2012 17th IEEE European Test Symposium (ETS)

Combining Dynamic Slicing and Mutation Operators for ESL Correction

Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik, Raimund Ubar
Department of Computer Engineering,
Tallinn University of Technology, Estonia
{urmas|hanno|maksim|jaan|raiub}@pld.ttu.ee

Giuseppe Di Guglielmo, Graziano Pravadelli, Franco Fummi
Department of Computer Science,
University of Verona, Italy
Email: {name.surname}@univr.it

Abstract

Verification is increasingly becoming the bottleneck in designing digital systems. In fact, most of the verification cycle is not spent on detecting the occurrences of errors but on debugging, consisting of locating and correcting the errors. However, automated design-error debug, especially at the system-level, has received far less attention than error detection. Current paper presents an automated approach to correcting system-level designs. We propose dynamic-slicing and location-ranking-based method for accurately pinpointing the error locations combined with a dedicated set of mutation operators for automatically proposing corrections to the errors. In order to validate the approach, experiments on the Siemens benchmark set have been carried out. The experiments show that the proposed method is able to correct three times more errors compared to the state-of-the-art mutation-based correction methods while examining fewer mutants.

1 Introduction

Increasing design costs are the main challenge facing the semiconductor community today. In particular, assuring the correctness of the electronic design, since the early stages of the design cycle, contributes to a major part of the problem [1]. However, *localization* and *correction* of design errors, i.e., *debug*, has received far less attention than *error detection*, both, in terms of research works and industrial tools introduced [2]. As a consequence, in industrial practice, debug is still a human-based activity that affects time-to-market [3].

The debugging approaches proposed in the past for logic and register-transfer (RT) levels, e.g. [16][17][18], cannot be applied for designs at electronic-system level (ESL) or their feedbacks are not sufficiently readable for this abstraction level. At ESL, designs are described in an algorithmic way with a high level of abstraction with respect to the final hardware implementation [1].

In this context, more effective methods for automating ESL debug are highly requested. Consider the case where a designer has a bug-affected ESL implementation. That is, an erroneous behavior of the implementation, with respect to the expected functionality, has been already (automatically) detected. Automated debug of errors consists of two steps: error localization and error correction. Error localization identifies the portion of the design responsible for the erroneous behavior, while error correction is responsible for locally modifying the functionality of the identified portion.

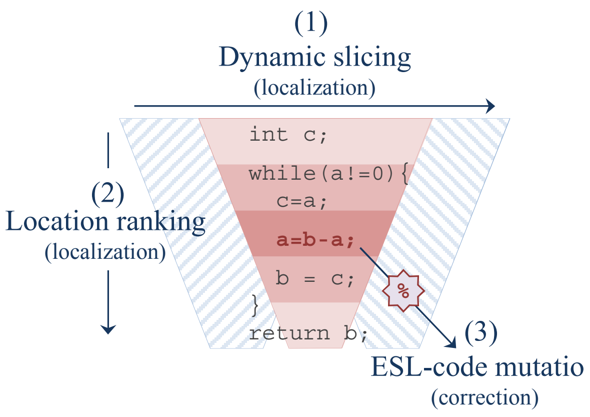


Figure 1. The proposed automated debugging approach for ESL designs relies on dynamic slicing and location ranking for error localization and code mutation for error correction.

For error localization, simulation-based [4][5][6][7][8][20] and formal approaches [9][10] are known. It is widely accepted that simulation-based techniques scale well with the design sizes, but are not exhaustive; while, formal techniques provide a high grade of confidence in the results, but are susceptible of the design complexity.

For error correction, error matching [12][13] and re-synthesis [14][15][16][17][18] have been investigated in literature. In particular, re-synthesis provides a correction which is represented as a partial truth table based on the stimuli under consideration. This kind of correction is not readable and cannot be easily understood and verified by the design engineer. Moreover, the resynthesized erroneous portion of the design is likely to fail when new stimuli will be added to the suite.

Recently, a more effective approach has been proposed for software debugging, which relies on simulation-based localization and error-matching correction [19]. The authors apply a diagnosis tool, i.e., Tarantula [20], for calculating suspiciousness scores for design portions and exploit mutation-based techniques to repair C and Java applications. However, the methodology is still affected by application size (in terms of number of mutations per lines of code) and targets mainly the control flow of software applications.

In this paper, we propose an approach, which, as opposed to [19], implies a dynamic slicing based methodology for error localization combined with a dedicated set of mutation operators for automated correction of system-level design errors. Different from [19] we consider corrections also in the data part of the design. However, the number of mutations required by the presented method is still smaller than in [19] due to more accurate localization achieved by dynamic slicing versus considering executed sentences applied in the Tarantula tool [20].

Figure 1 provides an overview of the proposed approach, which is based on three main iterative phases: *dynamic slicing*, *location ranking*, and *ESL-code mutation*.

The dynamic slicing reduces the debugging analysis to all the statements that actually affect the (erroneous) value of an output for a given input. In particular, it provides the relevant subset of the design statements where the location-ranking phase has to investigate for the possible cause of the error. As shown in Figure 1, a design may be thought of as a collection of "threads", each computing the value of an output. The dynamic slicing isolates the threads computing the erroneous outputs for the given inputs, i.e., the erroneous slice.

Then, the location ranking assigns a score to each statement in the erroneous slices. This score is the ratio between the number of statements executed during runs resulting in erroneous and correct behaviors. Intuitively, if a statement occurs very frequently in erroneous executions, it very likely contains a bug. In Figure 1, a more intense hue is associated with highly scored statements.

Finally, the ESL-code mutation addresses the correction of statements highly ranked as sources of the erroneous behavior. Typically, mutation techniques are used in testing for modeling realistic faults on correct design; in the adopted error-correction methodology, the mutation of a bug-affected design may result in a realistic correction.

