A Formal Approach to Debug Polynomial Datapath Designs

Bijan Alizadeh

School of Electrical and Computer Engineering, College of Engineering, University of Tehran, Tehran, Iran

Abstract - By increasing the complexity of digital systems, debugging of such systems has become a major economical issue. In this paper, we introduce a mutation-based debugging technique that allows us to efficiently locate and then correct bugs in datapath dominated applications such as in Digital Signal Processing (DSP) for multimedia applications and embedded systems. In order to evaluate the effectiveness of our approaches, we have applied the proposed debugging technique to several industrial designs. The experimental results show that the proposed debugging technique enables us to locate and correct even multiple bugs in a reasonable run time and memory usage.

I. INTRODUCTION

The growing market for datapath dominated applications such as DSP for multimedia applications and embedded systems requires suitable Computer-Aided Design (CAD) support for their design, verification and debug. Such designs implement a sequence of algebraic computations such as addition, subtraction and multiplication over bit-vectors so that they are usually modeled as systems of multivariate polynomials of finite degree. One way to synthesize such algorithmic/behavioral level descriptions of arithmetic datapaths into hardware is to make use of high level synthesis tools. However, to obtain optimized hardware, designers prefer to manually refine such algorithmic level descriptions to Register Transfer Level (RTL) models by adding more and more implementation details at succeeding steps. This manual process is error prone and the lack of powerful and automatic debugging tools at this level greatly reduces designer's productivity when repairing even very simple errors. It should be noted that repairing functional design bugs is a two-step process: 1) error diagnosis discussing how to determine the location of a suspected error and 2) error correction discussing how to fix the error.

To address these problems, we adapt software mutation-based debugging techniques [1-3] into polynomial datapath hardware as will be explained in Section IV. Fig. 1 shows the proposed debugging approach. It only requires buggy implementation and algorithmic level specification as systems of multivariate polynomial functions. Since the specification is a high level model of what should be implemented, we assume that the specification is fault free and consider it as a golden model. First of all, the equivalence between the implementation and specification is checked. If they are not equivalent, it means that there is a bug in the implementation to be fixed. To do so, the main idea is introducing changes into the implementation and then observing if the specification is capable of detecting these changes by checking the equivalence between two given models. It is considerably faster and more scalable than gate-level diagnosis [4-8] because errors are modeled at a higher level. Please note that the implementation is often modeled with fixed-size datapath architectures so that polynomial computations are carried out over n-bit integers where the size of the entire datapath is fixed by way of signal truncation. Hence, equivalence verification of two polynomial functions extracted from high level specification and the implementation must be carried out over finite word-length. For doing so, *Modular Horner Expansion Diagram* (*M-HED*) [9] as a canonical representation of polynomial functions over finite word-length, is used as shown in Fig. 1. The buggy implementation is changed by considering different mutants until the implementation equals to the specification.

This way, much more complicated polynomials can be quickly processed as experimentally shown in Section V. These results give us ways to use the proposed framework not only in verification and debugging but also in synthesis and optimization of large complicated arithmetic circuits. Hence, the main contribution of this paper is to propose a debugging technique based on the *M-HED* as a canonical representation of polynomial functions over finite word-length to automatically localize and then correct bugs when two given polynomial functions are not functionally equivalent.

The paper is organized as follows: Section II provides a brief review of related work. In Section III we briefly describe a canonical graph-based representation of polynomial functions over bit-vectors, i.e., *M-HED* [9], that is utilized for equivalence verification purposes. Section IV presents our proposed debugging technique to automatically locate and correct bugs when the hardware implementation and algorithmic level specification are not equivalent. In Section V we discuss experimental results and finally a brief conclusion and future work are shown in Section VI.

