diagram and overall channel reliability. This constraint reduces the number of valid codewords from all possible 4 PAM-4 sybmols ($4^4 = 256$) to $\nu = 139$ [4], [9]. A very naive way to define f and g would be to randomly select any of the $\binom{\nu}{2^n} \cdot (2^n!)$ possible encodings (select a subset of size 2^n, then pick one of $2^n!$ possible functions) and implement it using a lookup table (LUT) in a hardware description language. Fig. 2 shows why such a naive approach is undesirable. It shows a histogram of the number of gates required to synthesize one million randomized LUT implementations compared to the number of gates in the best human-engineered solution. As the figure shows, there is a significant gap between them. Finding encodings with low gate counts is one of the challenges we address in this paper.

B. Error Detection and Block Ordering

Another challenge with constrained codes is their interaction with link error detection mechanisms like CRC. CRC



```
module mta_lut_encoder
(
    input  logic [6:0] x,
    output logic [7:0] y
);

always_comb begin
    case (x)
        7'b0000000 : y = 8'b00000000;
        7'b0000001 : y = 8'b00000010;
        7'b0000010 : y = 8'b00000100;
        7'b0000011 : y = 8'b00000110;
        7'b0000100 : y = 8'b00001000;
        7'b0000101 : y = 8'b00001010;
        7'b0000110 : y = 8'b00001001;
```

Fig. 2: Gate Count Histogram for 1M Randomized LUT Mappings for MTA Encoder

schemes operate by performing calculations on received data and comparing the result with the transmitted CRC data, which is computed before transmission [10]. The effectiveness of the error detection depends on several factors including the number of bits covered by the detection and the *CRC polynomial*, which is a parameter used in the calculations. The choice of CRC polynomial determines the minimum number of bit errors that must occur to pass through undetected by the CRC calculation, also known as the hamming distance (HD) [11].

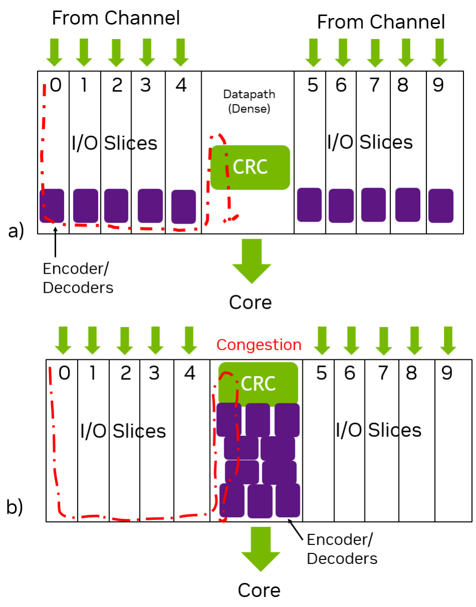


Fig. 3: Floorplanning Tradeoffs for Placement of CRC and Coding Block (a) CRC Applied After Decoder (b) CRC Applied Before Decoder

CRC blocks typically operate on a packet of data combining several individual I/O channels. The channel coding scheme however, usually operates on each channel separately. It is advantageous from a floor-planning perspective to place the CRC block after the channel coding blocks, as shown in Fig. 3(a). The encoder/decoder blocks are represented by the purple boxes and are located in each I/O slice. After decoding, the data is sent to the center block where it is combined with all other I/O data to perform CRC. Unfortunately, performing CRC after decoding can lead to error magnification. To avoid this, the encoder/decoder blocks can be placed after the CRC block as shown in Figure 3(b). However, this arrangement creates congestion and is likely to require more area.

To see how the floorplan in Figure 3(a) can magnify errors, consider Figure 4(b), where a single-bit channel error becomes a triple-bit error post-MTA decoding for an $n = 7, k = 1$ MTA code. In contrast, applying CRC before decoding, as shown in Figure 4(a), avoids this error expansion. Given the preference for the Figure 3(a) setup, we explored whether specific functions f and g could prevent or reduce error growth, aiming to ensure that any residual error growth remains within acceptable limits for the system's error tolerance and reliability requirements.

