Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Verification of Galois Field Based Circuits by Formal Reasoning Based on Computational Algebraic Geometry

Alexey Lvov · Luis A. Lastras-Montaño · Barry Trager · Viresh Paruthi · Robert Shadowen · Ali El-Zein

Received: date / Accepted: date

Abstract Algebraic error correcting codes (ECC) are widely used to implement reliability features in modern servers and systems and pose a formidable verification challenge. We present a novel methodology and techniques for provably correct design of ECC logics. The methodology is comprised of a design specification method that directly exposes the ECC algorithm's underlying math to a verification layer, encapsulated in a tool "BLUEVERI", which establishes the correctness of the design conclusively by using an apparatus of computational algebraic geometry (Buchberger's algorithm for Gröbner basis construction). We present results from its application to example circuits to demonstrate the effectiveness of the approach. The methodology has been successfully applied to prove correctness of large error correcting circuits on IBM's POWER systems to protect memory storage and processor to memory communication, as well as a host of smaller error correcting circuits.

Keywords Galois finite fields · error correcting circuits · formal verification · Buchberger algorithm

1 Introduction

ECCs are widely used in practice to protect data against random errors that inevitably occur during transmission as well as during prolonged storage. As semiconductor technology is scaling down to the nanometer regime and tens of gigabits per second transmission rates, error-free data handling requires larger and more sophisticated error correcting circuits, with the code construction and encoding/decoding algorithms almost always going beyond the templates found in

Alexey Lvov, Luis A. Lastras-Montaño, Barry Trager
IBM T.J.Watson Research Center
Yorktown Heights, NY, 10598
E-mail: {lvov, lastrasl, bmt}@us.ibm.com

Viresh Paruthi, Robert Shadowen, Ali El-Zein
IBM Systems and Technology Group
Austin, TX, 78758
E-mail: {vparuthi, shadowen, elzein}@us.ibm.com

classical literature due to feature set requirements. For example, the IBM z196 systems feature "RAIM" (Redundant Array of Independent Memory, [1], [2]) with a 90 byte ECC that allows the system to recover instantaneously from a full DIMM failure even in the presence of additional chip failures. Each such error correcting circuit has to be individually designed and programmed by a human designer. The resulting implementation complexity in hardware can lead to design errors which can cause costly re-spins of the Silicon and derail schedules. Establishing correctness/verification of such complex hardware is of critical importance, though poses formidable challenges.

Traditional verification methods such as software simulation, hardware accelerated simulation or post-Silicon debug offer insufficient coverage given the difficult nature of the logic and the large solution space to be investigated. State-of-the-art formal verification algorithms (which inherently check circuit behavior against all possible legal combinations of inputs) offering high capacity have been found lacking in proving correctness because of their inability to exploit the specifics of the underlying algebra - Galois field arithmetic.

We propose a solution to the problem of complete symbolic verification of logical circuits which substantially rely on arithmetic over Galois fields. Most of the error correcting circuits fall in the above category, as well as some of the circuits for data encryption and arithmetic logic unit (ALU).

The verification technique is encapsulated in a reasoning tool "Blue Code Verifier" - "BLUEVERI" - and applies algebraic geometry methods (e.g. checks on the consistency of polynomial systems of equations using the concept of Gröbner basis and the associated Buchberger's algorithm) to the problem of verifying circuits defined over Galois fields in order to establish correctness of the logic circuit against a mathematical specification. The methodology has been successfully applied to verify real life error correcting codes at IBM resulting in substantially improved verification quality, by providing full proof of the correctness of the design which was otherwise unobtainable, and in improved productivity, via significantly reduced verification time and effort. We expect the improvements to accumulate as the methodology gets applied "out-of-the-box" to future processor chips employing even stronger ECC designs, and will be key to integrate commodity memories in products as well as in the design of communication link transceivers. The techniques involved are applicable to other types of logic circuitry based on Galois field arithmetic such as Elliptic Curve Cryptography.

2 Previous Art

Simulation-based methods such as software simulation or hardware-accelerated simulation are inapplicable to the problem of complex ECC verification. This is due to the fact that the problem has large numbers of inputs which precludes an exhaustive exploration to fully verify the ECC circuitry to cover all possible combinations of input bit strings and injected errors (within the claimed error correction capability of the code) and check to see if in each case the decoded bit string is equal to the original one. Directed simulation to cover the vast majority, if not all, of "corner cases" again requires a careful analysis of the code to enumerate correction capability and features - a process which is inherently subject to human limitations and errors. Systematic methods such as SAT or graph-based canonical representations of the logic with Decision Diagrams (DD) such as BDDs [3], BMDs [4], FDDs [5] run out of steam quickly due to the large input space and the complexity of the underlying logic employing exclusive-ORs. Our experience suggests that these existing decision procedures have difficulty scaling to designs beyond circuits with more than 24-bit inputs. Enhanced verification techniques leveraging Transformation-based Verification (TBV) [12] concepts to simplify then prove the designs become capacity gated for 32-bit Galois field algorithms and beyond. Satisfiability Modulo Theory (SMT) solvers which utilize specialized theories to address specific problem domains (e.g. bit-vectors) do not address polynomial equation solving over Galois fields. Our approach addresses this niche and proposes a methodology to solve such systems of polynomial equations over Galois fields efficiently.

A search for verification of Galois field circuits reveals the following applicable references - [6] and [7]. [6] defines a formal first-order logic language for symbolic arithmetic over an arbitrary binary Galois field along with a set of rules for manipulation of formal sentences (such as transformation of the sentence into prenex normal form, usage of DeMorgan's law, elimination of variables etc.). The correctness criterion for parts of some ECC circuits can be formally expressed in this language, e.g. finding the error locator polynomial from the value of the syndrome for Reed-Solomon codes. A formal reasoning in the language is then applied to prove or disprove the correctness statement. The method is only applicable to verification of algorithms which are correct in any $GF(2^k)$ independently of the value of k. In our method the size of the field is specified; in particular this allows the use of constants of the field other than '0' and '1' in the circuit. The method does not employ any of the computational algebraic geometry machinery; that bounds it to purely $GF(2^k)$ circuits (with no bit operations allowed), while our method works on circuits with mixed bit and GF signals (Boolean result of test value operations on GF signals is computed by building Gröbner basis of polynomial algebraic system).

The latter [7] applies Gröbner basis techniques to the very narrow problem of verifying multipliers over a large Galois field. The class of the multipliers is further limited to those based on representation of the large field as an extension of degree m of a smaller field of degree n. The paper reports practical results of verifying multipliers up to maximum field size of $GF(2^{1024}), (m = 32, n = 32)$, but it does not make any attempts to verify circuits other than this multiplier circuit with a fixed structure parameterized with only two integers m and n. In contrast our method is capable of verifying virtually any circuit built with GF, Boolean and mixed operations, with the runtime and memory being the only limiting factors for large circuits.

3 Galois Fields and Error Correcting Codes

In this section we give a concise background on finite fields and their applications to error-correcting codes. Refer to [10] pp. 1-286 and [9] pp. 1-146 for a detailed presentation.

3.1 Finite Fields

A Galois (finite) field is a finite set GF of elements together with two binary operations "addition" and "multiplication" from $GF \times GF$ to GF which satisfy all the common laws of addition and multiplication for rational numbers. See [9] pp.11-12 Def. 1.29 for a formal definition.

For any given positive integer number k and any prime number p there exists a unique up to isomorphism Galois field of order p^k ([9] p.49 Theorem 2.5).

Fields of the form $GF(2^k)$ are called binary Galois fields. Each binary Galois field contains a subfield with just two elements '0' and '1' and is a vector space of dimension k over this subfield. Given a basis $b_1, b_2, \ldots, b_k \in GF(2^k)$ the elements of $GF(2^k)$ can be represented as k-tuples of coordinates in this basis or just bit strings of length k. The operation of addition becomes a bitwise XOR in this notation, e.g. "1011" + "0101" = "1110". Due to distributivity of multiplication the product of any two bit strings can be computed by using only the multiplication table for the basis elements $b_1, b_2, \ldots, b_k$ (often called table of structural constants). Here is an example of structural constants for $GF(4)$ and their usage for multiplication:

$\times$	$b_1 =$"10"	$b_2 =$"01"
$b_1 =$"10"	"01"	"11"
$b_2 =$"01"	"11"	"10"

"11" $\times$ "11" =
$(b_1 + b_2) \times (b_1 + b_2) =$
$b_1^2 + b_1 b_2 + b_2 b_1 + b_2^2 =$
"01" xor "11" xor "11" xor "10" =
"11".

Note that there are multiple different ways of choosing a basis in $GF(2^k)$, hence multiple different ways of representing multiplication by structural constants. On the other hand not any structural constant table yields a binary operation that is associative and reversible, so given a Galois field (in whatever abstract form), one must do some work in order to generate a valid table of structural constants. There exists several methods for obtaining some valid table of structural constants for a Galois field of a given size, e.g. [9] pp.66-67 Example 2.51, however we do not need to consider them in detail in this paper.

