Formal Methods in Computer-Aided Design 2022

# Error Correction Code Algorithm and Implementation Verification Using Symbolic Representations

Aarti Gupta
FVCTO[1]
Intel Corporation
Santa Clara, CA, USA
aarti.gupta@intel.com

Roope Kaivola
Core and Client Dev. Group
Intel Corporation
Portland, OR, USA
roope.k.kaivola@intel.com

Mihir Parang Mehta
FVCTO[1]
Intel Corporation
Santa Clara, CA, USA
mihir1.mehta@intel.com

Vaibhav Singh
FVCTO[1]
Intel Corporation
Portland, OR, USA
vaibhav.singh@intel.com

***Abstract*—Error-correction codes (ECCs) are becoming a de rigueur feature in modern memory subsystems, as it becomes increasingly important to safeguard data against random bit corruption. ECC architecture constantly evolves towards designs that leverage complex mathematics to minimize check-bits and maximize the number of data bits protected, as a result of which subtle bugs may be introduced into the design. These algorithms traverse a vast data space and are subject to corner case bugs which are hard to catch through constraint-based randomized testing. This necessitates formal verification of ECC designs to assure correctness of the algorithm and its hardware implementation. In this paper we present a technique of representing various ECC algorithm outputs as Boolean equations in the form of Boolean Decision Diagrams (BDDs) to facilitate reasoning about the algorithms. We also discuss the counting and generation of examples from the BDD representations and how it aids in tuning ECC algorithms for performance and security. Additionally, we display the use of Symbolic Trajectory Evaluation (STE) to prove the correctness of register transfer level (RTL) implementations of these algorithms. We discuss the scaling up of this verification methodology, using different complexity and convergence techniques. We apply these techniques to a number of complex ECC designs at Intel and showcase their efficacy on several categories of bugs.**

***Index Terms*—error correction codes, formal verification, symbolic simulation, binary decision diagrams**

## I. Introduction

With the ever-increasing capacity demands, memories are becoming denser and are more susceptible to soft errors. Error Correction Codes (ECCs) provide resiliency to the memory cell against errors due to cosmic rays, impurities during manufacturing, and other causes. Recent moves by chip manufacturers to extend ECC support to consumer processors, which was once limited to servers, emphasizes the universal necessity of ECCs. If the ECC fails, it will result in incorrect data getting read; in a safety-critical system, this can be catastrophic. ECC designs work by carefully adding data redundancy in the form of some check-bits to the data-stream while storing it. These check-bits and data-bits, which may have been corrupted during storage, are then used together to retrieve the original data. Though helpful in providing memory-protection, ECC designs are difficult to verify. ECC verification can be a challenge both for dynamic validation (DV) from the coverage perspective, and for formal verification (FV) from the convergence perspective. Consider the example of a Triple Error Correction Quadruple Error Detection (TECQED) design with 512 data-bits, 1-bit Metadata and 31 check-bits. Pre-silicon dynamic validation would require 4.87e163 input patterns to fully validate the design, a nearly impossible task, and post-silicon issues are discovered very late in the design cycle, not providing enough time to determine a robust fix. Owing to the complex equations generally used in ECC logic, these designs are not tractable by different industry standard FV tools. Most commercial model-checking tools are better suited to solve control path challenges and falter in achieving convergence on big datapath designs. Commercial datapath FV tools tend to rely on structural similarities of the reference specification and the implementation. Such similarities are absent in the case of closed-box ECC verification, where the specification is just a property stating, “the resultant data equals the received data”.

This paper shows our results in verifying diverse ECC algorithms and designs, across a range of datacenter and consumer processors, using an Intel-internal datapath tool, Forte/rSTE [3], [12]. The complexity of these verification tasks varied from a 64-bit corruption on a Dynamic Random-Access Memory (DRAM) device in a memory controller to a 512b-sized TECQED ECC in a data cache. We analyze our results with respect to different verification parameters (complexity, coverage, runtimes etc.) and compare with commercial tools.

In the remainder of this paper, we briefly introduce error correction (section II), and the underlying proof methodology with the Forte tool (section III). We explain the verification setup, and the properties we prove (section IV) on ECCs. We evaluate the results of these verification activities (section V) and sum up our contributions (section VII).