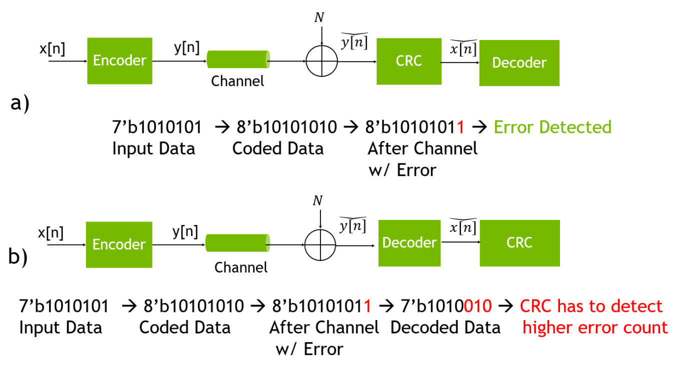


Fig. 4: Ordering of Encoding and CRC Blocks (a) CRC before the decoder (b) CRC after the decoder

To assess specific coding schemes, we compare the Hamming distances between pairs of codewords (set Y) and their mapped input pairs (set X) to determine if decoding increases the Hamming distance. An illustrative example is shown in Fig. 5. Inputs for the PAM-3 3b2s code are represented as nodes on the left, and the codewords as nodes on the right. Edges are color-coded by Hamming distance (1 = blue, 2 = red, 3 = green, 4 = magenta). Two assignments in function f are highlighted with dashed black lines: node $3'b100$ maps to $4'b0100$, and $3'b011$ maps to $4'b0101$. If $4'b0100$ is received instead of $4'b0101$, the Hamming distance in Y is 1, while in X, it is 3, meaning that decoding leads to error growth in this case. Enumeration of the 3b2s solution space is tractable, and we find that 72 out of the $\binom{\nu}{2^n} \cdot 2^n! = \binom{9}{8} * 8! = 362,880$ possible encodings result in no error growth. For larger codes, however, enumeration becomes intractable. We thus turn to formal methods.

III. Methods

Summarizing the above discussion, the key constraints for our encoder and decoder functions f and g are:

1) Injectivity
2) Codeword Restrictions
3) Error Growth Limits (Specific CRC/Coding Order)
4) Low Logic Complexity

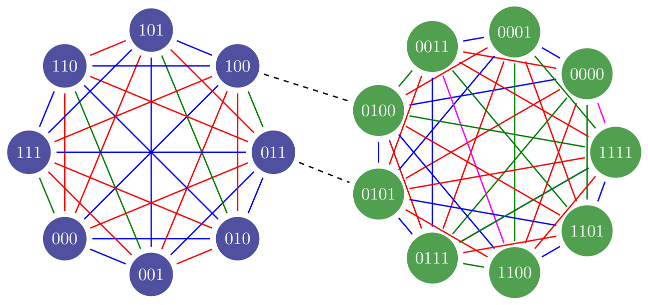


Fig. 5: Encoder/Decoder Mapping and Hamming Distances for PAM-3 3b2s Code. Edges represent hamming distance: 1 = blue, 2 = red, 3 = green, 4 = magenta.

Given that we are searching for a mapping from input data words to codewords, both of which are binary vectors, we can encode these constraints using the SMT theories of arrays and bit-vectors [12]. For a given n and k, we use an array indexed by a bit-vector of size n with values that are bit-vectors of size $n + k$ to represent the mapping, where logical assertions enforce the constraints, and SAT/UNSAT outcomes determine the existence of valid solutions.

Injectivity can be enforced by asserting that the values in the array are pairwise distinct, which requires $\binom{2^n}{2}$ assertions. Any code restrictions can be represented by simply asserting that each array value satisfies the required properties. For the PAM-3 case, for example, in which $\{00, 01, 11\}$ are used to encode the 3 values, we need $\frac{n+k}{2}$ disequalities per array value to ensure that "10" is never used to encode a value.

The next two sections describe how we handle constraints for error growth and low logic complexity, respectively. While our approach could address both constraints concurrently, we treated them independently in this study to assess the feasibility of achieving limited or no error growth, which has significant implications for top-level floorplanning, regardless of individual encoder or decoder complexity, as shown in Fig. 3.