For the rest of this paper we will work only with binary Galois fields assuming that their elements are represented by bit strings of length k and that the operation of multiplication is given by a table of structural constants.

3.2 Block Data Transmission Model and Error Correction

Data is stored by blocks of length r (Fig. 1a). During one transmission up to t errors (bit flips) may occur.

The idea behind block error correction is mapping of the original r-bit strings to a subset S of the set of all n-bit strings ($n > r$) prior to transmission. Set S is called the set of code words and must have properties
i) $|S| = 2^r$ and
ii) the Hamming distance, that is the number of positions in which bit strings differ, between any two bit strings from S is $\geq 2t + 1$.
The mapping (encoding) of data blocks into code words must be a one-to-one mapping. Generally this mapping is assumed to be given by a table, however for most particular codes short encoding algorithms exist.

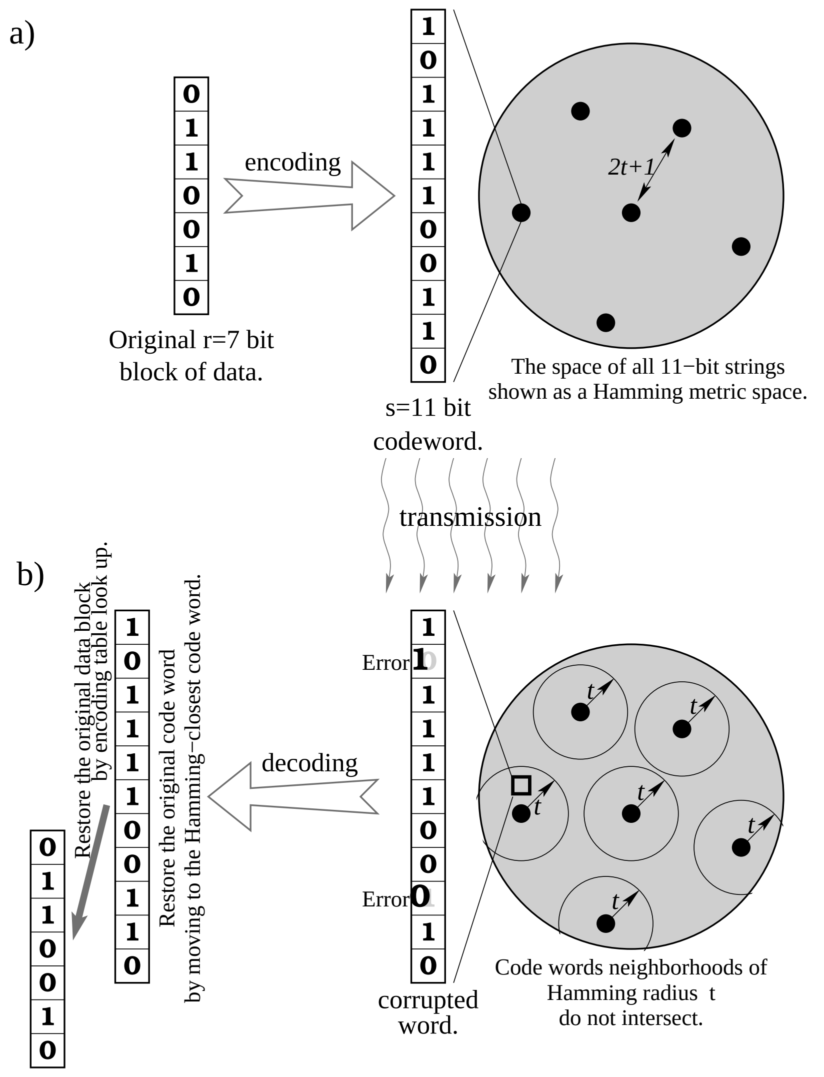
a)
0 1 1 0 0 1 0
encoding
Original r=7 bit block of data.
1 0 1 1 1 1 0 0 1 1 0
s=11 bit codeword.
2t+1
The space of all 11−bit strings shown as a Hamming metric space.
transmission
b)
Error 1
Error 0
corrupted word.
t
Code words neighborhoods of Hamming radius t do not intersect.
decoding
Restore the original code word by moving to the Hamming−closest code word.
Restore the original data block by encoding table look up.

Fig. 1 General schema of block error correction.

During transmission at most t bits of a code word may change their values (Fig. 1b). The received corrupted word is now at a distance $\leq t$ from the transmitted code word and at a distance $\geq t+1$ from any other code word. This property allows to uniquely reconstruct the transmitted code word from the received corrupted word by simply checking all code words in the encoding table and finding the one at a distance $\leq t$ from the received word. It remains to convert the transmitted code word back to the original block of data by using the same encoding table.

Note that our code also can be used for *detection* of up to $2t$ errors instead of correction of up to t errors. In fact, if more than zero but no more than $2t$ bits change their values during transmission then the corrupted word is different

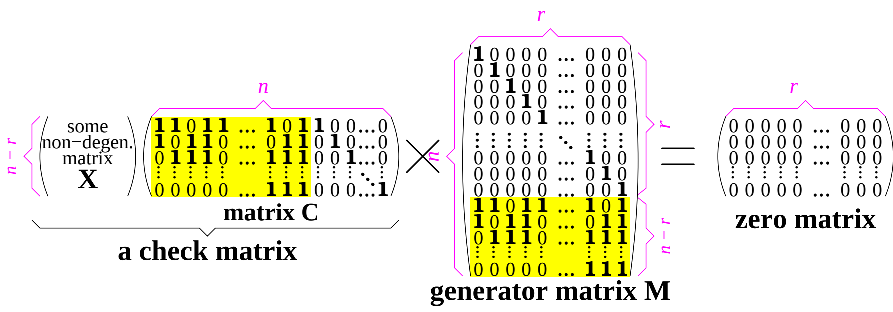

Fig. 2 Generator and check matrices of a systematic linear code.

from the original code word and also could not reach (by Hamming distance) any other code word because our code has a minimum Hamming distance of $2t + 1$ by construction. So our code in $2t$ error detection decoding mode should raise an "uncorrectable error" flag if and only if the received word is not a code word.

In general a block code with minimum distance $2t + 1$ can be used in any of the following decoding modes:
- detection of up to $2t$ errors,
- correction of up to 1 error and detection of 2 to $2t - 1$ errors,
- correction of up to 2 errors and detection of 3 to $2t - 2$ errors,
- . . . . . .,
- correction of up to $t - 1$ errors and detection of t or $t + 1$ errors,
- correction of up to t errors.

3.3 Systematic Linear Codes, Generator Matrix, Syndrome

Consider the space of data blocks as a vector space of dimension r over Galois field $GF(2)$. A code is called a *linear* error correcting code if the encoding mapping is a linear mapping from $\left(GF(2)\right)^r$ to $\left(GF(2)\right)^n$. In other words a code is linear if the encoding mapping has form $x \to Mx$ where M is a fixed r-by-n matrix of $GF(2)$ elements. Matrix M is called the *generator matrix* of the code.

A linear code is called *systematic* if the first r rows of matrix M form a unit r-by-r matrix. In terms of the encoding that means that the first r bits of any code word coincide with the r bits of the encoded data block. Denote the lower $n - r$ rows of M by G, G is an r-by-$(n - r)$ matrix. Denote the result of horizontal concatenation of r-by-$(n - r)$ matrix G and $(n - r)$-by-$(n - r)$ unit matrix by C. $CM = 0$. See Figure 2. Let X be an arbitrary non-degenerate $(n - r)$-by-$(n - r)$ matrix. Any matrix of the form XC is called a check matrix of the systematic linear code; different decoding procedures may use different check matrices.

Note that a check matrix of a systematic linear code is always a full rank matrix which null space is the set of the code words S. Moreover, this is a characteristic property of check matrices: any full rank n-by-$(n - r)$ matrix which null space is the set of the code words can be expressed as XC. Any check matrix of a systematic linear code uniquely defines the generator matrix and very often it is more convenient to describe a systematic linear code by specifying one of its check matrices rather than by the generator matrix.

Theorem 1 ([10] p.59 Theorem 4.11.extension) : A code with check matrix Y has minimum Hamming distance of at least $d+1$ if and only if any d columns of Y are linearly independent.

All decoding procedures for all systematic linear codes have the following common steps:

1/3) Denote the corrupted code word received by decoder by v. Compute $n-r$ dimensional vector $s \overset{\text{def}}{=} XCv$ (the check matrix XC is a fixed property of the decoder). Vector s is called the *syndrome* of v and carries full information about error locations and the presence of uncorrectable errors in the corrupted word. In particular in the assumption that the number of errors does not exceed the error correcting/detecting capability of the code, the syndrome is zero if and only if no errors occurred during transmission.

2/3) Extract the error location information and UE flag from the syndrome. This step varies greatly from code to code and from one decoding algorithm to another. For codes with large ranks and large error correcting capabilities this step usually heavily relies on Galois field algebra. However, given unlimited memory, this step always can be done by brute force, that is by simply constructing a full syndrome to <error locations and UE flag> lookup table.