[1] Formal Verification Central Technical Office

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_21

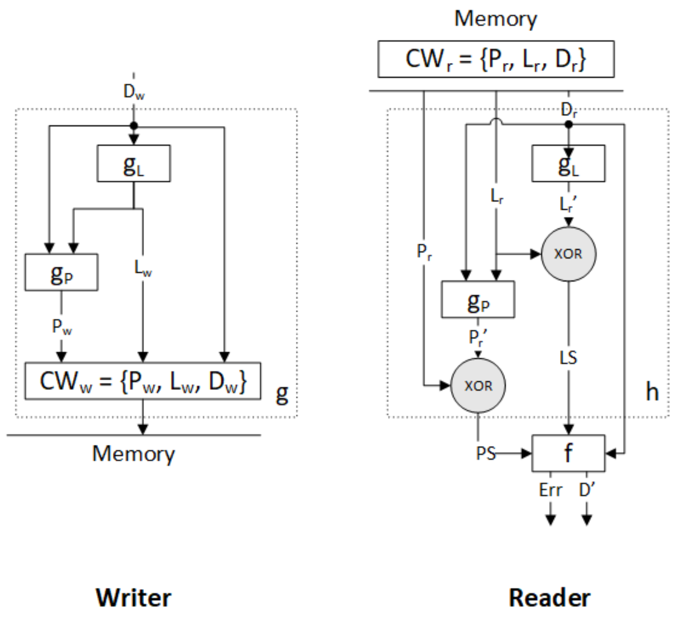


Fig. 1: ECC Writer and Reader

## II. Error Correction Codes

ECC functionality is usually implemented in hardware designs as two modules, a Writer and a Reader (Fig. 1). Using a generator function $g$, the Writer generates, from Data $D_w$, a codeword $CW_w$ which consists of $D_w$ appended with check-bits. There are two types of check-bits, locator bits $L_w$ and parity bits $P_w$. Once $CW_w$ is written to memory, it is subject to zero or more bits of corruption. Within the reader, the extractor function $h$ computes the locator syndrome $LS$ and parity syndrome $PS$. These syndromes are calculated by re-computing the locator bits $L_r'$ and the parity bits $P_r'$ and comparing them to their values $P_r$ and $L_r$ in the read codeword $CW_r$. Using these syndromes, the function $f$ can determine the presence of the error and determine the location of the error, finally returning corrected data $D'$ and error signal $Err$. $Err$ can be:

1) No Error (NE): No corruption detected in $CW_r$.
2) Correctable Error (CE): Corruption detected in $CW_r$ and fixed. Thus, output data $D' = D_w$.
3) Detectable but Uncorrectable Error (DUE): $CW_r$ corruption detected; but correction outside algorithm capabilities. Thus, $D' \neq D_w$.

Reliability, Availability, and Serviceability (RAS), feature sets that are associated with system resiliency in the presence of hardware faults, impose requirements that vary across designs. For example, some SRAM (Static Random-Access Memory) cache designs may need protection from bit-flips that can randomly happen at any bit-position in the cache-line, while other designs, such as DRAM, may need protection on groups of neighboring bits, which we will refer to as bit-groups. RAS requirements shape the choice of the ECC algorithm. These algorithms are based on the mathematical theories of Galois Extensions (Bose–Chaudhuri–Hocquenghem Codes) [4], [10], Lagrange Interpolation (Reed Solomon Codes) [15], and finite fields.

## III. Symbolic Simulation and Forte Toolset

Symbolic simulation extends standard digital circuit simulation with symbolic representations of values, covering behaviors of a circuit for all possible instantiations of the symbolic values in a single simulation. Used as a formal verification method, symbolic simulation is algorithmically simple and intuitive, which enables precise analysis and fine-grained mitigation of computational complexity, allowing the method to handle circuits that are above the capacity of standard formal model checking tools. Symbolic simulation excels in verification of deep targeted properties of fixed-length pipelines, in particular arithmetic and other datapath circuits. It has been the main vehicle for Intel arithmetic formal verification for over twenty years, and most arithmetic execution units of Intel processor designs have been exhaustively verified using it [3], [12]. It is the primary engine embedded in Intel's proprietary Forte/rSTE toolset. Symbolic simulation was first applied to ECC verification in 2005. Gradually, this application found its place in Server Memory Controller ECC (MC ECC) verification arsenal.

In a symbolic simulator the input stimulus may contain symbolic variables in addition to the concrete Boolean values 0, 1 and X. These symbolic variables are names of values, denoting sets of concrete values. The values of the internal signals computed in the simulation are then structural logical expressions on the symbolic variables on the inputs. For example, in a bit-level symbolic simulator, a single symbolic variable $a$ corresponds to the set of Boolean values consisting of both 0 and 1, and if stimulus to a symbolic simulation trace contains the variables $a$, $b$, and $c$, the internal signals might carry values like $a\&b$ or $a+(b\&c)$. The symbolic expressions in a simulation are commonly encoded using Binary Decision Diagrams (BDDs) [5].

The limits of computational capacity are the limits between what can and cannot be verified in practice. When attempting to resolve a capacity challenge, the crucial difference between symbolic simulation and other formal verification methods is that in symbolic simulation a capacity problem is extremely concrete. It manifests itself as a symbolic expression (BDD) that is too large, associated with a particular node and time in the simulation. This concreteness allows a user to analyze, understand and resolve the problem with a greater degree of precision than other methods of verification. This amenability to precise performance analysis is a key differentiator enabling the success of symbolic simulation. Direct user-level access to BDDs also allows advanced complexity management techniques, such as parametric substitutions and symbolic indexing, as well as automated analysis of the logical contents of a computation, for example, counting the precise number of input vectors satisfying or violating a given property.

In the Forte/rSTE toolset the base symbolic simulator STE is embedded in a code layer called relational STE (rSTE) in the context of a full-fledged functional programming language. Common computational complexity reduction techniques, including weakening, parametric substitution, etc., are made easily accessible to the user through programmable options to the tool. The framework also provides sophisticated debug support, breakpoints, waveform and circuit visualization, etc., to enable users to quickly focus on usual verification problems. The full programmability of the tool allows users to write reusable verification recipes that automate and structure shared or repeated tasks.