We also include in these sections the results of several experimental evaluations. We use two SMT solvers in our evaluations: Bitwuzla [13] and Z3 [14]. All experiments were run on Intel CPU (Xeon) cores running Linux (RHEL 7.x).

IV. Error Growth

Section II-B highlights the connection between error growth and hamming distance when performing CRC detection on decoded data. We now consider the general case with input set X and output set Y. We represent the Hamming distances between pairs of words in X by $H_{X_{ij}}$, and between their corresponding outputs in Y after encoding and decoding, by $H_{Y_{ij}}$. Table I shows the constraints we must enforce to ensure either no error growth or at most one bit of error growth. Note that the constraint for the latter incorporates an optimization: ensuring that the hamming distance is always at least 1 forces the codewords to be distinct, thus *removing* the need for separate injectivity constraints.

As discussed earlier, while no error growth is ideal, we also consider at-most-one-bit growth, which may be acceptable de-

TABLE I: Error Aliasing Constraints

Constraint	***CRC Detection Considerations***
$\bigwedge_{i,j} H_{Y_{ij}} \geq H_{X_{ij}}$	No error growth
$\bigwedge_{i,j} H_{Y_{ij}} \geq \max(H_{X_{ij}} - 1, 1)$	At most one bit error growth

pending on the CRC polynomial's detection capabilities. In our SMT implementation, we assert the corresponding constraint from Table I for each pair (i, j).

Listing 1 shows a small portion of our SMT implementation using Bitwuzla's Python bindings. We iterate over all the elements in X and examine all pairs of inputs in X and their corresponding mapped values in Y. Since the codes are defined as bit-vectors, we iterate over the bits to get the total hamming distance for the pairwise combinations in Y.

Listing 1: No Error Growth Bitwuzla Implementation

```python
for lcv in range(X):  # Iterating over all locations
    # only check lower triangular (HD symmetric)
    for j in range(lcv):
        hd_bv = bz.mk_term(bz.Kind.BV_XOR, [code[j], code[lcv]])

        # counter to hold number of set bits, intitialize to zero
        code_hd = b.mk_bv_value(bz.mk_bv_sort(n_k), "0", 10)

        for i in range(n_k):
            # Extract each bit and extend
            b_i = b.mk_term(b.Kind.BV_EXTRACT, [hd_bv], [i, i])
            b_i_e = b.mk_term(b.Kind.BV_ZERO_EXTEND, [b_i], [n_k-1])

            # Add position i to coutner (if XOR set)
            code_hd = bz.mk_term(bz.Kind.BV_ADD, [code_hd, b_i_e])
        # hdx holds the target
        bv = bz.mk_bv_value(bz.mk_bv_sort(n_k), str(hdx), 10)
        # Assert counter is greater than or equal to target
        cset = bz.mk_term(bz.Kind.BV_UGE, [code_hd, bv])
        bitwuzla_solver.assert_formula(cset)
```

We explored a few PAM-3 codes and the MTA code to evaluate our error aliasing SMT formulations. The results are summarized in Table II. For the simpler PAM-3 (3b2s) code, Bitwuzla finds a solution in 2.3 seconds. For the larger 11b7s code ($n = 11, k = 3$), Bitwuzla returns unsatisfiable (i.e., no solution) for the no-error-growth formulation after 4 days, but the single-error case times out after 7 days. Bitwuzla also failed to converge on the no-error-growth case for an MTA 7-8b code. However, by dividing the problem into 139 sub-problems (by seeding each sub-problem with a fixed encoding for one of the inputs), the no-error-growth case was shown to be impossible after 41min. For the single-bit case, we tried an alternative encoding using Z3. We model the encoding function as a 2D Boolean array and use pseudo-Boolean assertions for the constraints. This formulation returned a satisfying solution in 700 seconds. The discovered solution ensures that a single-bit channel error transforms into at most a double-bit error, and a double-bit error into at most a triple-bit error.