3/3) Once the error locations are known it remains only to flip the corresponding bits of the corrupted code word to obtain the transmitted code word and then to take the first r bits of this code word which (in systematic codes) form the original data block. In multiple-bits-per-symbol codes that we will discuss in the next subsection an additional step of finding *error magnitudes* is required. This procedure does not depend on the code or decoding algorithm, involves only linear algebra and is very simple in comparison to finding the error locations.

3.4 Multiple Bits per Symbol Codes, Reed-Solomon Codes

All the definitions and statements of the previous subsections remain correct if to replace the 2-symbol alphabet $GF(2)$ by a bigger binary Galois field $GF(2^k)$. For example data may be stored as blocks of bytes instead of blocks of bits and there may be strong grounds to assume that if a single bit in a byte is corrupted then the other bits of that byte are also very likely to be corrupted. An error correcting code over $GF(2^8)$ with minimum Hamming distance (that is the minimum number of different bytes in any two code words) of $2t+1$ will be able to correct any number of bit errors providing that they occurred in no more that t different bytes of a transmitted block. Such error correcting codes are called *burst* error correcting codes.

Possibly the mostly well known family of systematic linear burst error correcting codes are Reed-Solomon codes; in particular Reed-Solomon ECC are used for data protection in all CD and DVD disks. All examples in our Experimental Results section also address Reed-Solomon codes.

Let g be a multiplicative generator of $GF(2^k)$, see [9] p.50 Theorem 2.8. Reed-Solomon ECC is a systematic linear burst ECC with code word length 2^k-1, minimum distance $d+1$, number of check symbols (syndrome coordinates) d and data block size 2^k-d-1.

R.S.ECC(k,d) can be defined by explicitely giving one of its check matrices:

$$\begin{pmatrix} 1 & 1 & \dots & 1 & 1 & 1 \\ g^{2^k-2} & g^{2^k-3} & \dots & g^2 & g & 1 \\ g^{2\cdot(2^k-2)} & g^{2\cdot(2^k-3)} & \dots & g^4 & g^2 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ g^{(d-1)\cdot(2^k-2)} & g^{(d-1)\cdot(2^k-3)} & \dots & g^{(d-1)\cdot 2} & g^{(d-1)} & 1 \end{pmatrix}$$

The generator matrix of R.S.ECC(k,d) can be computed numerically as follows. Its first 2^k-d-1 rows form a unit (2^k-d-1)-by-(2^k-d-1) matrix. The last d rows can be computed by bringing the rightmost d columns of the check matrix to unit d-by-d matrix by elementary transformations of rows (this computation requires $GF(2^k)$ arithmetic). Unfortunately the generator matrix does not have any visible regularity in its structure.

The most notable property of R.S.ECC(k,d) is that it achieves very large minimum distance of $d+1$ by using very few check symbols d.
Proof: Any d columns of the check matrix form a Vandermonde matrix [10] pp. 90-92. So any d columns are linearly independent and by Theorem 1 R.S.ECC(k,d) has a minimum distance of at least $d+1$.

4 Proposed Method

Our method was first inspired by the need to verify a large 1024-bit input error correction circuit responsible for protecting the memory store as well as the communication between a POWER processor and memory. A traditional formal verification approach to verify the circuitry quickly became intractable given the vast search space.

The main idea is to use the fact that algebraic ECCs operate mostly on the elements of finite fields, and there are powerful techniques for symbolic reasoning in this domain. The process of verification of such circuits reduces to the verification of a number of algebraic statements of the type “A certain system of multivariate polynomials over a finite field implies some other system of multivariate polynomials over a finite field”. The latter problem relates to computational algebraic geometry and can be solved by building Gröbner bases for certain sets of polynomials by using Buchberger’s algorithm ([8], pp.77, 82-87).

4.1 Verification Setup

The verification set-up consists of two parts: the circuit to be verified, and a check file containing information about the set of legal inputs and the expected values for some set of “crucial” signals; an example of the latter would be an uncorrectable error flag (see subsection 5.1) or a signal that tests the equality between two bit vectors (see subsection 5.2). The verification task at hand is to formally prove (or disprove) that for any legal combination of inputs, the values of the crucial signals match their expected values.

In a standard processing methodology, the circuit is generally represented by a directed graph where the edges are wires carrying only Boolean signals, and nodes

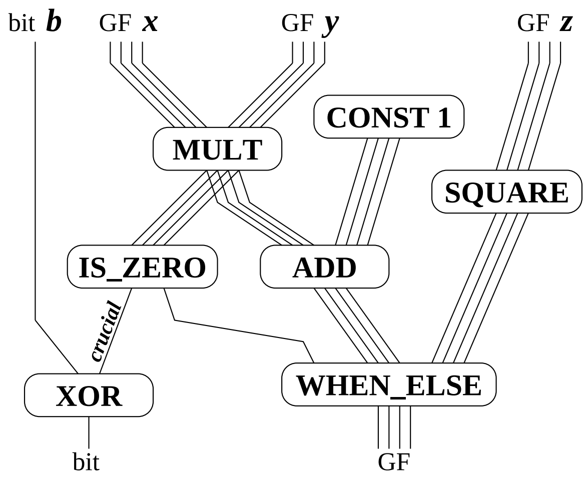

Fig. 3 Example of BLUEVERI circuit representation.

are gates performing only basic Boolean operations. Since we assume that a large portion of the operations in the circuit are operations in $GF(2^k)$ arithmetic, we modify this representation by "glueing" together wires which represent the same $GF(2^k)$ elements and putting "black boxes" around the pieces of the circuit which represent basic $GF(2^k)$ arithmetic operations. Practically this is done by passing a special option to the HDL compiler, telling it to not synthesize functions from a given list. The circuit in our representation typically looks similar to the example on Fig 3.

After this transformation, each wire carries either a Boolean signal or a $GF(2^k)$ signal. For this reason, we generalize the concept of "gate" so that now each gate performs one of the following operations:

- Basic binary arithmetic operations on $GF(2^k)$:
  ADD (both $x+y$ and $x-y$), **MULT** (xy), **DIV** (xy^{2^k-2}).
- Any fixed set of unary operations on $GF(2^k)$ which are linear over $GF(2)$, e.g. Frobenius automorphism (square), projections on elements of a fixed basis, square root, bit permutations etc.
- Any fixed set of $GF(2^k)$ constants (functions without arguments).
- **WHEN_ELSE**(b, x, y) function which returns $GF(2^k)$ element x when bit b is 1 and $GF(2^k)$ element y otherwise.
- $GF(2^k)$ value test functions which return value is a bit:
  IS_ZERO(x), **IS_NONZERO**(x).
- Boolean functions:
  NOT, **AND**, **OR**, **XOR**.

The check file contains algebraic constraints on the $GF(2^k)$ inputs, optionally initial values for some Boolean and $GF(2^k)$ inputs, and the expected values for the crucial Boolean signals testing the desired behavior for the circuit. The crucial signals are restricted to Boolean because any condition on $GF(2^k)$ signals can be expressed as a condition on Boolean signals by adding just a few gates to the circuit. For example, if one wants to state that a GF signal x is equal to a given constant *const*, then one may alternatively assert that we expect

$$\big(\text{IS\_ZERO}(\text{ADD}(x, const))$$

to be equal to 1.

The algebraic constraints are specified in conjunctive normal form (CNF) whose literals are multivariate polynomial equalities or inequalities on the free variables associated with each of the $GF(2^k)$ inputs.

Here is an example of a check file for the circuit on Fig 3:

```
BEGIN_CHECK;

  IN_BITS_SETTINGS;
    b <= '0';

  EXPLICIT_EXPRESSIONS_FOR_SOME_GF_INPUTS;
    x <= "8F3A";

  ALGEBRAIC_CONSTRAINTS_ON_GF_INPUTS;
    [ (y^3 + z^5 == 0) or (y^2 + z != 0) ]
    and
    [ (y == 0) or (z == 0) or (y + z != 0) ]

  BIT_EXPECTED_VALUES;
    crucial must be '1';

END_CHECK;
```

We support multiple checks in one check file in which case our tool verifies them independently one by one, and appending new checks at the end of the file during verification (a necessary feature for the “fork on unresolved bits” mechanism outlined later).