An important aspect of the verification toolset is that it provides a general symbolic computation capacity for Booleans. Not only can circuits be simulated with symbolic values, but any user-written program operating on Boolean data can be symbolically computed. This feature is very useful for multiple purposes: *ad hoc* programmatic analysis of failures, breaking symbolic computations into parts to analyze complexity issues, and early algorithm experiments prior to the existence of hardware implementations of those algorithms.

## IV. ECC Formal Verification

The verification setup for ECC FV involves connecting the Writer and the Reader, as shown in Fig. 3, abstracting out the storage component which usually sits in between these two blocks in real designs and replacing it with a corruption model. This model explicitly adds the effect of corruption on the codeword $CW_w$ generated by the Writer before it is fed to the Reader.

In the setup described in Fig. 3, there are two inputs $D_w$ and $C$. For symbolic analysis of the logic, we can assume these inputs to be symbolic variables instead of fixed stream of 0s and 1s, representing all values in the input space. Symbolic simulation then traverses the design, transforming input variables as BDDs in accordance with the design's logic, and finally makes the transformed BDDs available at outputs $D'$ and $Err'$. Correctness is then evaluated as a comparison between the output BDDs and the input BDDs under specific assumptions on the corruption.

For an ECC to guarantee correction of up to $n$ bits/bit-groups and detection of up to $n+1$ bits/bit-groups of corruption, the following must hold:

- Property 1: $(Countbits(C) = 0) \Rightarrow NE$ and $D' = D_w$
- Property 2: $0 < Countbits(C) <= n \Rightarrow CE$ and $D' = D_w$
- Property 3: $(Countbits(C) = n+1) \Rightarrow DUE$ and no guarantee on $D'$

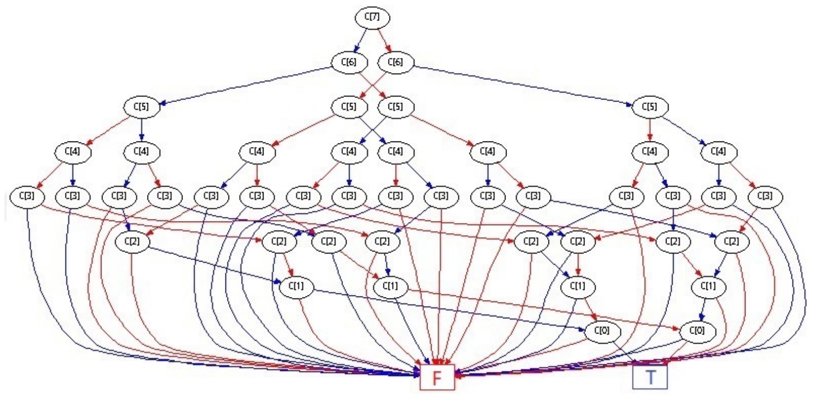
If the number of corrupted bits/bit-groups exceeds $n+1$, the algorithm makes no claims. For Single Error Correction Double Error Detection (SECDED), $n = 1$; for Double Error Correction Triple Error Detection (DECTED); $n = 2$ and for TECQED $n = 3$. DRAM ECCs employ custom algorithms at the level of devices, groups of bits of size 32 or 64, on a DIMM (dual inline memory module). The levels of protection provided by DRAM ECCs include full device protection, half device protection, and column protection.

In properties 1 to 3, it must be noted that the conditions NE, CE, and DUE are mutually exclusive and exhaustive. Different circuits implement this differently, but regardless it is necessary to prove mutual exclusiveness and exhaustivity. A circuit may encode 2 bits such that 00 is NE, 01 CE and 10 is DUE. In such a case we will need to show that 11 can not be computed. In other cases, each type of error is indicated by a separate signal, in which case we will need to show that these signals are mutexed. Usually, though, circuits indicate whether data was corrected, or not, with just one signal. If this signal is 1, then it is DUE; if 0, it is CE or NE. We will need to show that none of these three conditions overlap.

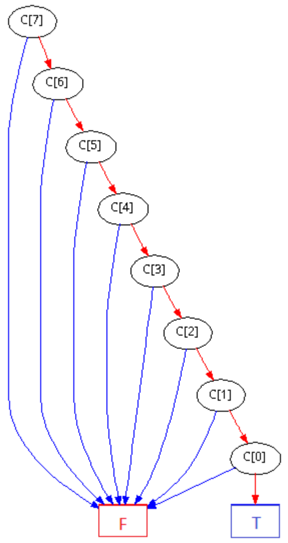
### A. ECC Implementation Verification

Using the symbolic simulator of Forte/rSTE toolset, the correctness of ECC designs can be ascertained without any reference to algorithms or design internals. This gives this technique a clear edge over other datapath FV tools which usually depend on a high-level model (HLM) against which an equivalence check is performed. Such HLMs are themselves prone to error and may incorporate an error which is also present in the design, in which circumstance a full equivalence check will nonetheless mask the bug. Moreover, such HLMs may need frequent remodeling in tandem with algorithm changes, which occur on a regular basis in the current landscape where ECC algorithms are continuously tuned in response to performance and security requirements.

