Formal Verification by The Book: Error Detection and Correction Codes

Conference Paper · March 2020

CITATIONS
2

READS
2,338

6 authors, including:

Keerthikumara Devarajegowda
Siemens EDA
32 PUBLICATIONS **171** CITATIONS
SEE PROFILE

Wolfgang Ecker
Infineon Technologies
175 PUBLICATIONS **1,109** CITATIONS
SEE PROFILE

Formal Verification by The Book: Error Detection and Correction Codes

Keerthikumara Devarajegowda*†, Valentin Hiltl*, Thomas Rabenalt*,
Dominik Stoffel†, Wolfgang Kunz†, Wolfgang Ecker*‡
*Infineon Technologies AG, Germany
Email: <Firstname>.<Lastname>@infineon.com
†Technische Universität Kaiserslautern, Germany
Email: stoffel@eit.uni-kl.de, kunz@eit.uni-kl.de
‡Technische Universität München, Germany
Email: wolfgang.ecker@tum.de

Abstract

We present an approach to exhaustively verify the correctness of Error Correction Code designs using formal methods. The proposed approach exploits the linearity of *Syndrome Generators* to prove that the detection and correction of bit errors depends only on the erroneous bit positions, and not on the original data vector. By proving the linearity property, the input analysis space for error detection and correction properties is significantly reduced by a factor of 2^n, where n is the width of data vector. As a result, the proof runtimes of the properties are also reduced. For example, the proof runtime of a 3 bit error detection/correction property requires less than 2 hours for passing on a 256 bit Error Correction Code design in a commercial formal verification tool. The approach has been successfully applied to multiple instances of Error Correction Code designs implemented across several safety-critical automotive system-on-chips. An in-house property generation tool has been employed to foster re-usability and verification productivity. Results show that the proof runtime of the properties leveraging the linearity feature decreases by a factor of 50, and scales proportionally with the width of the data vector and detection and correction capability of the codes.

I. Introduction

Safety-critical automotive applications have stringent requirements for functional safety and reliability. The devices used in safety-critical applications need to ensure that the basic functionalities of the device are unaffected, even during the deviation from normal working conditions (e.g., magnetic interference, radiation errors, temperature fluctuations, electrical transients, heat dissipation, etc.). To fulfill these requirements, devices used in safety-critical applications are equipped with additional hardware and software functions, and redundant memory elements. Error Detection and Correction Codes (ECCs) are one such safety mechanism that is widely implemented in memory and register file interfaces to safeguard memory elements against soft errors. Soft errors are errors in logic or data that are caused by radiation, electrical glitches or electro-magnetic interference during the lifespan of a device [10], [6].

ISO 26262, the functional safety standard for automotive products, requires that all features pertaining to safety be rigorously verified. However, the functional verification of ECC circuits poses a formidable challenge. To illustrate, consider the ECC circuit implemented in the program flash interface of an automotive SoC shown in Fig. 1.

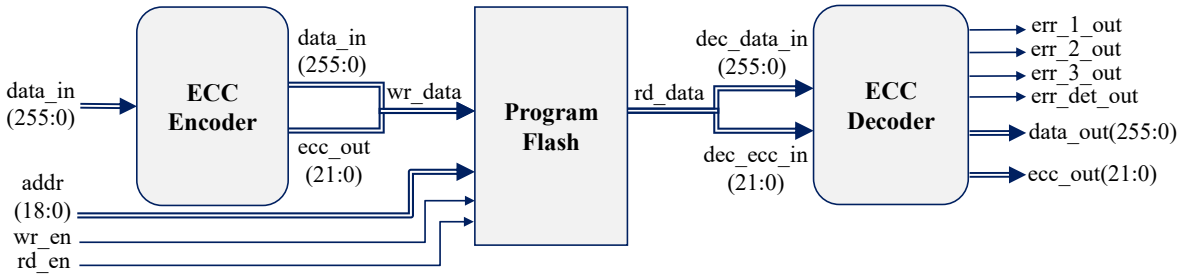


Fig. 1: ECC block in program flash interface
(Double-Error Correct and Triple-Error Detect (DECTED))

The ECC block shown in Fig. 1 is a Double-Error Correct and Triple-Error Detect (DECTED) module. During a write transfer (`wr_en=1`), the input data bits (`data_in`) are encoded in additional ECC bits (`ecc_out`) following a specific encoding algorithm (e.g., Reed Solomon encoding) [4]. The resulting concatenated codeword `wr_data` is written to the program flash memory. Soft errors may occur during the operation of a device, resulting in a corrupted codeword (non-codeword). When a read transfer (`rd_en=1`) takes place, the non-codeword is passed through the ECC decoder. The decoder computes the error syndrome of the non-codeword and sets the respective error flags depending on the number of bit errors. If there are no bit errors, or if the number of bit errors is within the correctable range, the bit errors are corrected and the `data_out` carries the original data value.