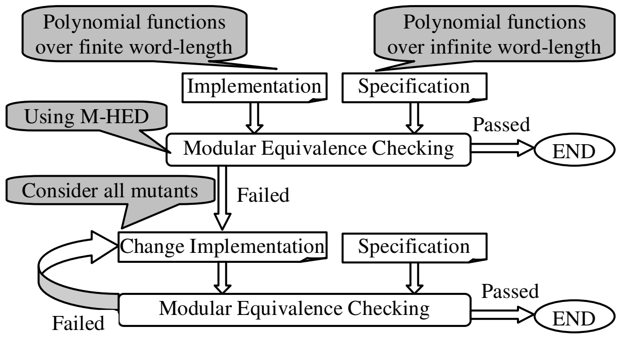
Polynomial functions over finite word-length
Polynomial functions over infinite word-length
Implementation
Specification
Using M-HED
Modular Equivalence Checking
Passed
END
Consider all mutants
Failed
Change Implementation
Specification
Modular Equivalence Checking
Passed
END
Failed

Fig. 1. Proposed debugging approach.

II. RELATED WORK

There is a large amount of literature on diagnosis and repair. However, most of it focuses on the gate level [4-8]. On the other hand, the lack of scalable and powerful RTL error diagnosis and correction system significantly reduces designer's productivity. To address this problem, several techniques have been proposed that work directly at the RTL. The techniques presented in [10-12] employ a software analysis approach that implicitly uses multiplexers (MUXes) to identify statements in the RTL code that are responsible for the errors. However, these techniques can return large potential error sites. One way to overcome this problem is to explicitly insert MUXes into the HDL code [13]. This approach makes use of hardware analysis techniques and greatly improves the accuracy of error diagnosis.

The authors of [14] present a debugging technique that uses formal analysis of an HDL description and failed properties to identify buggy statements. This technique can only be deployed in a formal property checking framework and cannot be applied in a formal equivalence verification flow common in the industry today. In addition, all techniques mentioned above cannot repair identified errors automatically. Finally, the work in [15] can diagnose and correct RTL design errors automatically, but it relies on state-transition analysis and hence, it does not scale beyond tens of state bits. There have been approaches based on SAT or SMT solvers to debug RTL designs using the erroneous traces [16, 17]. These methods depend on existence of such traces and also their corresponding correct results. In [16] a diagnosis method is presented which extends the SAT-based diagnosis for RTL designs. The design description as well as error candidate signals are specified in the word level and therefore word-level MUXes are added into error signals. Finally, to solve the resulting formula, a word-level solver is used.

This paper proposes a formal approach to debug datapath-intensive applications. In contrast to SAT-based debugging techniques that do not scale to computational expensive applications, our method is scalable and robust as experimentally proved in Section V. In addition, it can be integrated with other debugging techniques to extend the debugging potential offered by the conventional methods.

III. POLYNOMIAL EQUIVALENCE CHECKING

In this section, we briefly present our graph-based representation called *Modular Horner Expansion Diagram* (*M-HED*) [9] for functions with a mixed Boolean and integer domain and an integer range to represent arithmetic operations at a high level of abstraction, while other proposed Word Level Decision Diagrams (WLDDs) [18-20] are graph-based representations for functions with a Boolean domain and an integer range. Furthermore, the *M-HED* is able to deal with modular arithmetic computations over finite ring as discussed in the following subsections. Since the M-*HED* is an extension of another canonical representation, i.e., *HED*, we briefly explain the *HED* in subsection III-A. Then in subsection III-B, we discuss how the *M-HED* helps us to check the equivalence of two polynomials over finite word-length.