4.2 Verification Flow

The process starts by assigning a free variable (e.g. the symbolic string identifier used in the HDL file) to each of the $GF(2^k)$ inputs. Next the values of the crucial bit signals are computed one by one by applying the following recursive procedure. The procedures for “. . . execute the operation . . . ” will be explained for each type of operation subsequently.

```
COMPUTE_OUTPUT_OF_GATE(signal $g$) {
   // case $g$ is Boolean : Attempt to compute to const. '0' or '1' .
   // case $g$ is $GF(2^k)$ : Compute as a symbolic rational
                               expression in the free variables.
   for all inputs $g_i$ of $g$ {
     COMPUTE_OUTPUT_OF_GATE($g_i$)
   }
   switch (type of $g$) {
     ADD:  . . . Execute the operation . . .
     MULT: . . . Execute the operation . . .
     . . .  . . .
     XOR:  . . . Execute the operation . . .
```

```
    }
}
```

Given unlimited time and memory and assuming that all recursive sub-calls successfully compute values of $g_1, g_2, \ldots$ a call to `COMPUTE_OUTPUT_OF_GATE`(g) always succeeds if g is a $GF(2^k)$ signal. However, it may fail for Boolean signals because Boolean signals are (generally) not constants but depend on the inputs. If a Boolean signal cannot be computed to '0' or '1' we skip to the next check and add two new checks at the end of the check file assuming values '0' and '1' for that bit by applying the "fork on unresolved bit" procedure described later in this subsection. Note that although it may seem that this would fork on nearly every bit in the circuit, in our experience for ECCs the situation is typically just the opposite: given a restricted set of inputs (e.g. exactly one injected error) most of the Boolean signals in the circuit do not depend on the inputs; an example of this can be seen in subsection 5.1 in the computation of the uncorrectable error flag of a decoder [1]. Furthermore, BLUEVERI performs signal dependency checks that result in the value of many boolean signals in the circuit not being needed; such booleans never cause a fork as described above.

Given $g_1, g_2, \ldots$, we compute g depending on the type of operation as follows:
ADD and **MULT**: Perform the operation on the multivariate rational expressions. E.g. $\mathrm{ADD}(\frac{x}{y+z}, \frac{y}{x+z}) = \frac{x^2+xz+y^2+yz}{xy+xz+yz+z^2}$, $\mathrm{MULT}(x+1, y+1) = xy + x + y + 1$ etc.
UNARY_LINEAR_i: Any operation on $GF(2^k)$ which is linear over $GF(2)$ can be given by a linearized polynomial (a polynomial containing only terms of the form cx^{2^t}, see [9] pp.107-124). Substitute the input rational expression into the linearized polynomial. E.g. in $GF(16)$ $\mathrm{Tr}(x) \overset{\text{def}}{=} x^8 + x^4 + x^2 + x$, $\mathrm{Tr}(y + z^3) = y^8 + y^4 + y^2 + y + z^{24} + z^{12} + z^6 + z^3$.
CONST_i: Set signal g to the constant (a rational expression containing no free variables).
WHEN_ELSE(b, X, Y): Set rational expression g to rational expression X if b is '1' and to rational expression Y otherwise.
IS_ZERO, **IS_NONZERO**, **NOT**, **AND**, **OR**, **XOR**: Computation of values of gates with Boolean output constitutes the most complex part of our algorithm.

To compute the value of g we first find the maximal connected island of ancestors of g with Boolean outputs, that is the subgraph consisting of all gates h_j such that there exists a directed path from h_j to g and all gates on this path except for h_j itself are elementary Boolean gates (NOT, AND, OR or XOR). An example is shown on Fig. 4. Note that any value test function (IS_ZERO or IS_NONZERO) in the subgraph must be a top most gate. The input signals g_i of the subgraph are either $GF(2^k)$ inputs of value test functions or Boolean inputs of the whole circuit.

By inductive hypothesis for our recursive function `COMPUTE_OUTPUT_OF_GATE`(g) all $GF(2^k)$-type g_i have already been assigned some rational expression in the free variables, and all Boolean type g_i have been computed to constant '0' or '1' (this is possible for all Boolean inputs to the circuit due to an explicit assignment in the "In bits settings" section of the check which may be set either by the user or as a result of forking on unresolved bits).

[1] Very often the uncorrectable error signal is both an internal signal upon which further things depend and also an output by itself.

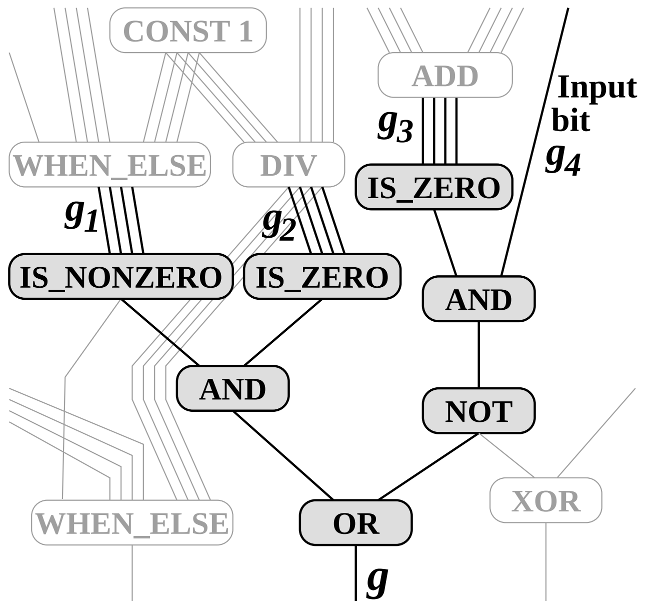

Fig. 4 Example of maximal "algebraic system" subgraph for signal g.

The Boolean function given by the subgraph can be written as a conjunctive normal form whose literals are $g_i = 0$ or $g_i \neq 0$, where g_i are rational expressions. As we will show in the description of DIV operation, we always make sure the denominators of our rational expressions cannot be zero. This allows replacing of $g_i = 0$ and $g_i \neq 0$ literals by numerator$(g_i) = 0$ and numerator$(g_i) \neq 0$ polynomial equalities/inequalities and expressing g as an algebraic system of the form

$$\begin{cases} \big[P_{11}(x_0, x_1, \ldots) =, \neq 0\big] \text{or} \ldots \text{or} \big[P_{1r_1}(x_0, x_1, \ldots) =, \neq 0\big], \\ \quad \ldots \quad \ldots \\ \big[P_{s,1}(x_0, x_1, \ldots) =, \neq 0\big] \text{or} \ldots \text{or} \big[P_{s,r_s}(x_0, x_1, \ldots) =, \neq 0\big], \end{cases} \tag{1}$$

where P_{ij} denote arbitrary polynomials in the free variables $x_0, x_1, x_2, \ldots$ associated with the $GF(2^k)$ inputs of the circuit.

The algebraic constraints on the inputs are also given as CNF, and form an algebraic system of the same type.
g is constant '0' if and only if
{input constraints CNF} AND {g-subgraph CNF} (2)
is unsatisfiable.
g is constant '1' if and only if
{input constraints CNF} AND NOT{g-subgraph CNF} (3)
is unsatisfiable.
Each of the expressions (2) and (3) can be converted to a single CNF of the form (1). It suffices to show how to check whether a system of the form (1) is unsatisfiable.

<`BEGIN` *Satisfiability checking algorithm*>.
The first step is to get rid of inequalities in the system. For each i, j for which we

have inequality $P_{ij}(x_0, x_1, \ldots) \neq 0$ we introduce an auxiliary free variable t_{ij} and replace the inequality by

$$t_{ij} \cdot P_{ij}(x_0, x_1, \ldots) - 1 = 0.$$

One can easily check that if the system before replacement is satisfiable in variables

$$\{x_0, x_1, \ldots, <\text{all previously added auxiliary variables}>\}$$

then the system after replacement is satisfiable in variables

$$\{x_0, x_1, \ldots, <\text{all previously added auxiliary variables}>\} \cup \{t_{ij}\}$$

and vice versa.

All CNF-literals of the new system are polynomial equalities. Denote them by

$$Q_{ij}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) = 0.$$

Next we replace all OR operations with multiplication:

$$\begin{cases} Q_{11}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) \cdot \ldots \cdot Q_{1r_1}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) = 0, \\ \ldots \quad \ldots \\ Q_{s,1}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) \cdot \ldots \cdot Q_{s,r_s}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) = 0. \end{cases} \tag{4}$$

System (4) is a regular algebraic system of multivariate polynomials over $GF(2^k)$.

By Hilbert's Weak Nullstellensatz a system of multivariate polynomials is unsatisfiable over an algebraically closed field if and only if the ideal generated by the polynomials of the system coincides with the whole ring (i.e. contains 1) (refer [8], pp. 169-173).

$x \in GF(2^k)$ if and only if $\big[x \in \text{alg\_closure}\big(GF(2^k)\big)$ AND $x^{2^k} - x = 0\big]$.
For each variable x_i add equation $x_i^{2^k} - x_i = 0$.

$$\begin{cases} Q_{11}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) \cdot \ldots \cdot Q_{1r_1}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) = 0, \\ \ldots \quad \ldots \\ Q_{s,1}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) \cdot \ldots \cdot Q_{s,r_s}(x_0,\, x_1,\, \ldots,\; t_{i_0 j_0},\, t_{i_1 j_1},\, \ldots) = 0, \\ x_0^{2^k} - x_0 = 0, \\ \ldots \quad \ldots \\ x_{\text{last}}^{2^k} - x_{\text{last}} = 0. \end{cases} \tag{5}$$

System (5) is satisfiable in the algebraic closure of $GF(2^k)$ if and only if the original system (1) is satisfiable in $GF(2^k)$.

It remains to build a Gröbner basis of the ideal given by the polynomials of system (5). This can be done by Buchberger's algorithm ([8], pp. 77, 82-87). The original system (1) is unsatisfiable in $GF(2^k)$ if and only if this Gröbner basis contains 1.
<`END` *Satisfiability checking algorithm*>.