Exhaustive verification of ECC circuits requires to consider every possible input and bit error combinations. The ECC circuit shown in Fig. 1 has an astronomically high number of input combinations (2^{256}) and an exhaustive analysis covering the entire input space is impossible. In addition, the input stimuli require to consider an exhaustive set of error injections depending on the detection and correction capability of the circuit. For example, the number of bit error combinations for the ECC circuit shown is given by $\binom{278}{3} + \binom{278}{2} + \binom{278}{1} \approx 3.58 * 10^6$. Because of the huge analysis space (input and error combinations), simulation-based methods including post-silicon debug are inapplicable for the problem of ECC circuits.

On the other hand, formal verification (FV) is inherently exhaustive and checks the design behavior against all possible legal input combinations [1], [9]. The requirements of an ECC circuit can be captured in a set of properties. However, the formal engines which implement systematic SAT solving or graph-based canonical representations of the logic with Binary Decision Diagrams (BDD) often run into complexity issues due to the huge analysis space. As a result, for the type of ECC circuits shown in Fig. 1 the full proof within a reasonable amount of time cannot be expected.

Although FV tools run into complexity issues when the analysis space is huge, they are well suited to prove the underlying mathematical properties implemented by the RTL designs [13], [7]. ECC decoders implement *Syndrome Generators* to identify the bit positions with an error. The output of a syndrome generator is used to set the corresponding error flags and to correct the data vector. Syndrome generators exhibit the characteristics of a linear algebraic function. The input to a syndrome generator, possibly a non-codeword, can be expressed as the sum of a codeword and an error word. The output of the generator is the superposition of the syndromes of both components, i.e., the syndrome of the codeword (which is necessarily zero) and the syndrome of the error word. Hence, proving the linearity of syndrome generators proves that the syndrome output depends only on the erroneous bit positions and not on the input data vector. Once the linearity of the syndrome generator is proven, the properties capturing the detection and correction capabilities of the circuit are proven by assuming a fixed, arbitrarily chosen data vector.

The approach has been successfully applied to multiple ECC designs implemented across several safety-critical automotive SoCs. The proposed approach together with the automated generation of properties yielded high verification quality and productivity. The property runtimes are at least 50 times faster on big ECC designs (width of data vector > 64 bits). Further, following the approach we are able to achieve convergence (full proof) of properties on a 256 bit ECC design (Fig. 1) within 2 hours of computation time using a commercial FV tool. Previously, these properties failed to converge after 100 hours of computation time.

The remaining part of the paper is structured as follows: Section II provides a brief overview of the existing approaches for ECC verification. Section III describes the working of ECC designs and the linearity of syndrome generators. Section IV describes how the linearity property of syndrome generators can be used to prove the correctness of ECC designs using formal methods. Section V describes the automatic generation of properties using an in-house property automation tool. Application of the presented approach on several ECC designs and the property runtimes are detailed in Section VI. A brief summary of the work in Section VII completes the paper.

II. Related Work

ECC designs are typically verified by following the current industry standards for pre-silicon verification, simulation and formal methods. A UVM-based approach for verifying ECC circuits is proposed in [12]. The approach uses assertions to validate the detection and correction behaviors in a coverage-driven verification methodology. For small ECC designs of low data width ($<$ 10 bits), simulation methods can provide the required coverage. However for ECC designs of moderate to large size, simulation-based methods, including both software simulation and hardware-assisted simulation, are inapplicable.

Techniques that use formal methods to validate the correct design of ECC circuits have been proposed. For ECC designs of moderate size (data width $<$ 32 bits), FV tools provide exhaustive analysis and require a permissible amount of computation resources. In [3], an approach is proposed to formally verify large ECC designs with automatically generated properties. The properties are constructed in such a way that only a small window of the data vector (e.g., 16 of 256 bits) is left unconstrained while the rest of the data bits are constrained to a fixed value and assumed to be bit error free. Following the approach, when the size of unconstrained data vector window is increased (data width $>$ 32 bits), the FV tools run into known complexity issues of memory and runtime. Although such approaches provide better coverage than simulation-based methods, they do not provide exhaustive analysis covering all input and bit error combinations.