A. Horner Expansion Diagram (HED)

The *HED* is a binary graph-based representation which is able to represent polynomial function by factorizing variables recursively as shown in (1), where *const* is a term which is independent of variable *y*, while *linear* is another term which is served as the coefficient of variable *y* [21].

$$F(y,\ldots)=F(y=0,\ldots)+y\times[F'(y=0,\ldots)+\ldots]=const+y\times linear \quad (1)$$

Definition 1 (HED Representation): *HED is a directed acyclic graph G=(VR, ED) with vertex set VR and edge set ED. While the vertex set VR consists of two types of vertices: Constant (C) and Variable (V), the edge set indicates integer values as weight attribute. A Constant node v has as its attribute a value val(v)∈ Z. A Variable node v has as attributes an integer variable var(v) and two children const(v) and linear(v) ∈ {V, C}.*

According to the above definition, Fig. 2 depicts vertex v in the HED that denotes an integer function f^v defined recursively as follows:

- *If v ∈ C (is a Constant node), then f^v = val(v).*
- *If v ∈ V (is a Variable node), then f^v = const(v)+var(v)*linear(v).*

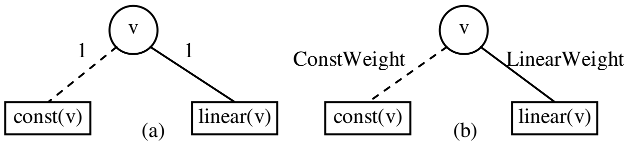

Fig. 2. Integer function representation in *HED* (a) before normalization and (b) after normalization.

Analogous to BDDs and *BMDs, the *HED* can be reduced by removing redundant nodes and merging isomorphic nodes. In order to further reduce the *HED* graph, the numeric values from the *Constant* nodes are moved to the related edges and taken into account as edge *weight*. As a result, *Constant* nodes 0 and 1 are only needed in the graph. Furthermore, the weights can be propagated to the upper edges. This procedure is called *normalization* (see Fig. 2(b)). To do so, any common factor between the weights assigned to the edges of *const* and *linear* portions, is extracted by taking the greatest common divisor (*gcd*) of the argument weights.

As a simple example, Fig. 3 illustrates how $f(x, y, z)=24-8z+12yz-6x^2-6x^2z$ is represented by the *HED*. Let the ordering of variables be x > y > z. First the decomposition w.r.t. variable *x* is taken into account. As shown in Fig. 3(a), after rewriting f(x, y, z) = (24-8z+12yz)+x(-6x-6xz) based on (1), *const* and *linear* parts will be 24-8z+12yz and -6x-6xz, respectively. The *linear* part is decomposed w.r.t. variable *x* again due to x^2 sub-monomial. After that, the decomposition is performed w.r.t. variable *y* and then *z* as shown in Fig. 3(b). In order to reduce the size of the *HED* representation, redundant nodes are removed and isomorphic sub-graphs are merged. In Fig. 3(b), 24-8z, 12z and -6-6z are rewritten by 8[3+z(-1)], 12[0+z(1)] and -6[1+z(1)], respectively. In order to normalize the weights, gcd(8,12)=4 and gcd(0,-6)=-6 are taken to extract common factors. Finally, Fig. 3(c) shows the normalized graph where gcd(4,-6) = 2 is taken to extract common factor between out-going edges from *x node*. In this representation, dashed and solid lines indicate *const* and *linear* parts, respectively. Note that in order to have a simpler graph; paths to 0-terminal have not been drawn in Fig. 3(c).

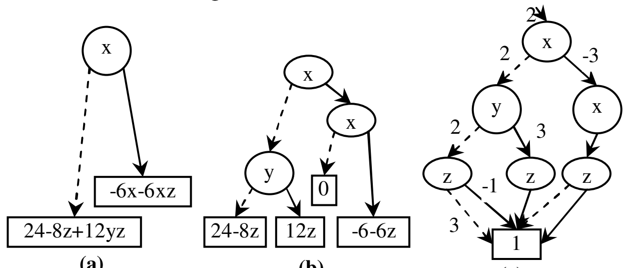

Fig. 3. HED representation of 24-8z+12yz-6x^2-6x^{2}z.