To understand the nature of this verification process, let us take an example SECDED design protecting 4 bits of data (D[0]—D[3]) using 4 check-bits. The corruption vector (C[0]—C[7]) represents corruption that can happen at any bit position of the 8 bit codeword (data and check-bits). After symbolic computation of BDDs at each relevant node and times of interest, the BDD at the output port 'NE', which indicates absence of corruption on read data, may look like the BDD in Fig. 2 (a). Importantly, this BDD only makes reference to corruption bits, although the symbolic simulation accounts for fully symbolic data bits. This suggests that the symbolic condition for 'NE' depends only on corruption bits and is independent of the data bits. It can also be noted that in this BDD there are several paths that lead to the terminal node 'T', while the naive expectation would be for a single path to reach this terminal i.e., the no-corruption path. This is due to the fact that ECC algorithms are constructed to guarantee error correction and detection up to a maximum bound of corruption, while the corruption vector that we considered allows corruption on every bit of codeword i.e., up to 8 bits of corruption. Therefore, to verify the algorithm's properties, we must evaluate this BDD under the implication of the max-bound condition. Forte provides debug hooks that allows users to access the BDDs at different design nodes at various times, thus the 'NE' BDD can be extracted and evaluated for satisfiability using simple Forte commands when $Countbits(C) \leq 2$. Under this condition, property 1 is



(a)



(b)

Fig. 2: BDD for NE in Example

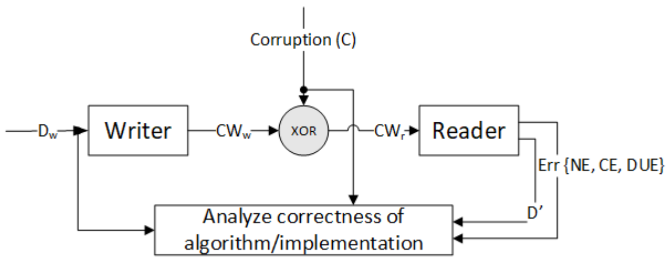


Fig. 3: ECC FV Setup

substantiated and the only satisfiable path is the one where C[0]-C[7] are all false. Symbolic analysis can be done in a similar fashion on other properties.

We saw that a simple 4 bit SECDED could result in a 44-node BDD in Fig. 2 (a) for an error signal in the circuit. Computing and storing BDDs of this kind is a likely limiting factor as design complexity increases. By means of various techniques described below, we could limit the BDD sizes to smaller bounds and scale this technique to designs where commercial datapath tools failed to converge.

*1) Parametric Substitution:* In many circumstances, symbolically simulating for a subset of data, i.e., data under a specified condition, is more efficient than symbolically simulating with unconstrained data. In such circumstances, parametric substitution [2] is very effective. A generic correctness statement of a design can be represented as:

$$P(x) \rightarrow Q(x)$$

Where $P$ is a constraint on the data space, $x$ is a vector of BDD variables, and $Q$ is a function that carries out symbolic simulation. Under parametric substitution, we use a function $param$ to compute a parametrized functional vector representation of $P$ and rewrite the correctness statement as:

$$Q(param(P(x)))$$

As an example, Fig. 2 (a) depicts the BDD for the No Error (NE) signal of the 4-bit SECDED design, when computed in a simulation with fully unconstrained values. This BDD captures the behavior of the design for any number of corruptions from zero to eight. However, the design is only expected to produce reasonable output when the number or corrupted bits is at most two, in other words when the condition $Countbits(C) \leq 2$ is true. We can compute a parametric substitution from this condition, and instead of simulating the system with fully unconstrained symbolic corruption bits, we can simulate it with small BDD's for the corruption bits, restricting the behavior only to the interesting cases. Conceptually, the parametric substitution produces BDD's for the corruption bits that allows the first two corruption bits to have any values, but any subsequent bits can only be high if at most one higher bit is already high. In the resulting simulation, the BDD for the No Error (NE) signal is as depicted in Fig. 2 (b), a considerable simplification when contrasted with the general case.

*2) Case-Splitting:* With case-splitting, we decompose the data space into a number of sets and separately verify the circuit for each set. This reduces the BDD complexity and search space for each case in a divide-and-conquer fashion. ECCs naturally lend themselves to a case-split on the number of bits of corruption that are allowed. For example, a SECDED design can be decomposed into 3 cases: no corruption, 1b corruption, and 2b corruption. Parametric substitution of the case constraint will lead to even smaller BDDs. In the case of the example illustrated in Fig. 2, it will lead to a zero-sized

BDD with only a terminal vertex “True” or “False” for the signal ‘NE’.

Further case-splitting can be done based on the locations of the (one or more) corruption bits. This has been essential in our verification of 512-bit TECQED designs, as it made convergence of the proof possible. In addition, case-splitting is useful towards reducing the runtimes of existing proofs by means of parallel processing.

*3) Symbolic Indexing:* Symbolic Indexing [1] is an efficient technique that can logarithmically scale down the number of variables a BDD is dependent on. Taking the example of 4-bit-SECDED, if we replace the 8-bit corruption vector (C[0]—C[7]) with two vectors (CI1[0]-CI1[2]) and (CI2[0]-CI2[2]), where the value CI1 gives the index of first bit that is corrupted and CI2 gives the index of second corrupted bit, then the same symbolic corruption information can be relayed to the simulator using 6 variables instead of original 8. Generally speaking, a symbolic corruption on an ECC design with codeword length $n$ and up to $k$ bits of corruption can be represented using $k \times log_2(n)$ variables using symbolic indexing, which would otherwise require $n$ variables. This state space reduction becomes all the more important as we move to larger designs such as 4096-bit-SECDED, where this technique allows use of two 13-bit corruption-index vectors instead of a 4110-bit corruption vector.