If the value of g is proved to be a constant '0' or '1' assign this value to g (computation successful). Otherwise fork on the unresolved Boolean signal g as follows:

Add two copies of the current check at the end of the check file as given below.

- If g is an input Boolean signal add `g <= '0'` to the "In bits settings" section of copy_1 and `g <= '1'` to the "In bits settings" section of copy_2.
- Otherwise add NOT(System (1)) to the conjunctive normal form in "Algebraic constraints on GF inputs" section of copy_1 and System (1) to the CNF in "Algebraic constraints on GF inputs" section of copy_2.

Skip the current check and continue to the next one with the two additional checks added at the end of the queue. As a side note, the two examples in subsections 5.1 and 5.2 do not require branching of this type for completion.

The only operation we have not explained yet is division. **DIV** : In logical circuits division is usually implemented as `if` $y \neq 0$ `return x/y; else return 0;` (which is equivalent to xy^{2^k-2}). To compute the result of division we first attempt to prove that the constraints on the inputs imply that the divisor is either always $= 0$ or always $\neq 0$ by the same algebraic method as for the gates with Boolean output. If successful, we simply assign 0 or the rational $expressionx/y$ to g. Otherwise we fork on the test of [denominator $= 0$] the same way as shown above for non-input Boolean signals.

We have shown how to compute value of any gate given the values of its inputs. $GF(2^k)$ signals are computed as symbolic rational expressions in the input signals, and Boolean signals must compute to constant '0' or '1' creating new branches with additional algebraic constraints on the inputs if necessary. This completes the description of our algorithm.

Our actual C implementation contains many more features than described above. The most important ones include:

- Careful manipulations of conjunctive normal form systems: A brute force manipulation of CNFs, and opening parenthesis in polynomial products which come from large OR-clauses would cause an immediate exponential explosion of the size of the system. However special care is taken of systems of the form (1) which most commonly appear in algebraic circuits. This prevents a rapid increase of the size of the system - at least for typical cases. In particular, if g-CNF has only one OR clause of length ≥ 2, i.e. has form $\Big([P_{11} =, \neq 0]\,\texttt{OR} \ldots \texttt{OR}\,[P_{1r} =, \neq 0]\Big)\,\texttt{AND}\,\ [P_{21} =, \neq 0]\,\texttt{AND} \ldots \texttt{AND}\,[P_{s,1} =, \neq 0]$, our implementation ensures the size of any system for which we build a Gröbner basis is simply equal to the sum of the sizes of the input constraints system and g-CNF system.
- "Lazy" signal computation method: In order to find values of expressions such as ('1' *or* x), ('0' *and* x), (when '1' : *const* else x) etc., we do not compute x. This gives a significant speed up especially when the signals whose values we need to verify are localized in a relatively small part of a large circuit.
- Verification flow control: The user can control a number of verification process options such as whether to spend more time on Gröbner basis computation of a given bit vs. fork; whether to attempt to save time by skipping the $x \in GF(2^k)$ constraints which makes false negatives (but not false positives) possible; etc.

4.3 Verification Result

The verification process can have three possible outcomes:

1. For all checks all crucial bit values are computed and match the expected values.
2. One of the checks (including checks added by "fork on unresolved bit") fails because the value of one of the crucial bits is opposite to the expected value specified in the check file.
3. One of the checks (including checks added by "fork on unresolved bit") fails to compute one of the crucial bit values due to insufficient time or memory.

"BLUEVERI" is primarily targeted at giving a formal proof of the correctness of the design. In case of a failure (situation 2 or 3) it does not automatically generate a counter example but instead provides symbolic values for as many signals of the circuit as the time and memory constraints permitted it to compute. An interactive bug tracing interface then allows the user to browse the graph of signals and view their values in the form of symbolic rational expressions and algebraic systems. Usually ECC algorithms are described and proved by human designers in a very similar form: algebraic manipulations with variables over finite fields. So in most cases the ability to see the algebraic expressions for intermediate steps of the computation allows the programmer to compare the execution of the circuit directly to the original human readable description of the ECC given in a book or a paper and find the bug.

However if an explicit counter example is necessary it can be manually found in situation 2 (the opposite statement proved explicitely) by a sequence of binary subdivisions of the set of admissible input signals. That is: fix bit 1 of input signal 1 to '0' and run the test, then fix bit 1 of input signal 1 to '1' and run the test. One of these two subtests must explicitely fail giving an explicit value of bit 1 of signal 1 of counter example. Set bit1-sig1 to this value and do a fork test for bit2-sig1 in the same fashion. Continue until the values of all bits of all input signals of counter example are computed.

In situation 3 (timed out / out of memory) nothing can be said about the circuit for sure. However if the structure of the circuit reflects the original algebraic ideas of the ECC well enough it is very likely that an excessive runtime is due to a presence of a bug in the implementation. In that case a counter example can not be found by either binary subdivision or Gröbner basis approach because finding a solution to an algebraic system is always more time and memory consuming than simply showing that the set of solutions is non-empty. One practical attempt of resolving this problem is performing the following two mutually complementary computations in parallel until one of them succeeds:

1) Attempt to obtain a formal proof of correctness by running BLUEVERI with gradually increasing time and memory limits.

2) Attempt to obtain a counter example by continuously running "good old" random sampling test.

5 Experimental Results

If there is no restriction on time and memory the verification process is guaranteed to prove or disprove the specification in the check file. We will give in what follows two simple examples (subsections 5.1 and 5.2) where this is accomplished within a reasonable amount of time, demonstrating the power of reasoning at the Galois field level as opposed to the Boolean level. For complex, real-life designs (as exemplified in subsection 5.4) we have found it useful to help BLUEVERI by manually partitioning the search space, resulting in very little use of the "forking" feature described earlier. In addition, in some instances care is taken to specify the circuit in otherwise equivalent forms to aid BLUEVERI in keeping down the size of its internal rational expressions and the complexity of algebraic systems it generates; this was not necessary in the two examples below.

5.1 The Uncorrectable Error Flag of a Sample Reed-Solomon Decoder

As a first example, we consider a Reed-Solomon code with symbols belonging to a finite field $GF(q)$ with $q = 2^k$ elements for some integer k. We shall assume that the length of this code is $n = 2^k - 1$. Let d denote the number of check symbols of the Reed-Solomon code. We assume that this Reed-Solomon code has been furnished with a decoder that is capable of correcting any one symbol error, and can detect up to $d-1$ different errors. This decoder has a number of different components, one of which is responsible for the computation of the uncorrectable error flag. This flag is a single Boolean output that is raised whenever the decoder has detected 2,3, or up to $d-1$ errors, and kept low whenever the error scenario corresponds to a single error, or alternately whenever there is no error.

For our choice of Reed-Solomon code, the d-symbol syndrome of this Reed Solomon code can be computed from a (potentially corrupted) encoded vector $v \in GF(q)^n$ using the formula

$$S_i = \sum_{j=0}^{n-1} v_j \omega^{ij} \qquad (w^{ij} \text{ means raise } w \text{ to the power of } ij)$$

for $i \in \{0, \cdots, d-1\}$, where ω denotes a primitive element of the field. Denote the *error vector* affecting v by e. $e \in F_q^n$, $v = e + x$ where $x \in F_q^n$ is the uncorrupted codeword. Vector x has zero syndrome, therefore v can be replaced by e in syndrome computation.

$$S_i = \sum_{j=0}^{n-1} e_j \omega^{ij} \tag{4}$$

The design of the uncorrectable error flag for this scenario is a well understood problem; for the sake of demonstration we deduce what might be a reasonable method to test it directly through formal methods. It can be easily seen from (4) that if there is only one error in e then the syndromes have form

$$S_i = \text{const1} \cdot \Big(\text{const2}\Big)^i$$

and satisfy the following condition: $S_i S_{i+2} = S_{i+1}^2$ for $i = 0, \cdots, d-3$. Furthermore it is also known whenever e has at least one error and at most d errors, one or more of the $\{S_i\}_{i=0}^{d-1}$ is nonzero. Therefore one can compute the uncorrectable error flag through the following code, written using BLUEVERI VHDL style semantics:

symbol errors	expected UE	8 bit symbols		
		BLUEVERI	input bits	SXS
1	false	Success after 0.1 s.	16	Success after 14 s.
2	true	Success after 1 s.	32	Gives up after 24 h.
3	true	Success after 1 s.	48	N/A
4	true	Success after 33 m.	64	N/A
5	true	Gives up after 6 h.	80	N/A

symbol errors	expected UE	4 bit symbols		
		BLUEVERI	input bits	SXS
1	false	Success after 0.1 s.	8	Success after 0.7 s
2	true	Success after 1 s.	16	Success after 3 s
3	true	Success after 1 s.	24	Success after 55 m
4	true	Success after 33 m.	32	Gives up after 24h
5	true	Gives up after 6 h.	40	N/A

Table 1 Experimental results for the formal verification of the uncorrectable error flag of a single error correct, multiple error detect Reed-Solomon decoder. SXS refers to Sixth Sense, a bit-level formal verification tool set developed at IBM.

```
t_comp : for i in 0 to d-3 generate
  t(i) <= add( mult(s(i),s(i+2)) , square(s(i)) );
end t_comp;
snz <= is_nonzero(s(0)) or ... or is_nonzero(s(d-1));
tnz <= is_zero(s(0)) or ... or is_zero(s(d-1))
       or
       is_nonzero(t(0)) or ... or is_nonzero(t(d-3));
UE  <= snz and tnz;
```

As written above, `snz` and `tnz` represent two distinct systems of equations which BLUEVERI will treat independently of each other. On the other hand, BLUEVERI will attempt to establish whether `tnz` (for example) is true or false by examining the properties of `s(0), ... s(d-1), t(0), ... t(d-3)` simultaneously as opposed to testing whether each `s(i), t(i)` is zero or not individually.