In [8], a verification technique that is encapsulated in a reasoning tool called 'BLUEVERI' is proposed to verify large ECC designs. The tool is specifically applicable for the logic implementations defined over Galois fields to establish the correctness against mathematical specifications. The design to be verified and a check file containing a set of legal input values and the expected values for certain signals in the design are provided as inputs to the tool. The tool then employs custom-built algorithms to prove the correctness of the logic implementation. A similar approach is proposed in [5] which uses abstract interpretation theory to validate the ECC codes. Although these approaches are applicable to much larger ECC designs, they require extensive knowledge of the design implementation, require more manual effort and are limited to the respective toolchains.

The approach presented in this paper is independent of any specific toolchain and can be realized using any FV tool. The approach makes use of a succinct quality of the formal engines to prove the underlying mathematical reasoning of ECC circuits. The properties are simple and straightforward to develop and can be automated from high-level requirements.

III. Error Correction Codes

Error Correction Codes were primarily created to ensure reliable data transmission over unreliable noisy communication channels. In recent years, in order to safeguard against soft errors and to guarantee a tolerable level of risk, ECC circuits found their usage in safety-critical automotive chips to monitor the data corruption in memory systems. ECC designs consist of two stages: ECC encoding and ECC decoding (as shown in Fig.1).

A. ECC Encoder and Decoder

The function of an ECC encoder is to encode the data bits with additional check bits. The resulting codeword which is formed by combining the data bits and check bits (or parity bits) is written to the memory. Several encoding schemes are available for computing the check bits and a specific encoding scheme is selected based on the width of data bits, the bit error rate (BER), the minimum *Hamming distance* of the code and the error detection efficiency. The Hamming distance is the number of bit changes between two valid codewords. The minimum Hamming distance of the encoding scheme decides the number of bit errors that can be detected and corrected. Hamming codes, Hsiao Codes, Reed-Solomon Codes and Bose-Chaudhuri-Hocquenghem Codes are commonly used encoding schemes for memory elements [4], [11].

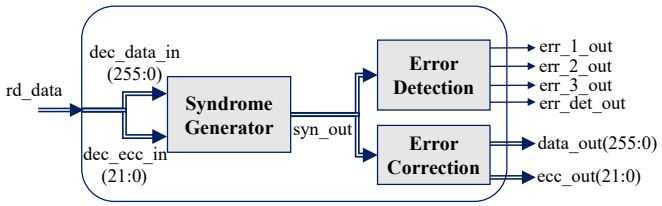


Fig. 2: Block diagram of a DECTED decoder

The function of an ECC decoder is to decode the codeword read from memory and to take required actions depending on the number of bit errors. The block diagram of a typical ECC decoder is shown in Fig. 2. The diagram shows an extended DECTED decoder from Fig. 1 (DECTED = double-error correction / triple-error detection). The *Syndrome Generator* computes the error syndrome (`syn_out`) for the incoming data vector (`rd_data`). The syndrome generation is based on the encoding scheme used to generate the check bits. The error syndrome output identifies the bit positions with an error. The *Error Detection* unit sets the corresponding error flags (`err_1_out`, `err_2_out`, `err_3_out`, `err_det_out`) and the *Error Correction* unit corrects the corrupted bits based on the syndrome output.

B. Linearity of Syndrome Generators

The output of syndrome generators is independent of the codeword and depends only on the bit error positions. In mathematical terms, the syndrome generators implement a *Linear Function*. A linear function preserves additivity, i.e., it satisfies the following rule:

$$f(x) + f(y) = f(x + y) \quad \ldots\ldots\ldots(1)$$

where x, y are arbitrary vector spaces in the field M. A linear function preserves the vector addition and scalar multiplication irrespective of the vector spaces and the scalar value.

We consider Hamming codes for illustrating the linear property of the syndrome generators. Let c be a codeword of width n, formed by combining k data bits and $(n-k)$ parity bits. We call the set C of all codewords an $[n, k]$ Hamming code over the field M. Let us consider a $[7, 4]$ Hamming code with an additional parity bit for the detection of double-bit errors such that its Hamming distance is 3. Besides detecting double-bit errors, such a code is capable of *correcting* single-bit errors. Let d be the word being decoded. The syndrome of the word d is computed according to Eq. 2.

$$\mathbf{syn(d) = H \cdot d^T} \quad \ldots\ldots\ldots(2)$$

where H is the parity check matrix of order $(n-k) \times k$. The parity check matrix is chosen in accordance with the ECC encoder. Vector addition and matrix multiplication in the field M are linear operations based on modulo-2 arithmetic. Let the parity check matrix for the $[7, 4]$ Hamming code be given by:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \ldots\ldots\ldots(3)$$