*4) Variable Ordering:* BDD size is very sensitive to its variable order [7]. Variable order of a BDD determines the order in which variables will appear for all its node-traversal paths. The optimal variable order is required to ease BDD computations on bigger circuits like memory controllers where one design may support multiple ECC schemes. In verification of such designs, it is advisable to put control variables before data variables. This is because the control variables may choose a completely different mode of operation in the circuit; and having them at the top of the BDD tree simplifies the branches by preventing a commingling of different ECC schemes. For example, variables on signals that select the ECC mode, or signals that are used for configuration settings such as error masking, should take precedence in ordering relative to variables for corruption and data.

*5) Dynamic Weakening:* Symbolic simulation on ECC designs may sometimes encounter a BDD blow-up. Forte assists in investigating and resolving such a bottleneck through dynamic weakening. The user can provide a maximum bound of BDD limit, and whenever BDD size at an internal node during the symbolic simulation exceeds the provided limit, tool automatically ‘weakens’ that node i.e., replaces that BDD with an ‘X’. This new value is then propagated through the circuit simulation. If the weakened node was irrelevant to the final output computation, then it saves unnecessary simulation on that path, else the X propagation reaches the output nodes. In these cases, the BDD representation at output node can be of form $BDD_A + X(BDD_B)$, where $BDD_A$ represents the variable-assignments that give concrete values 1 and 0 to the output and $BDD_B$ represents the variable assignments that can lead to X. Forte’s schematic viewer enables chasing this X and determining the cause of the divergence. The tool also facilitates substitution of variables with random example values. This makes sample cases more concrete and easier to debug.

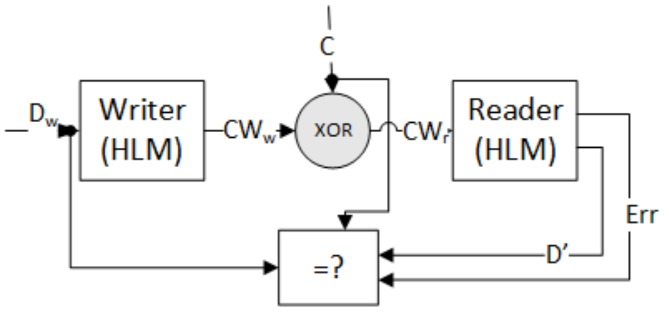


Fig. 4: Architectural ECC FV

## B. ECC Architectural Verification

Forte can also be used to check algorithm architecture, in addition to its use in closed-box verification of design properties. This mode, however, does need algorithm understanding and modeling the Writer and Reader parts of algorithms as HLMs, but the goal remains the same i.e., checking overall correctness of algorithm by means of property checking. This is done by using a verification setup similar to the design verification, only replacing the Writer and Reader design blocks, as shown in Fig. 4, with their HLMs written in Forte’s functional language *reFLect* [9]. Verification tasks of this nature, instead of using the symbolic simulation capability of Forte, use its symbolic computation feature. In a manner akin to abstract interpretation [6], the input variables are propagated through the logical functions present in the HLM, undergoing BDD transformations at each function. Finally, BDDs are derived at the output of the HLM, which can then be used for reasoning about the correction and detection properties of the ECC algorithm. This architectural verification is independent of the design, and in practice it is often carried out before the algorithm is implemented in RTL. This shortens the feedback loop of design and verification, thus reducing time to market for such designs.

## C. Counting and Enumerating Error Patterns

In modern server designs, some algorithms provide protection of a bit-group within specific published bounds. Design pressures to add metadata bits to the bit group lead to customizations which reduce the number of check-bits and result in such a lower bound being chosen over a guarantee of full correction. For example, a customization to include directory bits, poison bits, and tag bits (i.e., metadata) may lead to an algorithm which claims, “100% detection, and better than 99.999% correction.” This claim implies that fewer than

$0.001\%$ of all possible block corruption patterns can lead to a DUE. This performance-accuracy tradeoff makes verification of this claim complex. In contrast to the properties explained earlier in this section, which were of the nature "under the given conditions, BDDs on the outputs that indicate error must evaluate to True or False", our claim now involves exact counting of the paths that lead to the terminal vertices. Additionally, an algorithm may make a conditional claim such as "Errors that fall on both right and left half of device are outside scope of ECC and are not corrected but detected for ˜99.999% of error patterns." Such a claim, in general, relates several design-outputs under a specific corruption condition. This claim bounds the number of memory failures that can go undetected, also known as SDCs (Silent Data Corruptions). To verify this, we need to count all corruption variable assignments under which output BDD for DUE error signal evaluates to False, but output data $D'$ is not equal to write data $D_w$. Thus, the property to be checked becomes:

$$satCount(Cond \rightarrow \neg DUE \,\&\, (D' \neq D_w)) < x$$

Here, Cond is the corruption condition under which counting is performed, $x$ is an upper bound on the number of expected SDCs, and satCount is a count of the number of satisfying assignments to a given formula.