In order to test the ability of a model checker to prove the correctness of this implementation of the uncorrectable error flag, we assume that the syndrome generation portion of the decoder has been proved correct separately; this task is in fact generally computationally simpler than the one currently at hand. We then build a module that accepts inputs `e_m(0)...e_m(t-1)` (for the error magnitudes) and inputs `l(0)...l(t-1)` (for the error locations) where t is the maximum number of errors one can inject into the decoder during the test; in this particular example for the uncorrectable error flag to be correct it is known that $t = d - 1$. This module emulates the syndrome generator and computes `s(0)...s(d-1)` using the equation `s(i)` $= \sum_{i=0}^{t-1}$ `l(i) e_m(i)` (as per Equation 4), and then passes the resulting syndromes to a module that computes the uncorrectable error flag as previously described.

In order to test a variety of error scenarios, we can place constraints on `e_m(i)` and `l(i)`. For example, one can restrict the test to have exactly two errors by specifying the following constraints:

```
e(0) != 0, e(1) != 0, l(0) != 0, l(1) != 0
add(l(0),l(1)) != 0, e(1) = ... = e(t-1) = 0
```

Note that in a field of characteristic 2, addition is equivalent to subtraction, and hence the addition constraint effectively constrains `l(0) != l(1)`. These constraints can be specified in a BLUEVERI check file as equal/not equal to zero conditions on multivariate polynomial expressions. When BLUEVERI examines the dependencies of the `UE` signal, it finds that it depends on `snz` and `tnz`. BLUEVERI must either resolve that both are true, or that at least one of them is false. As described earlier, this is accomplished by invoking an attempt to compute the Gröbner basis of various system of equations related to the constraints and the expressions defining `snz` and `tnz`. Similar experiments can be conducted by updating the constraints to specify "at least two, but not more than y errors" where y is a number between 2 and $d-1$.

In order to test the capability of BLUEVERI as applied to this problem and contrast it with that of a formal prover (we chose SixthSense, IBM's state-of-the-art formal and semi-formal verification tool set, for that purpose), we set up a test with $d = 8$, $k = 8$ (using $GF(256)$) and with the capability to inject from 2 up to 7 errors at arbitrary locations, since the corresponding Reed-Solomon decoder is supposed to be able to detect all those errors. We also set up a parallel test with $k = 4$ which is a considerably simpler problem for a Boolean oriented formal verification system such as SixthSense [12]. The SixthSense and BLUEVERI experiments do not have any special tuning of the VHDL or the tool to improve the outcomes.

We refer the reader to Table 1 where the experiments were performed in a single processor (POWER6 processor @ 5GHz running AIX) and the SixthSense was run as a single software thread mainly orchestrating redundancy removal and SAT algorithms. In this set of experiments, BLUEVERI was configured to reason about the circuit with the variables (due to inputs or constraints) belonging to the *algebraic closure* of the fields. This in essence means that we did not constrain the variables to belong to the field $GF(256)$ (resp. $GF(16)$) depending on whether the symbols used were 8 bit (resp. 4 bit) symbols. The consequence of this is that although the BLUEVERI results are listed under 8-bit column, they in fact hold *for any field size*, including larger field sizes which would be even harder for a bit-level verification system to handle. Both formal systems were able to prove the correctness of the uncorrectable error flag under the single error scenario quite easily, but SixthSense was not able to prove the correctness of this flag in the double error case in the amount of time indicated in the table. In order to test the sensitivity of SXS to the field size, we performed a similar experiment for a Reed-Solomon code defined over $GF(16)$. In this case we saw better results from SixthSense, since we were able to prove the correctness of double and triple error detect cases but not four error case. It is worth noting that the field size determines many important properties of an error control code, including the total codeword length, and thus it cannot be modified for the purposes of formal verification since the resulting code is entirely different and, in all likelihood, not applicable to the original problem.

5.2 Computing Error Magnitudes in a Reed-Solomon Code

One of the tasks that an error control decoder for a code defined over multibit ($q > 2$) symbols must perform is to compute the locations of the symbols in error and then to compute the multibit pattern that one must add to those locations in order to correct the codeword. This multibit pattern is called the *error magnitude*. Suppose that there are t errors in a codeword, and let `s(0)`, $\cdots$, `s(t-1)` be the first t syndromes (note that this example is for a different setting than the example in the

errors	8 bit symbols			4 bit symbols	
	BLUEVERI	input bits	SXS	input bits	SXS
2	Succ. 2 s.	32	Gives up after 24h	16	Succ. 0.6s
3	Succ. 2.1 s.	48	N/A	24	Succ. 16m
4	Succ. 2.1 s.	64	N/A	32	Gives up after 24h
5	Succ. 2.3 s.	80	N/A	40	N/A
6	Succ. 3.1 s.	96	N/A	48	N/A
7	Succ. 49.4 s.	112	N/A	56	N/A
8	Succ. 8m	128	N/A	64	N/A
9	Succ. 53m	144	N/A	72	N/A

Table 2 Experimental results for the formal verification of the error magnitude computation stage of a Reed-Solomon code.

previous subsection). From (4), we can derive that error magnitude computation can be carried over using the equation

$$\begin{bmatrix} \texttt{e\_m(0)} \\ \vdots \\ \texttt{e\_m(t-1)} \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ \texttt{l(0)} & \cdots & \texttt{l(t-1)} \\ \vdots & \ddots & \vdots \\ \texttt{l(0)}^{t-1} & \cdots & \texttt{l(t-1)}^{t-1} \end{bmatrix}^{-1} \begin{bmatrix} \texttt{s(0)} \\ \vdots \\ \texttt{s(t-1)} \end{bmatrix}$$

The inverse matrix above can be derived analytically. It is well known that the inverse is non singular if and only if the locations `l(i)` are all distinct of each other. This restriction can be specified through $\binom{t}{2}$ constraints each of which is a polynomial with two monomials. We refer the reader to Table 5.2 where we show that in this case, BLUEVERI was able to show the correctness of the corresponding circuit with up to 8 errors, while SixthSense was unable to finish the double error case within the time allocated. As in the previous subsection, in this particular example the result for BLUEVERI is actually field size independent since it exploits only the algebraic properties of the symbols. It is worth noting that the Gröbner basis machinery in BLUEVERI does get involved in proving the correctness of this circuit. This is because the inversion of the Vandermonde matrix results in rational expressions (as opposed to plain polynomial expressions) whose denominator could be zero. The task of Gröbner in here then is to show that the denominator is not zero given the assumptions on the inputs, so that BLUEVERI can proceed with the corresponding algebraic simplifications leading to the desired result.

5.3 Solving the Key Equation for a Reed-Solomon Code

The standard decoding procedure for correcting random errors in Reed-Solomon codes consists of 3 stages. In the first stage syndromes are computed from the potentially corrupted encoded vector. This has been described in previous sections and as noted, verification of syndrome generators is very easy using BLUEVERI technology, even for very long codewords and large numbers of syndromes. This is essentially due to the fact that these circuits have no branching and are linear functions of the data, which is the easiest case for BLUEVERI. The next stage involves using the syndromes to compute a pair of polynomials which can be used by the final stage for locating and correcting the errors. One polynomial $\Lambda(x)$ is called the error locator polynomial since its roots can be used to determine the locations of the errors in the codeword. The other polynomial $\Omega(x)$ is call the error evaluator polynomial since it is used along with the error locator polynomial to compute the values of the errors at each error location. This pair of polynomials are minimal solutions to a particular congruence equation called the "Key Equation". If we have r syndromes, $S_0, \ldots, S_{r-1}$, then we first form the syndrome polynomial $S(x) = \sum_{j=0}^{r-1} S_j x^j$. $\Lambda(x)$ and $\Omega(x)$ are the least degree solution to the equation:

$$S(x)\Lambda(x) \cong \Omega(x) \pmod{x^r}$$

There are several algorithms for solving this equation, but a recent algorithm [13] due to Sarwate and Shanbhag has the advantages not requiring divisions and reducing the latency to only 1 Galois Field multiplication followed by 1 addition. We decided to implement this algorithm as part of a high performance Reed-Solomon decoder operating over the field GF(1024) (i.e. k=10) and with r=14 syndromes. In this situation we can correct up to 7 random errors. Although we were implementing a published algorithm, we needed to be sure our implementation was correct so we attempted to verify it using BLUEVERI.