Let us assume that $c = 0101010$ is the original codeword written to the memory. When there is no bit error, the data read is equal to the original codeword i.e., $d = c$. The syndrome of the codeword c (no error) is given by:

$$\mathbf{syn(c) = H \cdot c^T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \implies \textit{a null vector} \quad \ldots\ldots\ldots(4)$$

Let $d = 0101110$ be a corrupted word (non-codeword), namely codeword c with bit 5 flipped. The syndrome of this non-codeword is given by:

$$\mathbf{syn(d)} = \mathbf{H} \cdot \mathbf{d^T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \implies error\ in\ 5^{\text{th}}\ bit \quad \ldots\ldots\ldots(5)$$

Since vector addition is a linear operation in modulo-2 arithmetic, the non-codeword can be written as the sum of the error-free codeword and the error vector (e) i.e., $d = c + e$. For $d = 0101110$, the error vector $e = 0000100$. The syndrome of e can be computed as:

$$\mathbf{syn(e)} = \mathbf{H} \cdot \mathbf{e^T} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \ldots\ldots\ldots(6)$$

Because of linearity, $syn(d) = syn(c + e) = syn(c) + syn(e)$. By construction of the parity check matrix, the syndrome of a codeword c is always zero: $syn(c) = 0$. Hence, the syndrome of an erroneous non-codeword d depends only on the error word:

$$syn(d) = syn(c + e) = syn(c) + syn(e) = syn(e) \quad \ldots\ldots\ldots(7)$$

IV. Exploiting the Linearity of Syndrome Generator for FV

For ECC designs of moderate size (data width < 32 bits), modern FV tools provide full proof within a reasonable amount of runtime. However, FV tools still suffer from complexity issues for large ECC designs due to reasons outlined in Section I. In Section III, we presented the linearity property of the syndrome generators. Since FV tools are best suited to prove the underlying mathematical reasoning implemented by the RTL designs, the characteristics of a syndrome generator can be captured in properties and proven in a FV tool. Once these properties are proven, the properties capturing the detection and correction ability of the ECC designs can be proven by assuming a fixed, arbitrarily chosen data vector. Exploiting the specifics of syndrome generator characteristics drastically reduces the analysis space for FV tools as demonstrated in the following.

A. FV environment setup for ECC

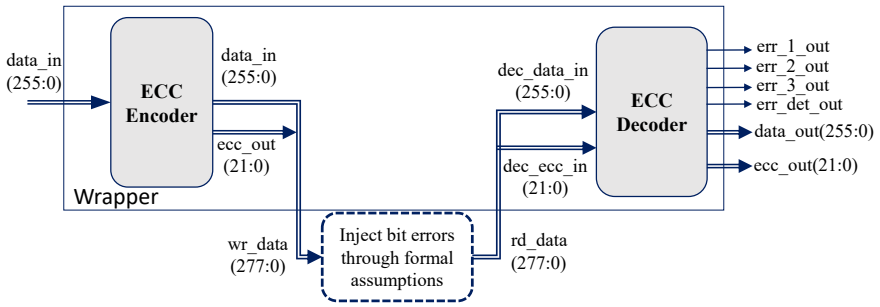


Fig. 3: Formal Verification setup for DECTED ECC design

A general setup for ECC circuit verification with formal methods is depicted in Fig. 3. As the functionality of the program flash interface (from Fig. 1) is irrelevant for the correct functioning of the combinatorial ECC logic, a verification wrapper is used that instantiates only the ECC encoder and decoder components. The encoder's output (`wr_data`) and the decoder's input (`rd_data`) are wired through to the wrapper's interface as shown. The bit errors are assumed referring to the encoder output and the decoder input to specify all possible error scenarios to be handled by the ECC logic.

B. ECC properties without considering the linearity

"Brute-force" properties capturing the double-bit error detection and correction behavior are shown in Fig. 4a and Fig. 4b, respectively. The properties shown do not consider the linearity characteristics of the syndrome generator. Since FV tools are inherently exhaustive in nature, these properties encompass $2^{256} \times \binom{278}{2}$ (input and double-bit error) combinations. Due to the huge analysis space, the full proof for these properties cannot be expected within a reasonable amount of time. During our experiments, the commercial FV tool ran out of available computation memory after 100 hours of computation time.

```
property double_bit_error_detect;
  $countones(wr_data ^ rd_data) == 2
  |->
  err_2_out && err_det_out &&
  !err_1_out && !err_3_out;
endproperty
```

(a) Property for double-bit error detection

```
property double_bit_error_correct;
  $countones(wr_data ^ rd_data) == 2
  |->
  data_out == data_in;
endproperty
```