The corruption condition and the DUE/SDC conditions can be composed together to form a new BDD, and we can count the number of satisfying instances through procedures written in *reFLect*. We are also able to enumerate the corruption patterns that lead to SDC or DUE in addition to counting them. This data is sometimes needed by memory vendors and is also helpful during debugging to understand the frequency/location of failures.

One consideration while generating these counts is the avoidance of duplicates, which we illustrate for the example of a SECDED algorithm. To count the SDC cases for 3 bit corruptions, we define symbolic indices $p1$, $p2$ and $p3$. Once we compute the SDC condition, there could be cases that are counted multiple times, such as $p1 = 0$, $p2 = 1$, $p3 = 2$ and $p1 = 0$, $p2 = 2$, $p3 = 1$. However, by assuming without loss of generality that $p1 > p2 > p3$ in the condition in the above expression, the counting of duplicate cases is avoided.

## V. Results

We discuss the impact seen from this verification effort on ECC designs of varying complexity. In the past 2 years, we have verified 14 ECC designs and their corresponding algorithms, resulting in the discovery of 48 bugs overall and proving the absence of bugs in customer releases. These ECCs are the state of the art for commercial designs. They represent a full range of Intel designs and were not cherry-picked for the case study.

Quantitatively, Table I lays out the results of ECC FV spanning multiple projects and design generations. Table I compares ECC property checking using Forte against established industrial EDA (Electronics Design Automation) tools tuned for control-path and data-path FV. Since our BDD-based technique with Forte allows us to do a closed-box checking without reference to design internals, we explored the feasibility of similar testing with the EDA tools for a fair comparison. Tool #1 and Tool #2 in Table I can use the same verification setup as shown in Fig. 3 and allow the user to state the design properties by means of System Verilog Assertions (SVA). Both these tools use various engines that can run in parallel to achieve a concrete result and may give a bounded proof in case if they fail to converge. As seen from Table I, these tools are able to converge on small-sized designs based on simple ECC algorithms such as SECDED, but as the design size or algorithm complexity increases, convergence is not seen. Our techniques, however, achieve convergence in a matter of minutes in all of the designs under consideration. Tool #2 is more tuned towards datapath verification, but no difference was observed between Tool #1 and Tool #2 with respect to convergence on these tasks. Typically, datapath FV commercial tools do better on arithmetic designs than standard model checkers due to their word-level engines. However, the arithmetic in ECC algorithms is primarily bit-level and, as seen from our results, word-level processing was not particularly useful here.

The size of ECC designs ranged from 3K gates (smallest) to over a million gates (largest). However, more than the design size the proof convergence depended on arithmetic complexity of the algorithm itself. For example, algorithm offering bit protection were more amenable to FV proofs compared to algorithms doing bit-group level protection. Also, the complexity increased as the number of bits under protection umbrella grew. For instance, the number of case-splits required to achieve proof convergence were 17K for a 512 bit TECQED and only 300 for a DECTED design of the same data-width, while none of the SECDED designs verified needed a case-split. Within the same algorithm category, the complexity was directly proportional to the data-size. So, a 32 bit SECDED is much easier to verify compared to a 4096 bit SECDED.

Qualitatively, we consider it instructive to categorize the kinds of bugs we have found. This analysis is intended to help both design experts and verification experts identify common patterns that lead to design errors.

### A. Architectural Bugs

Architectural FV allows early bug investigation, even before the implementation of an algorithm in RTL. As a result, bugs found in this process are prevented from ever entering the RTL design. This is a worthwhile exercise since the algorithms themselves are complex enough, owing to the interplay between different architectural features, to give rise to corner case bugs. For example, our recent investigation of single block corruption in a new ECC scheme in a memory controller found exactly 3 failure cases out of $18 \times 2^{32}$. Previously, some of our FV investigations have found corner case bugs

| **Algorithm** | **Protection level** | **Data width in bits** | **Engineering effort in person-days** | **Property Convergence** | | |
|---|---|---|---|---|---|---|
| | | | | **Forte** | **EDA tool #1** | **EDA tool #2** |
| SECDED | Bit | 1-256 | $< 2$ | Yes | Yes | Yes |
| | Bit | 4096 | $< 2$ | Yes | No | No |
| DECTED | Bit | 256 | $< 4$ | Yes | No | No |
| | Bit | 512 | $< 4$ | Yes | No | No |
| TECQED | Bit | 512 | $< 15$ | Yes | No | No |
| Custom ECC schemes for DRAM device protection | Bit groups (16/32/64 bits) | 512 | Continuous engagement across design cycle | Yes | No | No |

TABLE I: Comparison of Property Checking with Different Formal Tools. EDA Tool #1 is a Model Checking Tool and EDA Tool #2 is a Commercial Datapath FV Tool