Our implementation was a single latched module which iterates 14 cycles in order to produce the locator and evaluator polynomials. Since the current implementation of BLUEVERI requires pure combinatorial logic (no latches), we needed to transform our implementation into a latch free version. Our circuit operates in a simple feedback manner and always iterates exactly 14 times, therefore the transformation simply consisted on cascading 14 copies of our circuit with the latches removed, feeding the output of each stage as input to the next stage. Even though our circuit has no divisions, it does have one branch point during each iteration which depends strongly on the input syndromes. Thus without constraining the syndromes, BLUEVERI would be unable to decide which branch to take and would need to use its forking mechanism to explore all branches. Since verification of this circuit is an extremely difficult task we decided to avoid using the forking mechanism, and instead we added constraints to the input syndromes to force the algorithm to take a single path through the circuit. Since there is only 1 branch point per iteration and we iterate 14 times, this would mean at most $2^{14} = 16384$ cases to be handled.

In general it can be a difficult problem to find the constraints on the inputs in order to force an algorithm to take a particular branch; this is why the forking mechanism is needed to handle some situations. For Sarwate's algorithm the branch points are determined by the sequence of "discrepancies", which are inner products between the candidate current locator polynomial coefficients and sliding sequences of syndromes. However one can show that the branching is also controlled by particular determinants of a matrix built from the syndromes. For any $k \leq 7$, the locator polynomial for exactly k errors can also be determined from the $(14-k) \times (k+1)$ syndrome matrix constructed in the following way. In row 0 we put syndromes 0 to k, in row 1 syndromes 1 to $k+1$ and generally in row i we put syndromes i to $k+i$. As an example, the syndrome matrix for 5 errors would be:

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \\ s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \\ s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \\ s_4 & s_5 & s_6 & s_7 & s_8 & s_9 \\ s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \\ s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} \\ s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} \\ s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} \end{bmatrix}$$

If there are exactly 5 errors then this matrix would have rank 5 and a generator of its 1 dimensional nullspace gives the coefficients of an error locator polynomial. In general for k errors the determinant conditions on the $(14-k) \times (k+1)$ syndrome matrix are that it has rank k and that the principal $k \times k$ subdeterminant is nonzero. Since the principal $k \times k$ matrix has full rank, the condition of rank k can be verified by testing that the determinants of all $(k+1) \times (k+1)$ submatrices formed from $k+1$ sequential rows of this matrix are zero. In light of this, we generated input constraints for BLUEVERI by asserting the appropriate symbolic determinants to be zero or nonzero. The situation of k errors does not uniquely determine the branching in Sarwate's algorithm, since it also depends on the determinants of the $k-1$ principal minors of size less than k. Each of these can be zero or nonzero and there are thus 2^{k-1} subcases for k errors. So the input constraints for each subcase of k errors contain the k determinants of principal minors and the $14-2*k$ determinants of all $(k+1) \times (k+1)$ sliding blocks. Thus the input constraints for k errors are determinants of submatrices of size at most $k+1$.

During our verification exercise, we were able to verify many of the cases described previously. Most cases we were unable to handle were due to size constraints, as on some occasions, the intermediate expressions exceeded address space limitations since some of the tools we were using were only available as 32 bit applications and not 64 bit applications. For cases when we couldn't completely verify k errors using all 14 syndromes, we reduced the problem by either reducing the number of syndromes or setting some of the syndromes to zero since the number of syndromes is constrained to be at least twice the number of errors. We summarize below what we were able to demonstrate:

1. All cases 1,2, or 3 errors with all 14 syndromes.
2. All cases of 4 errors with 12 syndromes (last subcase took 2 days to verify).
3. All cases of 5 errors with 10 syndromes.
4. Most subcases of 6 errors with 12 syndromes and first syndrome set to zero.
5. Most subcases of 7 errors with 14 syndromes and first 2 syndromes set to zero.

The reader may notice that cases with more syndromes than twice the number of errors require more effort on the part of BLUEVERI. This is because even after finding the correct error locator, on successive iterations the locator is scaled in Sarwate's algorithm by a computed constant in $GF(2^k)$. Since the actual value of this constant depends on the syndromes, performing these successive iterations in BLUEVERI involve multiplying the coefficients of the locator polynomial by an symbolic expression in the syndromes which continue to cause their size to grow making subsequent computations more expensive.

This helps explain why, for example, proving the case for 4 errors and 12 syndromes just barely worked, even though only the first 8 syndromes are needed to get the locator polynomial. The difficulty comes in subsequent iterations involving the remaining syndromes which continue to increase the size of the locator polynomial, making **IS_ZERO** questions much more difficult. Even these partial verification results are remarkable since we are not aware of any other verification tools which could be successfully applied to a circuit of this complexity.

5.4 A Note on a Real Life Application of BLUEVERI

The examples in the previous subsections are meant to illustrate the capabilities of a formal verification system such as BLUEVERI when compared to Boolean oriented systems. In our experience, the implementation of a real-life encoder/decoder employs many custom algorithm variants as one tries to address problems that are specific to the application at hand. In the most significant application of BLUEVERI so far, we have succeeded in proving the correctness of an ECC of a POWER microprocessor that is based on the mathematics of Reed-Solomon codes. The correctness criteria included all correctable and uncorrectable cases for which we had given guaranteed behavior (e.g. recovery from complete chip failures and detection of multiple errors). The ECC, from the decoders perspective, had over 1000 bits of input including several tens of bits worth of configuration parameters. The number of syndrome bits produced by the decoder was over 100 bits, although our testing did include testing the behavior of the encoder with analytically generated symbolic syndromes, it was not limited to it - approximately half of the total testing time exercised the more than 1000 bits of input of the circuit directly. The number of Galois field and Boolean elements in the corresponding graph is over 100,000 (compared to at most a few hundred in the previous experiments). Because of the complexity of the problem, we had to case-split to create 1M different tests, each of which exercised formally a particular region of the test space. It took about 2 weeks to prove the correctness of the entire design in a 10 machine Linux (x86) cluster.

6 Technical Solutions

The BLUEVERI tool leverages IBM's existing front-end and simulation tools and flows. For language processing we are using Portals, IBM's HDL compiler, which accepts the synthesizable subset of standard VHDL and Verilog languages. Portals performs behavioral synthesis on procedural HDL and produces an elaborated netlist, for BLUEVERI this is in the DADB logic database. DADB is a box-pin-net logic database used for verification flows, such as topology checking and simulator model build, which supports client transforms via a dynamically loaded plugin architecture.

Portals was modified for BLUEVERI to support the blackboxing of function calls, enabling the logic to be represented in a form amenable to analysis by BLUEVERI. High level language constructs which are output by Portals into the netlist, such as case statements, can be synthesized into lower level representations by the use of DADB client transforms.

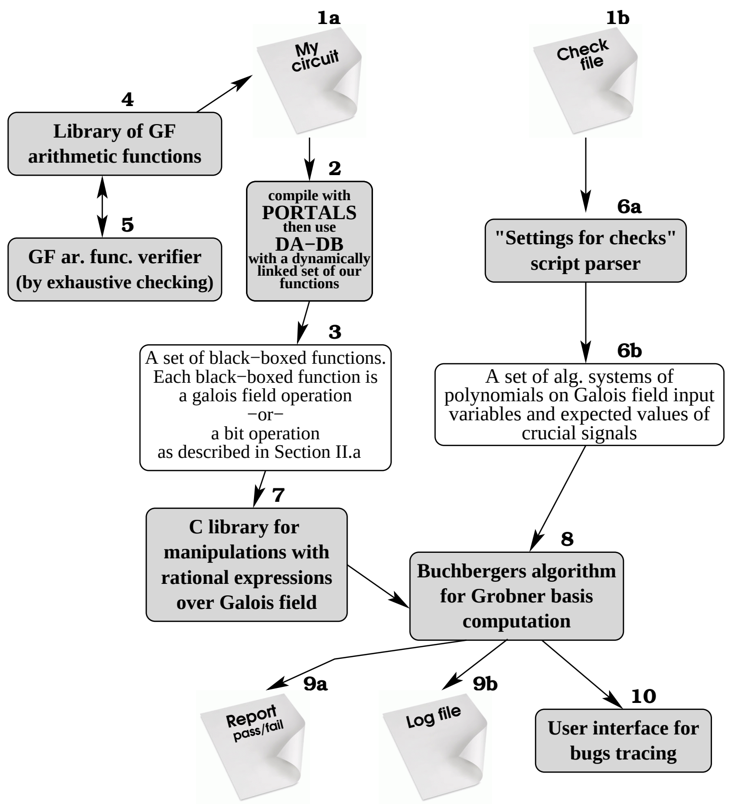
1a
My circuit
1b
Check file
4
Library of GF arithmetic functions
5
GF ar. func. verifier (by exhaustive checking)
2
compile with PORTALS then use DA−DB with a dynamically linked set of our functions
6a
"Settings for checks" script parser
3
A set of black−boxed functions. Each black−boxed function is a galois field operation −or− a bit operation as described in Section II.a
6b
A set of alg. systems of polynomials on Galois field input variables and expected values of crucial signals
7
C library for manipulations with rational expressions over Galois field
8
Buchbergers algorithm for Grobner basis computation
9a
Report pass/fail
9b
Log file
10
User interface for bugs tracing

Fig. 5 General schema of BLUEVERI tool.