TABLE II: Error Aliasing Results

Case	$H_{X_{ij}} \leq H_{Y_{ij}}$	$H_{Y_{ij}} \geq \max(1, H_{X_{ij}} - 1)$
PAM-3 $n = 3, k = 1$	SAT (2.3sec)	Not Applicable
PAM-3 $n = 11, k = 3$	UNSAT (4 days)	Timeout
MTA $n = 7, k = 1$	UNSAT (41min)	SAT (700 sec)

V. Minimal Logic Investigations

The most difficult task is trying to minimize hardware complexity. As before, we build on the SMT encoding described in Section III: assume a is an array of size 2^n containing bit-vectors of size $n + k$, with corresponding injectivity and codeword constraints.

A. All in One, Two-Level Sum of Products Formulation

Our initial approach uses a two-level Sum-of-Products (SoP) formulation to define a parameterized circuit structure representing the encoder and decoder. In this framework, constraints are instantiated once for each of the 2^n array entries to ensure correctness across all inputs. We solve for the parameters for both the encoder and decoder at the same time. Solving for both together offers advantages over sequential design, as an optimal structure for one of the two can require an inefficient design for the other. Also, by including both, the injectivity constraints can be removed, since the encoder and decoder logic constraints together imply injectivity.

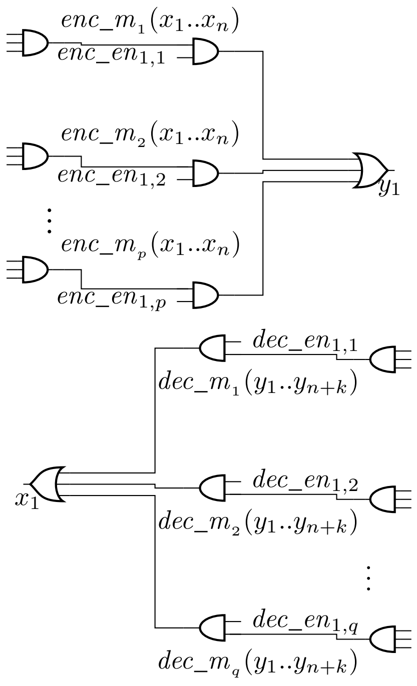


Fig. 6: Joint encoder/decoder circuits in SoP formulation (single slice). There are p, q logic gates allowed for the encoder and decoder respectively.

We describe the constraints for a single array entry. Let $\mathbf{x} = x_1, \ldots, x_n$ represent the bit-vector for the array index (input to the encoder), and $\mathbf{y} = y_1, \ldots, y_{n+k}$ represent the bit-vector for $a[\mathbf{x}]$ (the corresponding codeword). Fig. 6 illustrates a one-bit slice of the required hardware, generating the encoder output bit y_1 and the decoder output bit x_1.

The structure consists of two sum-of-products circuits, one with p encoder minterms and the other with q decoder minterms, both implemented as AND gates over selected bits of $\mathbf{x}$ (for the encoder) or $\mathbf{y}$ (for the decoder). These minterms are consistent across bit slices, with enable signals (enc_en and dec_en) determining whether a minterm contributes to the final OR gate for each output pin. Each bit slice follows this structure, differing only in the enable signals, resulting in $n \cdot p$ encoder enable signals and $(n + k) \cdot q$ decoder enable signals.

Besides finding the enable signal values, the SMT solver must also make good minterm selections. For the encoder, each minterm is defined by two n-bit vectors: enc_dc_i encodes "don't care" bits (0 if x_k is in the minterm, 1 otherwise), and enc_neg_i encodes whether to negate a bit (0 to negate, 1 otherwise). Similarly, dec_dc_i and dec_neg_i define the decoder minterms. For example, with $n = 7$ and $k = 1$, setting the first encoder minterm to $x_4\overline{x_2 x_1}$ corresponds to $enc\_dc_1 = 1110100$ and $enc\_neg_1 = 1111100$ (in little-endian).

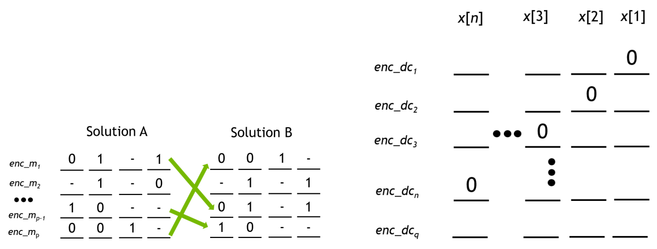
Listing 2 shows a small portion of our SMT implementation using Bitwuzla with Python bindings. We iterate over all the elements in X and add constraints on each pin as described above and illustrated in Fig. 6.