B. Modular Horner Expansion Diagram (M-HED)

In order to verify polynomial datapaths over bit-vectors, we have extended the *HED* to manipulate modular arithmetic [9]. To make this paper self-contained, we briefly review the main idea behind the *M-HED*. Although the equivalence verification over Z_m is known to be NP-hard when m≥2 [22], analyzing polynomials over arbitrary finite integer rings and their properties are useful to deal with the equivalence checking problem. The theory of univariate vanishing polynomials over Z_m, $m \in N$, $m>1$; i.e. those polynomials f such that $f(x) \bmod m \equiv 0$, has been presented in [23]. The authors of [24] have extended the concepts from [23] and derived a unique form representation of a multivariate polynomial over finite integer rings of the form Z_{p^n}, where p is any prime integer. Let us consider a simple example that defines functions $f_1[3:0] = 15(Y[3:0])^3 - 5(Y[3:0])^2 + 19Y[3:0] + 6$ and $f_2[3:0] = 7(Y[3:0])^3 + 3(Y[3:0])^2 + 3Y[3:0] + 6$. While f$_1$ is not equivalent to f$_2$ as polynomial functions over Z, they are equivalent over Z_{2^4}, i.e. $f_1 \bmod 2^4 \equiv f_2 \bmod 2^4$. Computing their difference over Z_{2^4} results in $f_1[3:0] - f_2[3:0] = 8(Y[3:0])^3 - 8(Y[3:0])^2 + 16Y[3:0]$. While the result is non-zero polynomial, $(8Y^3 - 8Y^2 + 16Y) \bmod 16 = 0, \forall Y \in \{0,1,...,15\}$ and we say $8Y^3$-$8Y^2$+$16Y$ vanishes in Z_{2^4}. In general it is not straightforward at all to see if given polynomials are vanishing ones or not. Actually a straightforward approach which expands everything into Boolean domain does not clearly work.

In [9] we have discussed how the properties of polynomial functions over finite integer ring [23, 24] allow us to reduce two polynomial functions to a canonical form based on the *HED* [21]. We follow the basic ideas shown in [25, 26] but make use of the *HED* for efficient implementations and manipulations of polynomials with fixed bit-width that results in a new package called *M-HED* [9, 27, 28]. Therefore, equivalent polynomial functions over finite integer rings, i.e., datapaths with finite bit-width in hardware designs, are automatically identified due to canonical representation of the *M-HED*. Since we only use the *M-HED* to check the equivalence between two polynomials over finite word-length and manipulating polynomials with fixed bit-width is not our contribution in this paper, we do not discuss it in more details and for more information please read our previous works [9, 27, 28].

IV. DEBUGGING APPROACH

This section addresses the problem of locating and correcting bugs when two given models are not equivalent. Formally, the debugging problem is characterized by a given specification (M_S) and an implementation (M_I). A specification must specify the correct behavior that maps an input state to the expected output state. We say that the implementation is correct iff it is equivalent to the given specification. Otherwise, we say that the implementation is buggy.

Mutation testing [1-3] was originally developed for constructing a set of tests which distinguish between a given software program and any nonequivalent program and for measuring the quality of test cases. The nonequivalent program is generated using the given one and applying the mutation transformation. In this paper, we make use of the idea of mutation testing for polynomial datapath debugging. In the following, we define polynomial mutation formally and give an algorithm for debugging. A valid mutation transformation (E_1, E_1') maps an expression E_1 to a different expression $E_1' \neq E_1$. We distinguish two types of mutations: 1) replacement of an arithmetic, logical or assignment operator by another operator from the same class, 2) eliminating/adding arithmetic, logical or assignment operator. Given an implementation M_I and a transformation (E_1, E_1'), we can define a mutant M_I' as an implementation obtained by replacing the expression $E_1 \in M_I$ with expression E_1'. We write (M_I, < (E_1, E_1') >) to represent the mutant M_I'. (M_I , < (E_1, E_1'), …, (E_n, E_n') >) denotes a sequence of n mutations of M_I where first (E_1, E_1') is applied, then (E_2, E_2') on the resulting mutant, and so on.