The BLUEVERI analysis tool has its own custom input netlist format. A netlist translator was built as a DADB client to enable the tools flow from Portals into BLUEVERI.

The MESA logic simulator is a high performance cycle simulator used for functional verification within IBM. MESA simulation models are built from logic netlists in DADB by using model build clients.

The BLUEVERI code is written in C. For the computation of Gröbner bases we use "SINGULAR" [11] a powerful program for algebraic geometry computations distributed under general public license. BLUEVERI runs SINGULAR as a child process and uses "expect.h", (a standard C library), for sending queries and receiving results from SINGULAR's Gröbner basis engine. The monomial ordering used in the Gröbner basis computation is graded reverse lexicographic ordering.

7 Future Research Directions

7.1 Converting Branching to Polynomial Operations

The current design of BLUEVERI requires that the predicates associated with essential branch points in the algorithm be decidable from the input constraints.

When this is not the case, a forking algorithm allows one to explore alternative paths automatically including additional constraint hypotheses. A possible alternative design would be to convert `IS_ZERO` operations into symbolic polynomials. This is only possible if we restrict ourselves to operating over a fixed finite field $GF(2^k)$ (recall that currently BLUEVERI provides the option of making this restriction). Once k is fixed, since we know that every element of the field satisfies $x^{2^k} = x$, we have that x^{2^k-1} is 1 if $x \neq 0$ and 0 otherwise. So we have:

$$\texttt{IS\_NONZERO}(x) \mapsto x^{2^k-1}$$

$$\texttt{IS\_ZERO}(x) \mapsto x^{2^k-1} + 1$$

We view the boolean values $\{0, 1\}$ as elements of our finite field $GF(2^k)$. The boolean operations `AND`, `OR`, `NOT`, `XOR` can also be described by polynomial operations:

$$\texttt{AND}(x, y) \mapsto xy$$

$$\texttt{NOT}(x) \mapsto x + 1$$

$$\texttt{OR}(x, y) \mapsto xy + x + y$$

$$\texttt{XOR}(x, y) \mapsto x + y$$

Given that `IS_ZERO` and boolean operations on the elements $\{0, 1\}$ can be described by polynomials, we can finally give a polynomial definition for the $\texttt{WHEN\_ELSE}(b, x, y)$ branching construct which yields x when $b = 1$ and y when $b = 0$.

$$\texttt{WHEN\_ELSE}(b, x, y) \mapsto bx + (1 + b)y = b(x + y) + y$$

Among the previous polynomial definitions, the most costly transformation is the `IS_ZERO` function which can greatly increase the size of expressions. The current mechanism in BLUEVERI was intended to help control the size of intermediate expressions, and clearly should be used in cases when the boolean predicates to `WHEN_ELSE` can be decided. However the tradeoff between using the forking mechanism and trying to directly encode boolean operations as polynomials should be explored. Particularly in situations where the degree k of the finite field is small, the polynomial version could be very effective.

7.2 Avoiding Gröbner Bases

The transformations from boolean and conditional operations described in the previous section are special cases of a more general phenomenon. Any purely combinatorial function which takes inputs from a finite field and produces an output in the same field can be described by a polynomial function of its inputs. With the changes described in the previous section for booleans and conditionals, BLUEVERI can generate a polynomial describing any such combinatorial circuit. If we assign the resulting polynomial to a new variable, and use a lexicographic ordering among variables, then as observed in [7] the set of equations we generate is automatically a Gröbner basis assuming we have no constraints on the inputs. If we have constrained inputs, then we first compute a Gröbner basis for the constraints. Then for each combinatorial block in the circuit, we produce polynomials representing each of its outputs as functions of its inputs. We choose a term ordering where each output variable is more main than any of its inputs. The set of polynomial equations generated in this fashion is automatically a Gröbner basis. Since we require that the input variables range over $GF(2^k)$, we have relations of the form $x^{2^k} - x$ for each input variable. This guarantees that our input constraints, and consequently all our output polynomials generate what is called a radical ideal. A radical ideal is an ideal with the property that if some power of an element is in the ideal, then the element itself is in the ideal. In this situation with very little work we can produce the generators for the radical ideal describing our circuit. We can defer the expression blowup described in the previous section by the use of new variables. For example if the input of the `IS_ZERO` function was a large expression, we would assign the input to a new variable V, and express the output of the `IS_ZERO function` in terms of the new symbol V, i.e. $V^{2^k-1} + 1$. In this way we are deferring the problem of taking large powers of multiterm polynomials. Now after generating the equations describing the circuit, if we have some output signal that we need verify is always 0 for any input satisfying our constraints, we need to verify that the output signal is contained in our ideal of equations. Since the generators form a Gröbner basis, this becomes the problem of reducing a polynomial modulo the set of generators. For generators coming from the circuit, reduction is the same as substitution for the output variable. For generators of the constraint ideal, reduction is a multivariate version of division with remainder. This reduction of the output signal modulo the generators of our ideal is the only computationally expensive step. If we can find an efficient way to implement this reduction operation, then we have a potentially efficient way to verify any combinatorial circuit operating on elements of a finite field.

8 Conclusions

In this article we presented a novel technique for designing and verifying circuits based on the mathematics of Galois fields. At the heart of our approach is the idea of exposing operations on Galois field directly to a verification layer (encapsulated in a tool called BLUEVERI) which leverages powerful techniques from algebraic geometry to reason about the properties of the abstract Galois field rational expressions generated in the circuit. Our circuits are specified using a subset of existing Hardware Description Languages and as such, remain fully synthesizable, an important attribute to reduce the possibility of human error in the design process.

We demonstrated the value of the ideas we proposed in the context of two problems representative of the type of situations encountered when designing error correcting codes. In both instances, we showed BLUEVERI can significantly outperform conventional bit-level formal verification. We outlined a successful application of the BLUEVERI system to prove correctness of a real production complex error correcting code implemented on a POWER microprocessor which otherwise could not be verified conclusively with traditional verification methods.

Acknowledgements The authors would like to thank Shmuel Winograd and Geert Janseen of IBM Research for insightful discussions to help shape the solution.

References

1. Meaney, P. J., Lastras-Montaño, L. A., Papazova, V. K., Stephens, E., Johnson, J. S., Alves, L. C., O'Connor, J. A. and Clarke, W. J., *IBM zEnterprise redundant array of independent memory subsystem* IBM Journal of Research and Development, Jan-Feb, Vol. 56, 2012.
2. Lastras-Montaño, L.A.; Meaney, P.J.; Stephens, E.; Trager, B.M.; O'Connor, J.; Alves, L.C., *A new class of array codes for memory storage*, Information Theory and Applications Workshop (ITA), 2011 , vol., no., pp. 1–10, 6–11 Feb. 2011
3. R. E. Bryant *Graph Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on Computers, vol. C-35, pp. 677-691, August 1986.
4. R. E. Bryant and Y-A. Chen *Verification of Arithmetic Functions with Binary Moment Diagrams*, Design Automation Conference 1995.
5. U. Kebschull and W. Rosentiel *Efficient graph-based computation and manipulation of functional decision diagrams*, European Conference on Design Automation, pp. 278–282, 1993.
6. S. Morioka, Y. Katayama and T. Yamane *Towards Efficient Verification of Arithmetic Algorithms over Galois Fields.* Proc. Computer Aided Verification 2001, vol. 2102, pp. 465–477.
7. Jinpeng Lv, Priyank Kalla and Florian Enescu *Verification of Composite Galois Field Multipliers over* $GF((2^m)^n)$ *Using Computer Algebra Techniques.* Proc. IEEE International High Level Design Validation and Test Workshop 2011, pp. 136–143.
8. David Cox, John Little, and Donald O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, 2010. ISBN: 0-387-35650-9.
9. Rudolf Lidl and Harald Niederreiter, *Finite Fields.* Encyclopedia of Mathematics and Its Applications, Volume 20. Cambridge University Press, 1997. ISBN: 0-521-39231-4.
10. Oliver Pretzel, *Error-Correcting Codes and Finite Fields.* Oxford Applied Mathematics and CS Series. Oxford University Press, 1992. ISBN: 0-198-59678-2.
11. W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, SINGULAR 3-1-3 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de (2011).
12. H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, *Scalable automated verification via expert-system guided transformations*, Formal Methods in Computer-Aided Design, 2004, pp. 159-173.
13. D. Sarwate and N. Shanbhag, *High-Speed Architectures for Reed-Solomon Decoders*, IEEE Trans. on VLSI Systems, Vol. 9, No. 5, pp. 641–655, Oct. 2001.