Listing 2: SMT code snippet for all-in-one formulation.

```
#Iterating over all locations
for lcv in range(X):
    for z in range(n_k):  # iterate over pins
        # initialize to constant
        minterm_sop = zero_val
        for j in range(ENC_GATES):  # iterate over gates
            # now perform AND with enable at position GATE
            enable_j = bz.mk_term(bz.Kind.BV_EXTRACT, [enc_gate_enable[z]],[j,j])
            minterm_res = bz.mk_term(bz.Kind.BV_AND,[enc_minterms[j], enable_j])
            # now perform OR operation with minterm_sop
            minterm_sop = bz.mk_term(bz.Kind.BV_OR, [minterm_res, minterm_sop])
        # for current codeword, set equal to decoder logic circuit output
        encode_equality_term = bz.mk_term(bz.Kind.EQUAL, [bits[z], minterm_sop])
        bitwuzla_solver.assert_formula(encode_equality_term)
```

B. Symmetry Breaking Strategies

In this formulation, observe that there are many symmetric solutions. For example, any permutation of satisfying minterms is also a solution, as shown in Fig. 7a. In our experiments, we use two symmetry-breaking strategies. First, we enforce that a bit-vector consisting of the concatenation of the minterms and don't-care bitvectors is monotonically increasing with the minterm index. Second, we utilize our domain knowledge that each pin must be used at least once for the encoder and decoder. This means the don't care bit-vector has to be zero (bit is used) for each location at least once across gates. We show this in Fig. 7b.



(a) Symmetric Solutions (b) Don't Care BV

Fig. 7: Symmetry Breaking Techniques: (a) Symmetric Solutions, (b) Symmetry Breaking Strategy with Don't Care BV

C. All in One Formulation Results

We applied Bitwuzla using our SMT formulations with symmetry breaking to several PAM-3 and MTA codes. For these experiments, we set $p = q$ (i.e., same maximum number of minterms for both the encoder and the decoder) as a starting point to simplify the initial exploration of the design space. For the MTA $n = 5, k = 1$ case, we successfully solved the encoder and decoder in a single formulation, obtaining one solution with $p = 16$ and another with $p = 14$. An attempt to find a solution with $p = 12$ fails: the query is unsatisfiable. Table III summarizes these results (the column labeled "Gates" indicates the value used for p), showing non-monotonic behavior in solver times: the 16-gate case took longer than the 14-gate case, but the unsatisfiable 12-gate case took the longest. This suggests that, while reducing the number of gates can decrease complexity (as seen with the 14-gate case), the need to explore the entire search space when a query is unsatisfiable can require a significant amount of solving time.

TABLE III: Selected Bitwuzla Results Using All-in-One Formulation (*Encoder Only for* $n = 7, k = 1$)

Code	**Case**	**Gates**	**CNF Clauses**	**CNF Vars**	**Solver Time (h)**	**Result**
MTA	$n=5, k=1$	16	17499	52390	5.67	SAT
MTA	$n=5, k=1$	14	14117	42274	0.67	SAT
MTA	$n=5, k=1$	12	10735	32158	5.88	UNSAT
PAM3	$n=5, k=1$	10	12880	38416	0.015	SAT
PAM3	$n=5, k=1$	9	11611	34627	15.96	SAT
MTA	$n=7, k=1$	56	1103263	365807	70.9	SAT
MTA	$n=7, k=1$	48	1047135	346927	109.3	SAT

For the PAM-3 (3s5b) $n = 5, k = 1$ case, we observed a more straightforward trend: reducing the gate count from 10 to 9 drastically increased the solver time despite the reduction in clauses and variables. This suggests that, even if a query is satisfiable, if the number of solutions is limited, the search can still be quite difficult. When scaling up to the $n = 7, k = 1$ code for MTA, solving both the encoder and decoder together was infeasible (timeout after 1 week), but we were able to obtain a solution for the encoder alone, albeit with a substantial increase in CNF clause count, variables, and solver time. Again, when reducing the gate count from $p = 56$ to $p = 48$, we observed an increase in solving time.