(b) Property for double-bit error correction

Fig. 4: Properties to detect and correct double-bit errors (in SVA) (without considering the linearity characteristic)

C. ECC properties with linearity

As described earlier, the characteristics of the syndrome generator can be captured as properties and proven in a FV tool. The property shown in Fig. 5a captures the value of syndrome output when there is no error in the codewords. Here `syn_out`is the output of the syndrome generator as shown in Fig. 2. The property shown in Fig. 5b captures the linear characteristic of the syndrome generator. $Syn(x)$ is the RTL function that returns the syndrome vector (`syn_out`) for an arbitrary word x. The *XOR* ($\wedge$) operator computes the bitwise modulo-2 addition. Once these properties are validated in a FV tool, the following inferences are established:

- The syndrome output for an error-free codeword is always a null vector.
- The syndrome generator implements a linear algebraic function. As a result, the syndrome generator preserves the modulo-2 addition operation irrespective of the value of the data vector.

```
property syndrome_error_free_codeword;
  wr_data == rd_data
  |->
  syn_out == 0;
endproperty
```

(a) Property for syndrome output of an error-free codeword

```
property linearity_syndrome_generator;
  Syn(x) ^ Syn(y) == Syn(x ^ y);
endproperty
//x, y are arbitrary words of same width
//'^' operator performs bitwise addition
```

(b) Property for linearity of syndrome generator

Fig. 5: Properties capturing the characteristics of a syndrome generator (in SVA)

Once the properties shown in Fig. 5 are proven, the remaining properties are verified under the assumption of an arbitrarily chosen, fixed data vector as shown in Fig. 6. The property shown in Fig. 6a captures double-bit error detect behavior with a fixed data vector (previously shown in Fig. 4a). Because of assuming a fixed data vector, the input combinations are effectively reduced from 2^{256} to 1. The number of double-bit error combinations that a FV tool has to consider remains unchanged at $\binom{278}{2} = 38503$. Similarly, the double-bit error correct property is also proven by assuming an arbitrarily chosen data vector as shown in Fig. 6b.

```
property double_bit_error_detect;
  data_in == 256'h234FDBE76B23ABF &&
  $countones(wr_data ^ rd_data) == 2
  |->
  err_2_out && err_det_out &&
  !err_1_out && !err_3_out;
endproperty
```

(a) Property for double-bit error detection

```
property double_bit_error_correct;
  data_in == 256'h5346BEAC72643BFD &&
  $countones(wr_data ^ rd_data) == 2
  |->
  data_out == data_in;
endproperty
```

(b) Property for double-bit error correction

Fig. 6: Properties to detect and correct double-bit errors (in SVA) considering the linear characteristics of syndrome generator

D. Coverage perspective

The presented verification approach is compositional and fully validates the correctness of the ECC encoder, the syndrome generator and the error detection and correction logic. For the syndrome generator as a core component of ECC circuitry, we prove the linearity property as shown in Fig. 5b. This property exercises the syndrome generator with its full input space as x, y are arbitrary words. The property in Fig. 5a verifies the encoder and the syndrome generator in a chain. The properties Fig. 5a and Fig. 5b together prove that the output of the syndrome generator is a function of the XOR difference between `rd_data` and `wr_data`, because `wr_data` is a codeword for which `syn_out=0`, and `rd_data` may be a non-codeword with some bits flipped from the original codeword.

After proving the properties in Fig. 5, the correctness of error correction/detection logic is proven based on the Fig. 3 design. We have to consider the full input space for which the design behavior is specified. For Fig. 3 this space does not contain all the combinations of `rd_data` and `wr_data`, because the design specification cares only about the situations when errors can be corrected/detected (here, up to 2/3 bit errors, respectively). Hence, the total input space is given by all situations when the Hamming distance (HD) between `rd_data` and `wr_data` is 0, 1, 2 or 3, i.e., the XOR difference between them has 0..3 ones.

Fig. 6 shows the properties for the case HD = 2, for a specific data input `data_in` of the encoder. Since these properties do not specify/constrain which bit positions are erroneous, the FV tool enumerates all possible bit error combinations $\binom{278}{2}$ such that HD = 2. These properties prove that the syndrome generator drives the error detection and correction blocks such

that they produce correct output, for the specific data input `data_in` of the encoder. When both properties have been proven for all four HD cases then all possible `syn_out` values have been generated and possible error detection/correction cases have been covered. This is independent of the specific data input `data_in` of the encoder.

Implicitly, this experiment also completes verification of the encoder that generates `wr_data`. We prove that decoding the codewords produced by the encoder recover the original data and/or detect/correct any corruption within the specified detection/correction range. As a result there is no need to separately prove the correctness of the ECC encoder.