Definition 2 (Mutation Diagnosis): *Let M_I be an implementation and M_S be a specification for M_I. A mutant M_I' = (M_I , < (E_1, E_1'), …, (E_n, E_n') >) is a mutation diagnosis iff M_I' is functionally equivalent to M_S.*

Based on the above definition, it is obvious that one efficient way to debug M_I is to compute a *mutation diagnosis* of M_I. The following subsections explain how to compute all possible mutants and how to debug the implementation by using the *M-HED* to do modular equivalence checking, respectively.

A. Mutant computation

In order to compute all possible mutations for a given polynomial implementation, statements are selected one by one and the basic mutation functions are applied resulting in a new mutant. The selection process is finished when all statements have been taken into account. To compute mutants of single statements, the algorithm *AllMutations* illustrated in Fig. 4 is used which requires only the buggy implementation M_I. Mutations for assignments include changing the target variable (lines 3-5 of Fig. 4), and mutating the expressions (line 6). To compute mutants for a given expression, *E*, the algorithm replaces variables and constants by other variables, constants or even expressions (lines 6-12 of Fig. 4). Operators are to be changed by other operators (see for loop in lines 16-18), and expressions with an operator are reduced to one argument (lines 14-15). In addition, this algorithm allows for adding an additional operator to an existing one (lines 9-11).

```
AllMutations($M_I$)
1  M = an empty set;
2  FOR(all assignment statement S∈$M_I$ of the form X = E)
3    FOR (all variables Y ≠ X)
4       Add ($M_I$, < (X,Y) >) to M;
5    END FOR
6    IF(E is a variable or constant)
7       FOR(all variables Y ≠ E)
8          Add ($M_I$, < (E,Y) >) to M;
9          FOR(all variables X and operators ⊕)
10            Add ($M_I$, < (E,E⊕X) >) to M;
11         END FOR
12      END FOR
13   ELSE IF(E is of the form X ⊕ Y)
14      Add ($M_I$, < (E,X) >) to M;
15      Add ($M_I$, < (E,Y) >) to M;
16      FOR (all operators ⊕′ ≠ ⊕)
17         Add ($M_I$, < (E,X ⊕′ Y) >) to M;
18      END FOR
19   END IF
20 RETURN M;
```

Fig. 4. *AllMutations* procedure to compute all possible mutants.

B. Mutation-based debugging technique

As mentioned before, mutation diagnoses are mutants making the implementation M_I equal to the specification M_S. In other words, finding such mutants provide insightful suggestions for correcting diagnosed errors. *MutationDiagnosis* algorithm depicted in Fig. 5 can be used to compute such diagnoses.

The algorithm starts by computing all possible mutants (line 1 of Fig. 5) by calling *AllMutations* algorithm discussed in subsection IV-A. Then, for each mutant *U*, the modified implementation (M_I' in line 3) is checked against the specification M_S based on the equivalence checking technique explained in Section III. If they are equivalent (line 4), it means that the error is not only located but also corrected according to the modification defined by *U*. The algorithm finishes at this point by returning the modification specified by *U*. If, however, they are not equivalent, *U* is removed from the set of all possible mutants *B* (line 6), and other mutant is checked (lines 2-8). If all mutants have been processed and no bug is found, it means that our debugging technique is not able to locate such bugs (line 9 of Fig. 5).

Although it is obvious that the number of mutants increases exponentially when multiple errors need to be considered, in this paper we only apply the proposed technique to two simultaneous errors by checking all combinations of single errors and proposing a heuristic to reduce the number of mutants is left as a future work.

```
MutationDiagnosis($M_I$, $M_S$)
1  B = AllMutations($M_I$); //all possible mutants
2  FOR (all mutants U ∈ B)
3    $M_I'$ = ($M_I$, U);
4    IF ($M_I'$ == $M_S$)  //equivalence checking using M-HED
5       Bug is located and corrected! RETURN U;
6    ELSE     B = B – U;
7    END IF
8  END FOR
9  BUG is not locatable!
```

Fig. 5. *MutationDiagnosis* algorithm to automatically localize and correct bugs.