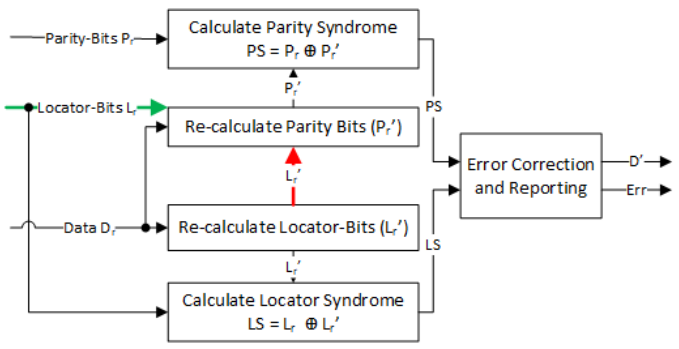

Parity-Bits P_r
Calculate Parity Syndrome
PS = P_r ⊕ P_r'
P_r'
Locator-Bits L_r
Re-calculate Parity Bits (P_r')
L_r'
Data D_r
Re-calculate Locator-Bits (L_r')
L_r'
Calculate Locator Syndrome
LS = L_r ⊕ L_r'
PS
LS
Error Correction and Reporting
D'
Err

Fig. 5: Implementation Error Example

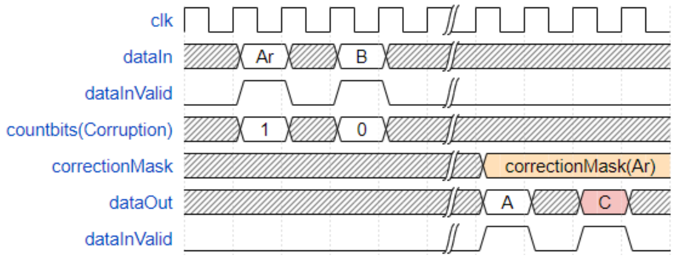


Fig. 6: Pipeline Bug Example

that escaped testing and subsequently led to the publication of customer errata [11].

## B. Implementation Errors

Even with a correct algorithm, an implementation can be erroneous, due to a variety of reasons such as specification ambiguity. We encountered one such bug while reading parity from memory; while the architecture specified a column major order read, the RTL implementation was row major. In another example, a simple misconnection led to a breakdown of ECC functionality. This case is illustrated in Fig. 5 which shows the functionality of a generic ECC Reader. The Reader reads the codeword from memory which is comprised of Data $D_r$ and check-bits (i.e., locator bits $L_r$ and Parity bits $P_r$). The Reader uses the read data $D_r$ and the Locator bits $L_r$ to re-calculate the new check-bits ($L'_r$ and $P'_r$). These recalculated values are then compared against the check-bits that were read from memory to compute syndromes that are then used to ascertain error presence and its correction. However, in the case presented in Fig. 5, instead of using original locator bit $L_r$, (green arrow indicated in Fig. 5) the recalculated version of $L'_r$ was used (red arrow in Fig. 5) to re-compute Parity bits. Due to this seemingly innocuous issue, 60% of 1b corruption cases that specification claimed to be correctable were marked uncorrectable in the design, and around 25% of 2b corruption cases led to fatal SDCs. The timely verification of these designs prevented these critical bugs from making their way into the final products.

## C. Pipeline Bugs

Frequently, bugs arise from pipelines where a signal was used at the wrong stage, or an incorrect clock-enable prevented the relevant values from propagating. One such failure is described in Fig. 6. Here 2 sets of data ($Ar$ and $B$) enter the Reader in succession, where $Ar$ has 1b corruption, and $B$ is not corrupted. The design was expected to correct the corrupted $Ar$ to its original value $A$ and to leave $B$ unchanged. However, $B$ was changed. It was found that an internal signal, correctionMask, used for fixing the corruption was not updated while processing $B$ due to an incorrect clock-enable, and its stale value resulted in a spurious correction. This behavior continued for a long time in the pipeline, until the next update of the clock enable signal. This shows that an algorithm, however carefully designed, can be rendered ineffective for a large number of corruption cases due to pipeline bugs. The fixing of this bug also shows the salutary effect of datapath FV on the surrounding control-path logic, as the closed-box verification approach focuses on the overall functioning of the design in addition to the correctness of the ECC algorithm.

## D. Specification Bugs

The RAS capabilities of an ECC design need to be clearly documented for customers in an External Design Specification document. Thus, these specifications need to be accurate and must reflect exact ECC capabilities that exist in the silicon product. Many of the complex algorithms may not provide 100% correction on a block, but nonetheless specify $x\%$ correction, $y\%$ detection, and $z\%$ silent data corruption. These data percentages are critical to memory vendors and need to be verified, but this verification is complex as it is not a simple true or false claim but involves exact counting of each category of results. Since the number of satisfying assignments can be counted using symbolic representations, it can be verified that both the ECC algorithms and their implementations deliver the claims that they make in the specification. We helped in fixing some of these results based on our calculations. In one such case, an anomaly was detected on the number of DUE counts where the actual counts offered by algorithm differed by the published claims by just 7.10e-13%.

### *E. Miscellaneous Bugs*

Since we analyze each ECC design in depth, we sometimes encounter issues such as efficiency bugs, where the design uses more check-bits than required by the algorithm, or parametrization bugs, where some design parameters are not passed-down correctly in the design.