V. Automated Property Generation

Property development is one of the major steps in the application of formal methods for design verification. The quality of properties is of great importance to ensure the complete coverage of design functionalities. The proposed approach of proving the specifics of a syndrome generator makes the properties for the class of ECC circuits straightforward to develop. Although different ECC designs implement different encoding/decoding schemes for error detection and correction, the properties required to verify the features of ECC designs remain similar. Therefore, an in-house property generation tool has been employed to foster re-usability and verification productivity.

At Infineon, an automation framework based on metamodeling has been widely used for code generation. The framework has been deployed for more than 100 applications and is the source of a high productivity benefit. The framework uses Unified Markup Language (UML) class diagrams for model definition and Python as the generation language. The framework is not only used for RTL generation but also for the automatic generation of properties [2]. The generation flow, which follows the idea of separating the generation steps into multiple layers, is depicted in Fig. 7.

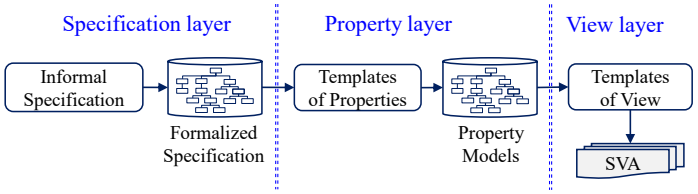
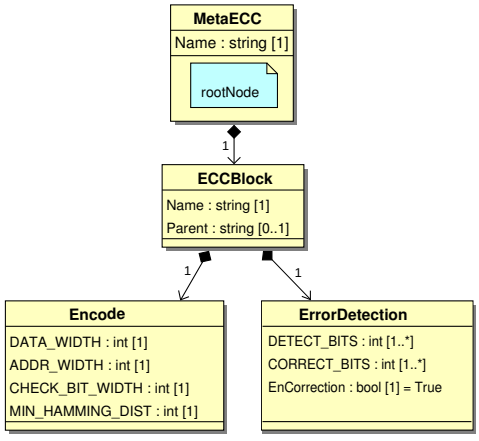


Fig. 7: Property generation flow

The first layer is the *specification layer*, which focuses on capturing the informal specifications in formal specification models. High-level details such as system configurations and intended behaviors of the design are considered. A model definition (metamodel) developed for ECC designs is shown in Fig. 8a. The metamodel shown (as UML class diagram) captures the requirements of an ECC design. The *ECCBlock* contains an *Encode* unit which captures the specifics of an encoder. The *ErrorDetection* unit captures the detection and correction capability of the ECC implementation. From this metamodel different instances can be created for different ECC implementations. An instance created for the ECC block (Fig. 1) implemented in the program flash interface is shown in Fig. 8b.