The effect of symmetry-breaking for the PAM-3 $n = 5, k = 1$ (3s5b) case is shown in Table IV, where we see a 30x speedup, resulting in a solving time of only 53 seconds (0.015 hours as shown in Table III).

TABLE IV: Symmetry Breaking Results

Dont Care Symmetry	**Minterm Ordering**	**Solver Time (s)**
No	No	1603
Yes	No	171
Yes	Yes	53

D. Iterative Solving and Multi-Level SoP Formulations

Given the challenges we faced scaling the problem to higher n, k values due to combinatorial explosion, we adopted a divide-and-conquer approach. Instead of solving the entire encoder-decoder problem simultaneously, we tackled it incrementally, focusing on a few pins at a time. Initially, we experimented with solving the encoder in isolation by moving away from the restrictive SoP formulation to a more flexible LUT-based structure. In the new formulation, a bit-vector bv_1 is first used to select a set of input bits. The selected input bits are then fed into a LUT block. For example, the MTA $n = 7, k = 1$ case is shown in Fig. 8. Here bv_1 selects a set of up to 3 variables (there are 63 possibilities, so bv_1 has 6 bits, since $2^6 = 64 > 63$). These are then fed into a "LUT-3" block. While a full LUT-3 has 8 bits for all truth table entries, we limited complexity by using a reduced set of functions, with bv_2 selecting from $2^4 = 16$ possible logic functions (Fig. 8).

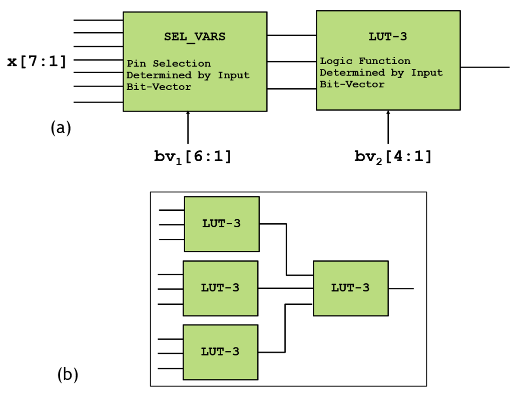

x[7:1]
SEL_VARS
Pin Selection Determined by Input Bit-Vector
LUT-3
Logic Function Determined by Input Bit-Vector
(a)
bv1[6:1]
bv2[4:1]
LUT-3
LUT-3
LUT-3
LUT-3
(b)

Fig. 8: Circuit Structure Details for Iterative Approach (a) Block Diagram for Base LUT-3 Cell (b) Example Circuit Structure for Intermediate Complexity

Our iterative algorithm, detailed in Algorithm 1, starts with a minimal circuit (e.g., a single LUT). If no solution is found, the complexity is incrementally increased until a satisfiable solution is achieved. After each step, solved pin values are added as hard constraints to the SMT solver. One potential drawback is that complexity may accumulate for pins solved later in the process, as illustrated by nested LUTs in Fig. 8(b). For this iterative approach, we utilized Z3's *Tseitin tactic*, which converts the problem into an equivalent Boolean satisfiability (SAT) problem in conjunctive normal form (CNF) [15], [16]. We then ran different SAT solvers in parallel. The flowchart, along with scripts and selected code, is available at https://github.com/sunil10110/formal.

When designing both the encoder and decoder simultaneously, careful consideration is required to preserve any assumptions about the code assignments y. We modified our approach to use a "ping-pong" technique, gradually introducing decoder output pins (x) while solving. This iterative method, which incrementally increases logic complexity to achieve a satisfiable solution for both the encoder and decoder, is detailed in Algorithm 1. Note that when solving for the encoder alone, the algorithm simplifies by removing the nested loop structure.

Table V summarizes the synthesis results for the MTA 7-8b encoder and decoder. Our SMT-based approach, which reduces the design time from weeks to days, offers a clear and structured methodology compared to the human-designed solution. Taking into account the total gate count and area for the encoder and decoder, the SMT joint design remains competitive: 94 gates and an area of 3.87 mm^2, compared to 85 gates and 3.27 mm^2 for the deployed solution. Furthermore, our SMT solution has fewer levels of logic (lower latency) compared to the deployed solution. The overall performance, paired with the substantial reduction in design time and a repeatable methodology, highlights the effectiveness of our approach.