V. EXPERIMENTAL RESULTS

In order to demonstrate the effectiveness of the proposed debugging technique several benchmarks, Inverse fast Discrete Cosine Transform (IDCT), Finite Impulse Response (FIR16), Differential Equation solver (DIFFEQ), 64-point Fast Fourier Transform (FFT64) and Viterbi decoder with K=3,7,9 (Viterbi3, Viterbi7, Viterbi9) have been taken into account [27-29]. These designs come from a variety of problem domains such as digital signal processing, multimedia and communication. We obtained them using various approaches such as high level synthesis, C code versus C code and C code versus RTL code. The *M-HED* has been implemented in C++ and carried out on an Intel 2.0 GHz Core2 Due with 3 GB main memory running Linux.

In the first experiment, we have considered two test-cases FFT64 and Viterbi3. Table I tabulates their results for three cases:

1) When two given descriptions are equivalent and there is no bug (column *NoBug*). In this case, the proposed debugging technique is not applied because there is no bug in the implementation to be localized. Columns *Memory Usage* and *Run Time* in Table I indicate the memory usage and the run time required to check that the two models are equivalent.
2) Intentionally modifying the functionality of one intermediate node by converting addition/multiplication to subtraction and vice versa (column *Buggy1*).
3) Intentionally modifying the functionality of three intermediate nodes by converting

addition/multiplication to subtraction and vice versa (column *Buggy3*).

Column *benchmark* gives the benchmark's name, whereas column *#M_S* provides the number of lines in C code after unrolling all loops. In column *#M_I*, the number of lines in RTL code after synthesizing is reported. The column *#add/#sub/#mul* provides the number of additions, subtractions and multiplications required in each benchmark respectively and columns *#Nodes* and *#Mutants* give the number of *M-HED* nodes required to represent two descriptions and the number mutants to be checked, respectively. The memory usage and runtime required for equivalence checking of the two descriptions or for error diagnosis and correction are provided in columns *Memory Usage* in MByte and *Run Time* in seconds respectively. It should be noted that, in practice, engineers often find error diagnosis more difficult than error correction. It is common that engineers need to spend days or weeks finding the cause of a bug. However, once the bug is identified, fixing it may only take a few hours. In our technique the search space of error diagnosis and error correction are the same and that is why our technique enables engineers to localize design errors and achieve high quality corrections within minutes.

In another experiment, we intentionally modify the functionality of one (single error) or two (multiple errors) intermediate nodes by replacing/eliminating/adding arithmetic operations and also modifying the connections as discussed in Section IV. Then we try to locate bugs using our debugging technique. Table II provides debugging results of all benchmarks for two cases, 1) single error: "*single*" indexes and 2) multiple errors: "*multiple*" indexes. Please note that *MemUsage* and *CPUtime* columns show the memory usage and the run time to localize bugs using the *M-HED*. Although in the case of multiple errors, the proposed algorithm is trying to check all possible combinations of single corrections to correct the implementation, coming up with a heuristic to reduce the search space of multiple errors is left as a future work. Obviously, our debugging technique is able to localize bugs in a reasonable run time and memory usage even for large industrial designs such as Viterbi7 and Viterbi9.

In general, our experiments with industrial designs demonstrate that our debugging technique is efficient and scalable. In particular, designs up to 30,000 lines of code which is the size that an engineer actively works on (see Viterbi9 benchmark) can be diagnosed within minutes with high accuracy. These results show that our technique can be applied to more complex design than existing solutions can tackle.