(a) Specification model definition of ECC designs

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaECC>
   <Name>Flash-Bus-Interface</Name>
   <ECCBlock>
      <Name>ECC_block</Name>
      <Parent>Program-Flash</Parent>
      <Encode>
         <DATA_WIDTH>256</DATA_WIDTH>
         <ADDR_WIDTH>19</ADDR_WIDTH>
         <CHECK_BIT_WIDTH>22</CHECK_BIT_WIDTH>
         <MIN_HAMMING_DIST>6</MIN_HAMMING_DIST>
      </Encode>
      <ErrorDetection>
         <DETECT_BITS>1</DETECT_BITS>
         <DETECT_BITS>2</DETECT_BITS>
         <DETECT_BITS>3</DETECT_BITS>
         <CORRECT_BITS>1</CORRECT_BITS>
         <CORRECT_BITS>2</CORRECT_BITS>
         <EnCorrection>True</EnCorrection>
      </ErrorDetection>
   </ECCBlock>
</MetaECC>
```

(b) Formalized specification for ECC block in Program Flash interface

Fig. 8: Capturing informal specifications in formal specification model

In the intermediate *property layer*, the *Templates of Properties* (ToP) extract the expected behavior of the intended design from specification models and define a property model for each specification item [2]. A property model is a temporal expression trace. The ToP are written in Python, which extract specification details of an encoder (data width, check-bit width, minimum Hamming distance) and decoder (detect-bits, correct-bits) blocks. ToP are implemented such that the property models are defined based on the values of various attributes. For example, the correction property models are defined only if the attribute *EnCorrection* is *True*. Finally in the *view layer*, the property models are mapped to a specific property description language (e.g., SVA) to generate the properties. The properties shown in Fig. 4, Fig. 5 and Fig. 6 are generated by the presented generation flow.

VI. Application on Real Designs

The presented approach has been applied to multiple instances of the ECC circuits implemented in several automotive designs. Our experiments are carried out on 3 different SoCs that are used in safety-critical automotive applications. SoC-1 is the central part of an MCU platform which is used in powertrain applications (including electric and hybrid vehicles) as well as safety applications (such as steering, braking, airbags and advanced driver assistance systems). SoC-2 is also a part of the MCU platform, which enables the electrification of powertrain functions in electric vehicles. SoC-3 is a Lidar sensor used in highly automated driver assistance systems.

A. Design information

Design information about exemplary ECC circuits implemented in the different SoCs is tabulated in Table I. Column 2 shows the type of interface where the ECC module is employed. Columns 3 and 4 show the width of data bits and check bits, respectively. Columns 5 and 6 depict the error detection and correction requirements of the respective ECC modules. The number of logic gates of each ECC implementation is shown in column 7. Since the ECC circuits are combinatorial in nature, the logic gates count represents the complexity for formal analysis.

TABLE I: ECC instances in different SoCs: Design information

Design	Interface Unit	Data bits	Check bits	Error Detection	Error Correction	#Logic Gates
SoC-1	Program flash interface (PF-ECC)	256	22	1,2,3 bit errors	1,2 bit errors	247k
SoC-1	Data flash interface (DF-ECC)	64	22	1,2,3,4 bit errors	1,2,3 bit errors	40.2k
SoC-2	Data memory interface (DM-ECC)	26	6	1,2 bit errors	1 bit error	1.95k
SoC-3	Register file interface (RF-ECC)	16	6	1,2 bit errors	1 bit error	1.2k

B. FV setup and property runtimes

The FV setup for the PF-ECC circuit is shown in Fig. 3. A similar FV setup is used for the remaining ECC modules. For the property development, we used the property generation tool outlined in Section V. Templates of Properties are set up once for the ECC metamodel and reused for all the ECC instances. To setup the initial property generation flow for PF-ECC design, it required 3 person-days of effort. For a new ECC design instance, creating the metamodel instance (Fig. 8b), generating the properties and setting up the regression flow requires one person-day of effort.

The PF-ECC had been previously verified with manually developed properties. Since the linearity characteristics of the syndrome generator had not been considered, complex properties were developed, in which only a small window of the data vector was allowed to be symbolic. The remaining data bits were constrained to a fixed value and assumed to be error-free. This approach is similar to the approach followed in [3]. Overall, the initial property development, setting up the FV setup and refining the properties to attain a full proof required 4 person-months of effort.

TABLE II: Proof runtimes for FV of ECC circuits in different SoCs

ECC instance	Proof runtime without linearity	Proof runtime with linearity	Proof runtime with linearity, induction-based
PF-ECC	Given up after 100 hours	08:54:55	3:03:46
DF-ECC	Given up after 100 hours	06:55:17	–
DM-ECC	00:01:40	00:00:10	–
RF-ECC	00:04:40	00:01:10	–

Note: The proof runtimes shown are for the complete property suite (in the format hh:mm:ss)

The proof runtimes of the ECC properties for 4 different designs are tabulated in Table II. For the property evaluation, we used the commercial FV tool OneSpin 360® DV-Verify on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32GB RAM. Column 2 shows the proof runtimes of the ECC properties where the data vector is allowed to be symbolic (e.g., Fig. 4). With this approach, for PF-ECC and DF-ECC, the FV tool ran out of computation resources after 100 hours of runtime. Column 3 shows the proof runtimes of the complete property suite following the linearity approach. The property suite also includes the properties (shown in Fig. 5) that validate the characteristics of the syndrome generator.

C. Further runtime improvements

In induction-based property checking, a certain operation of the design is separated into 2 or more small operations (sub-operation) and isproven as separate properties (e.g., induction base, induction step). These properties are compiled such that a set of properties capture the complete behavior of an operation [14], [9]. The consequent of the preceding property (e.g., ending state of the sub-operation-1) is part of the antecedent of the succeeding property (e.g., starting state of sub-operation-2). For PF-ECC design, we implemented a similar technique. Since the linearity of the syndrome generator is already proven with a separate property (Fig. 5b), the expression $Syn(x) \wedge Syn(y) = Syn(x \wedge y)$ is added to the antecedent of the remaining detection and correction properties. With this enhancement, the proof runtimes are further reduced. The proof runtime for