Algorithm 1 Iterative Approach: Encoder/Decoder Co-Design

```
while unsolved x,y pins do
    Choose $y[i], y[i+1]$ from unsolved code bits
    Add constraints for circuit structure on $y[i+1:i]$
    Import solved code bits
    Solve $y[i+1:i]$
    while UNSAT y do
        Increase circuit complexity
        Solve $y[i+1:i]$ with updated circuit structure
    end while
    Import $y[i+1:i]$ solved bits
    while UNSAT x do
        for $j$ in unsolved $x$ pins do
            Add constraints for circuit structure on $x[j]$
            Solve $x[j]$ pin with updated circuit structure
            if $x[j]$ pin solved then
                break
            end if
        end for
        Increase circuit complexity for decoder on x
    end while
end while
```

TABLE V: Synthesis Results for MTA 7-8b Encoder and Decoder

Case	Levels of Logic	Gate Count	Area (mm^2)
Encoder			
Random Mapping	12	118	4.82
Deployed Solution	9	46	1.76
SMT Encoder Only	6	37	1.45
SMT Joint	7	40	1.65
Decoder			
Random Mapping	12	156	6.47
Deployed Solution	9	39	1.51
SMT Joint	8	54	2.22

VI. Conclusion

We addressed two key challenges in high-speed link encoder and decoder design: error amplification in constrained coding with error detection (CRC) and digital logic complexity. By formulating the design problem as an SMT problem, we identified codes that meet specific error growth requirements. We also tackled the complexity of encoder/decoder designs, optimizing gate count and area through SMT formulations. While these approaches worked well for the smaller codes, scalability became a challenge for larger codes. To address this, we developed iterative algorithms that approximate human-designed solutions, but with much shorter development times. For even larger codes, our methods face scalability challenges, making this a promising area for future research.

References

[1] Kees A. Schouhamer Immink. Innovation in constrained codes. *IEEE Communications Magazine*, 60(10):20–24, 2022.

[2] A.x Widmer and P. A. Franaszek. A DC-Balanced, Partioned-Block, 8B/10B Transmission Code. *IBM Journal*, 1983.

[3] Timothy M. Hollis. Data bus inversion in high-speed memory applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(4):300–304, 2009.

[4] Sunil Sudhakaran, Tim Hollis, et al. From Simulation to Production: An In-Depth Look at Designing Productizing GDDR6x, the World's First PAM-4 Memory Interface. *DesignCon*, 2021.

[5] Zhenyu Liu. Pam3 mapping for 802.3bp. Beijing, China, March 2014. IEEE P802.3bp RTPGE Task Force.

[6] USB Promoter Group. *USB-4 Version 2.0 with Errata and ECN through June 29, 2023*, 2023. All rights reserved.

[7] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. In *Proceedings of the 16th Monterey Conference on Foundations of Computer Software: Modeling, Development, and Verification of Adaptive Systems*, page 33–54, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Robert L. Adler, Don Coppersmith, and Michael Hassner. Sliding block codes and their rate in information theory. *IEEE Transactions on Information Theory*, 29(1):5–22, 1983.

[9] Mike O'Connor, Donghyuk Lee, Niladrish Chatterjee, Michael B. Sullivan, and Stephen W. Keckler. Saving pam4 bus energy with smores: Sparse multi-level opportunistic restricted encodings. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1001–1013, 2022.

[10] Aram Perez. Byte-wise crc calculations. *IEEE Micro*, 3(3):40–50, 1983.

[11] P. Koopman and T. Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *International Conference on Dependable Systems and Networks, 2004*, pages 145–154, 2004.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[13] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[15] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. Tseitin or not tseitin? the impact of cnf transformations on feature-model analyses. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*, page 13, Rochester, MI, USA, October 2022. ACM.

[16] Microsoft. Z3 Guide: Bitvectors. https://microsoft.github.io/z3guide/docs/theories/Bitvectors/. [Accessed: March 17, 2024].