TABLE I. Debugging results of two benchmarks using M-HED based debugging technique

Benchmark	**#M_S**	**#M_I**	**#add / #sub / #mul**	**#Nodes**	**#Mutants**	**Memory Usage (MB)**			**Run Time (s)**		
						NoBug	**Buggy1**	**Buggy3**	**NoBug**	**Buggy1**	**Buggy3**
FFT64	1412	1640	1026/1026/1512	11220	10692	10.8	11.2	11.3	3.5	4.6	5.1
Viterbi3	8357	16565	16474/192/0	52827	50018	36.3	36.4	38	57.8	60.4	63

TABLE II. Debugging results of all benchmarks (non-equivalent data-path computations)

Benchmark	**#M_S**	**#M_I**	**#add & #sub / #mul**	**#Nodes**	**#Mutants**	**MemUsage(MB)**	**CPUtime (s)**
IDCT$_{single}$	202	571	480 / 272	1430	2276	5.15	3.9
IDCT$_{multiple}$				1453	7401	7.11	12.0
FIR16$_{single}$	25	41	15 / 15	37	93	0.19	0.1
FIR16$_{multiple}$				49	348	0.22	0.5
DIFFEQ$_{single}$	35	77	40 / 50	150	277	3.63	12.0
DIFFEQ$_{multiple}$				156	816	3.86	28.0
FFT64$_{single}$	1412	1640	2052 / 1512	3368	10712	1.87	4.2
FFT64$_{multiple}$				3391	38990	2.12	13.0
Viterbi7$_{single}$	4885	9397	9674 / 0	16250	29042	6.72	20.0
Viterbi7$_{multiple}$				16266	106177	6.79	56.0
Viterbi9$_{single}$	23911	37810	47830 / 0	59149	143790	24.76	99.0
Viterbi9$_{multiple}$				59204	535158	33.11	213.0

VI. CONCLUSION AND FUTURE WORKS

In this paper we have introduced a formal debugging technique based on equivalence verification between the buggy implementation and the specification using a canonical decision diagram namely *M-HED* that supports modular polynomial computations over ring Z_{2^n}. This way, the *M-HED* not only supports equivalence verification of polynomials with multiple bit-width operands, but also helps us to localize probable bugs when two given polynomials are not equivalent.

One possible avenue for future work is to apply such a debugging technique to control dominated applications. We also plan to address the synthesis and optimization of polynomials over finite field, where the circuits to be optimized are collections of multipliers and adders. Since *HED*s are kept small for circuits having multipliers and adders, intensive circuit transformation and optimization can be carried out within practical time.

References

[1] V. Debroy, and W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs", in *Proc. ICST, 2010, pp. 65-74.*

[2] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, *vol. 32, no. 10, pp. 831-848, Oct. 2006.*

[3] A. S. Namin, J. H. Andrews, and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, *vol. 32, no. 8, pp. 608-624, August 2006*.

[4] K. Chang, I. Markov, and V. Bertacco, "Fixing Design Errors With Counterexamples and Resynthesis", *IEEE TCAD, vol. 27, no. 1, pp. 184-188, Jan. 2008.*

[5] J. C. Madre, O. Coudert, and J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM", in *Proc. ICCAD, 1989, pp. 30-33.*

[6] A. Smith, A. Veneris, and A. Viglas, "Design Diagnosis Using Boolean Satisfiability", in *Proc. ASPDAC, 2004, pp. 218-223.*

[7] A. Veneris, and I. N. Hajj, "Design Error Diagnosis and Correction via Test Vector Simulation", *IEEE TCAD, vol. 17, no. 12, pp. 1803-1816, Dec. 1999.*

[8] Y. S. Yang, S. Sinha, A. Veneris, and R. Brayton, "Automating Logic by Approximate SPFDs", in *Proc. ASPDAC, 2007, pp. 402-407.*

[9] B. Alizadeh, and M. Fujita, "Modular-HED: A canonical decision diagram for modular equivalence verification of polynomial functions", in *CFV, 2008, pp. 22-40.*

[10] T. Y. Jiang, C. N. J. Liu, and J. Y. Jou, "Estimating likelihood of Correctness for Error Candidates to Assist Debugging Faulty HDL Designs", in *Proc. ISCAS, 2005, pp. 5682-5685.*