## VI. Related Work

Model-checking based FV techniques have been used for verifying ECC designs. For example, a 128-bit TECQED ECC was formally verified in [13], and a 256-bit Double Error Correction Triple Error Detect (DECTED) ECC design was formally verified in [8] using a commercial model checker. Both these proofs converged only after a lot of design interventions and rewriting the design to make the logic fully combinational. These interventions need special handling, and one needs to make sure the bridges between these abstract models are verified, maintaining overall coherence. In contrast, our approach does not need any reduction or abstraction of designs. Scaling up these approaches [8], [13] to bigger ECC designs will be difficult as model-checking tools get fatigued due to the inherent complexity of ECC designs and the vast input space. In [13], extreme convergence steps were taken to conclude the proof on a 128-bit TECQED with a proof runtime that is counted in days, while with our technique we could verify a $4\times$ data-width design (512-bit TECQED) in just 2 hours.

Lvov et al. [14] verified Reed-Solomon codes by computing Grobner bases, using the SINGULAR arithmetic engine. Their proofs are independent of data width and their runtimes are dependent only on the number of bits corrupted. However, their assumption of the insufficiency of BDD-based techniques for ECC verification has not been borne out in Forte, which is capable of crunching through Boolean equations of the required size. This is accomplished through variable ordering and parametric substitution techniques, as discussed further in section III. As a result, ECC verification in Forte becomes a much simpler matter of declarative specification of the desired ECC properties, without reference to the underlying algebraic structure.

## VII. Conclusion

The results discussed in this paper show the efficacy of our BDD-based symbolic representation in verifying properties of ECC designs at both the algorithmic and RTL level, finding bugs which would have been infeasible to find through testing. These techniques are scalable to large ECCs by means of parametric substitution and other complexity management techniques. The success of these techniques in discovering bugs on industrial designs allows the categorization of the most common kinds of ECC bugs, which in turn shapes the practice of ECC design towards avoiding these bugs from the very beginning.

These techniques are valuable because they allow for a closed-box approach that requires neither knowledge of the design nor an HLM for equivalence checking. Additionally, these Forte techniques outperform other closed-box tools. Forte differentiates itself here by allowing algorithmic verification, even in advance of the RTL being written, and by helping provide bounds on the incidence of certain kinds of errors. By facilitating efficient correctness proofs and supporting the development and tuning of ECC designs on multiple fronts, Forte-based ECC verification techniques position themselves to be useful well into the future.

## Acknowledgment

Formal verification of Error Correction Codes in the paradigm discussed in the current paper has been practiced at Intel since 2005. We would like to express our gratitude to all of our former colleagues who have contributed to this effort either conceptually or through code. In particular, we would like to thank Scott Huddleston for his seminal work on the error probability counting methods discussed in section IV-C, and Levent Erkok, Flemming Andersen, John Matthews and John Erickson for advancing the methodology over a series of verification efforts on successive families of memory controllers. We also thank Jing Ling, Hsing-min Chen, Wei Wu, and Saurabh Kolambkar for architecture and design help on various ECC circuits. We thank Disha Puri for carrying out comparison experiments on commercial datapath formal verification tools. Finally, we would like to thank Gavriel Gavrielov and Achutha Kirankumar V. M. for the opportunity to carry out this work.

## References

[1] S. Adams, M. Bjork, T. Melham, and C. H. Seger, "Automatic abstraction in symbolic trajectory evaluation," Formal Methods in Computer Aided Design 2007.

[2] M. D. Aagaard, R. B. Jones, and C. H. Seger, "Formal verification using parametric representations of Boolean constraints," Proceedings of the 36th annual ACM/IEEE Design Automation Conference 1999.

[3] Achutha Kirankumar V. M., A. Gupta, and R. Ghughal, "Symbolic Trajectory Evaluation. The Primary Validation Vehicle for Next Gen Intel® Processor Graphics FPU," Formal Methods in Computer Aided Design 2012.

[4] R. C. Bose and D. K. Ray-Chaudhuri, "On A Class of Error Correcting Binary Group Codes," Information and Control 1960.

[5] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Transactions on Computers, 100.8 (1986): 677-691.

[6] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. ACM Press. pp. 238–252.

[7] D. Deharbe and J. Vidal, "Optimizing BDD-based verification analysing variable dependencies," In XIV Symposium on Integrated Circuits and System Design (SBCCI'01), pp. 64-69. Computer Society Press, 2001.

[8] K. Devarajegowda, V. Hiltl, T. Rabenalt, D. Stoffel, W. Kunz, and W. Ecker, "Formal Verification by The Book: Error Detection and Correction Codes," DVCon 2020.

[9] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving", Journal of Functional Programming, 16(2):157-196, March 2006.

[10] A. Hocquenghem, "Codes correcteurs d'erreurs," Chiffres 1959.

[11] Intel Corporation, "Third Gen Intel® Xeon® Scalable Processors Specification Update", May 2022, https://www.intel.com/content/www/us/en/design/resource-design-center.html, Document ID 637780, Erratum ID ICX 66.

[12] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber and A. Naik, "Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation," Computer Aided Verification 2009.

[13] A. Kumar and K. Devarajegowda, "Verifying ECCs Used in Safety Critical Designs with Formal," Jasper User Group 2021.

[14] A. Lvov, L. A. Lastras-Montano, V. Paruthi, R. Shadowen, and A. El-Zein, "Formal verification of error correcting circuits using computational algebraic geometry," Formal Methods in Computer Aided Design 2012.

[15] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," Journal of the Society for Industrial and Applied Mathematics 1960.