PF-ECC properties with further improvements are shown in column 4 of Table II. Overall, the property runtimes for large ECC designs are significantly reduced using the linearity approach. The FV tool requires 1:43:20 (h:mm:ss) to provide a full proof for the triple-bit error detect property on PF-ECC design.

D. Observations

The following observations are drawn from the results obtained by the application of the proposed method on real designs.

Observation 1: The proposed approach of first proving the linearity of the syndrome generator and then proving the remaining properties by assuming a fixed, arbitrarily chosen data vector significantly reduces the property runtimes. On large ECC designs the properties that were previously unable to converge after 100 hours of computation time, converge within 2 hours of computation time, thus providing a significant runtime improvement (at-least 50X on large ECC designs). The proposed approach provides a complete coverage for bit error detection and correction behavior of the ECC circuits and provides a high verification quality.

Observation 2: The proposed approach simplifies a set of properties needed to verify an ECC design. Due to the similarity of bit error detection and correction properties, the properties can be automated for different instances of ECC designs. The property automation tool simplified the property development, enabled the reuse of generation flow for multiple ECC instances and improved the overall verification productivity.

Observation 3: The proposed approach in the paper makes a well-defined contribution to the formal verification of data transformation blocks. Designs that perform data transformation are generally considered as hard to verify with formal methods. The vast input analysis space of data transformation blocks increases the complexity for FV tools. However, a major class of data transformation designs implement specific polynomials (e.g., noise generators, MPEG encoders, MPEG decoders, etc.) to perform the data transformation. When the underlying transformation algorithm (e.g., a polynomial multiplication) has characteristics that are independent of the input data, the properties of the transformation algorithm can be separately proven, as demonstrated in this paper for syndrome generators. This is possible since the formal engines can better abstract the properties capturing the mathematical reasoning of the designs. Afterwards, the remaining properties of the design are proven by assuming a fixed, arbitrarily chosen data word. This approach decomposes the design structurally based on mathematical properties and runs the restricted formal proofs on the design blocks without compromising full coverage. The restrictions on the formal proofs exploit the mathematical property and reduce the search complexity, but still exercise the full design space so that there is no verification gap.

VII. Summary

We presented an approach to exploit the specifics of the underlying algebra implemented by the ECC designs using formal methods. The approach exploits the linearity of the *Syndrome Generator* to establish that the detection and correction of bit errors depends only on the erroneous bit positions and not on the input data word. Proving the linearity feature enables a significant reduction of the analysis space for formal tools. Results show a significant improvement in the property proof runtimes. The ECC properties that were previously unable to converge after long proof runtimes, now converge within few hours of computation time. A high verification productivity has been achieved by combining the approach with automatic property generation. The effort was reduced from months to days. The proposed approach is applicable to all design blocks that implement data transformation such that the transformation is independent of the input data.

References

[1] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1), July 2001.

[2] K. Devarajegowda and W. Ecker. Meta-model Based Automation of Properties for Pre-Silicon Verification. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 231–236, Oct 2018.

[3] K. Devarajegowda, L. Servadei, Z. Han, M. Werner, and W. Ecker. Formal Verification Methodology in an Industrial Setup. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 610–614, Aug 2019.

[4] D. L. Gonzalez. *Error Detection and Correction Codes*, pages 379–394. Springer Netherlands, Dordrecht, 2008.

[5] Charles Hymans. Verification of an error correcting code by abstract interpretation. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, pages 330–345, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[6] N Kanekawa, E. H. Ibe, T. Suga, and Y Uematsu. *Dependability in Electronic Systems: Mitigation of Hardware Failures, Soft Errors, and Electro-Magnetic Disturbances*. Springer-Verlag New York, 1st edition, 2011.

[7] T. Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[8] A. Lvov, L. A. Lastras-Montaño, V. Paruthi, R. Shadowen, and A. El-Zein. Formal Verification of Error Correcting Circuits Using Computational Algebraic Geometry. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 141–148, Oct 2012.

[9] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(11):2068–2082, Nov 2008.

[10] M. Nicolaidis. *Soft Errors in Modern Electronic Systems*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[11] D. Rossi, N. Timoncini, M. Spica, and C. Metra. Error Correcting Code Analysis for Cache Memory High Reliability and Performance. *2011 Design, Automation and Test in Europe*, pages 1–6, 2011.

[12] G. Visalli. UVM-based verification of ECC module for flash memories. In *2017 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, Sep. 2017.

[13] M. Wedler, D. Stoffel, and W. Kunz. Arithmetic Reasoning in DPLL-based SAT Solving. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 30–35 Vol.1, Feb 2004.

[14] M. Wedler, D. Stoffel, and W. Kunz. Exploiting State Encoding for Invariant Generation in Induction-based Property Checking. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 424–429, Piscataway, NJ, USA, 2004. IEEE Press.