[11] J. C. Rau, Y. Y. Chang, and C. H. Lin, "An Efficient Mechanism for Debugging RTL Description", in *Proc. IWSOC, 2003, pp. 370-373.*

[12] C. H. Shi, and J. Y. Jou, "An Efficient Approach for Error Diagnosis in HDL Design", in *Proc. ISCAS, 2003, pp. 732-735.*

[13] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic Fault Localization for Property Checking", *IEEE TCAD, vol. 27, no. 6, pp. 1138-1149, June 2008.*

[14] R. Bloem, and F. Wotawa, "Verification and Fault Localization for VHDL Programs", *TIV, vol. 2, pp. 30-33, 2002.*

[15] S. Staber, B. Jobstmann, and R. Bloem, "Finding and Fixing Bugs", *Springer-Verlag LNCS 3725, 2005, pp.35-49.*

[16] S. Mirzaeian, F. Zheng, and K.-T. Cheng, "RTL Error Diagnosis Using a Word Level SAT-Solver", in *Proc. ITC, 2008, pp. 1-8.*

[17] H. Mangassarian, A. Veneris, S. Safarpour , M. Benedetti and D. Smith, "A Performance-Driven QBF-Based Iterative Logic Array Representation with Applications to Verification, Debug and Test", in *Proc. ICCAD, pp. 240-245, 2007.*

[18] S. Horeth, and R. Drechsler, "Formal verification of word-level specifications", in Proceedings of Design Automation and Test in Europe (DATE), pp. 52-58, 1999.

[19] R. Drechsler, B. Becker, and S. Ruppertz, :The K*BMD: A verification data structure", in IEEE Design and Test, pp. 51-59, 1997.

[20] B. Becker, R. Drechsler, and R. Enders, "On the representational power of bit-level and word-level decision diagrams", in Proceedings of Asia and South Pacific-Design Automation Conference (ASP-DAC), pp. 461-467, 1997.

[21] B. Alizadeh, and M. Fujita, "A canonical and compact hybrid word-Boolean representation as a formal model for hardware/software co-designs", in *CFV, 2007, pp. 15-29.*

[22] O.H. Ibarra, and S. Moran, "probabilistic algorithms for deciding equivalence of straight-line programs", in *Journal of the Association for Computing Machinery, vol. 30, pp. 217-228, 1983.*

[23] D. Singmaster, "on polynomial functions (mod m)", in *Journal Number Theory, vol. 6, pp. 345-352, 1974.*

[24] N. Hungerbuhler, and E. Specker, "a generalization of the Smarandache function to several variables", in *Electronic Journal of Combinatorial Number Theory, vol. 6, 2006.*

[25] N. Shekhar, P. Kalla, and F. Enescu, "equivalence verification of polynomial datapaths with multiple word-length operands" in Proceedings of the conference on Design, Automation and Test in Europe (DATE), pp. 824-829, 2006.

[26] N. Shekhar, P. Kalla, and F. Enescu, "equivalence verification of polynomial datapaths using ideal membership testing", in IEEE Transactions on Computer-aided-design (TCAD), vol. 26, no. 7, pp. 1320-1330, 2007.

[27] B. Alizadeh, and M. Fujita, "A Unified Framework for Equivalence Verification of Datapath Oriented Applications", in IEICE Transactions on Information and Systems (IEICE TRANS. INF. & SYST.), Vol E92-D, No. 5, pp. 985-994, May 2009.

[28] B. Alizadeh, and M. Fujita, "Modular Datapath Optimization and Verification based on Modular-HED", in Transactions on Computer-aided design (TCAD), vol. 29, no. 9, pp. 1422-1435, September 2010.

[29] B. Alizadeh, and M. Fujita, "Automatic merge-point detection for sequential equivalence checking of system-level and RTL descriptions", in International Symposium on Automated Technology for Verification and Analysis (ATVA), LNCS 4762, pp. 129-144, 2007.