A final observation is necessary on the golden model, i.e., the reference behavior of the ESL design during the debug phase. In the adopted simulation-based approach, the golden model may be indifferently (i) industrial test cases, i.e., input stimuli and expected results, specified by module designers, (ii) input stimuli and assertions supplied with the design or (iii) results obtained by abstract or different reference, e.g., executable-UML models and alternative implementations.
The main contributions of the present work are:

- the development of an accurate-localization methodology of bugs based on location ranking and dynamic slicing that scale well with design sizes;
- the definition of a dedicated set of mutation operators to model realistic errors in ESL designs;
- the development of a readable error-matching-based correction for system-level designs.

As experiments show, our approach provides three times higher success rate in terms of fixed errors in comparison to the results published in [10] and [19], while examining significantly fewer mutants.

The rest of the paper is organized as follows. Section 2 provides an overview of the related works. Section 3 describes the methodology for design-error localization based on dynamic slicing. Section 4 introduces the mutation-based-correction methodology for system-level design errors. Section 5 provides experimental results validating the ESL-repair approach on the realistic buggy code versions represented in the Siemens benchmark suite [10]. Finally, conclusions are drawn in Section 6.

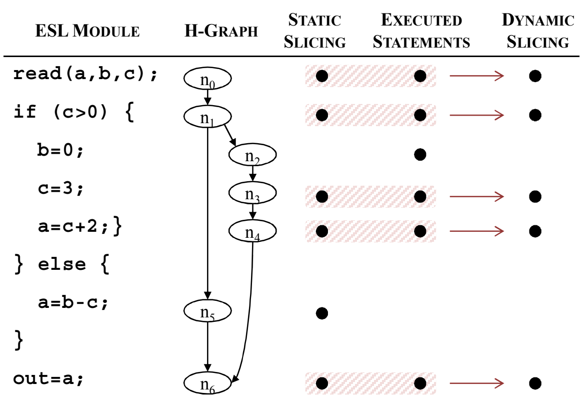
2 Related works

In debugging, the error localization is considered the most time expensive activity and its quality affects the following (manual or automatic) correction phase [21]. In manual error localization, engineers run the design with some input stimuli till they observe a failure; then, they iteratively place breakpoints, analyze the system status, and backtrack to the error origin using a source-level debugger, e.g., GNU GDB [22].

On the other hand, automatic error localization is based on different methodologies. In particular, they may be simulation-based and use coverage information [6][7][20], binary search [8], and statistical analysis [4][5]. As well, formal approaches for error localization exist that are very effective but may suffer the state-explosion of the underlying solver [9][10]. Of all these solutions, the Tarantula [20] coverage-based approach has been proven suitable for real-world designs. In the present work, we provide an improvement for error localization, which significantly reduces the overhead of the error-correction phase based on ESL-code mutation.

After an error is detected and localized, it should be corrected. Design-error correction for combinational circuits has been thoroughly studied for decades. There exist, both, error-matching-based [12][10][13] and re-synthesis [14] approaches. There have also been attempts to generalize the above methods for design-error correction of sequential circuits [14][15]. In particular, the SAT-based correction and re-synthesis approach developed by Smith et al. [16] has been extended to higher abstraction levels such as register-transfer level [17][18]. The re-synthesis approach for high-level design-error correction has two main limitations. The correction is *not readable* and thus cannot be checked by the designer. Moreover, the *correction is limited* to the set of used stimuli: this is due to the logic optimization freedom created by the partial truth table of the portion to be repaired.

Finally, in [10] a symbolic-simulation-based approach is proposed for both error correction and localization in ESL designs described as C programs. All the reasoning is done with a Satisfiability Modulo Theory (SMT) solver [11], thus it can be classified as a formal method. In particular, the approach performs the error correction by using approximation heuristics and a template-based methodology, which gives readable corrections. In the experimental-result section, we provide



```
read(a,b,c);
if (c>0) {
  b=0;
  c=3;
  a=c+2;}
} else {
  a=b-c;
}
out=a;
```

Figure 2. The ESL description is modeled as a flowgraph, i.e., hammock graph. Simulation and slicing are performed on the model representation.

comparisons of our approach and [10], showing better correction capability and preserving correction readability.

3 Dynamic Slicing for Error Localization

This section describes the proposed error localization methodology, which is based on dynamic slicing. In particular, Section 3.1 describes the adopted model for ESL descriptions; Section 3.2 summarizes the dynamic slicing methodology; finally, Section 3.3 describes the methodology for ranking the error location. The accurate localization of errors permits to significantly reduce the overhead of the following mutation-based correction phase.

3.1 Flowgraph based modeling of the algorithmic level

In this paper, we consider automated debugging of ESL designs. In order to formally represent the ESL algorithmic descriptions we have chosen the flowgraph model as an underlying model. In such flowgraph, there is a one-to-one correspondence between the program statements and nodes and edges represent the control flow of the program. More precisely, the model representation is a special case of flowgraph known as the *hammock graph* [25], which was proposed for program slicing in [26]. Hammock graph is defined as follows.

Definition 1: A *hammock graph* is a structure $H=<N, E, n_0, n_e>$, where N is a set of nodes, E is a set of edges in $N\times N$, n_0 is the *initial node* and n_e is the *end node*. If (n, m) is in E then n is an *immediate predecessor* of m and m is an *immediate successor* of n. A *path* from a node n_1 to a node n_2 is a list of nodes $p_0, p_1, ..., p_k$ such that $p_0 = n_1$, $p_k = n_2$, and for all i, $1 \leq i \leq k - 1$, (p_i, p_{i+1}) is in E. There is a path from n_0 to all other nodes in N. From all nodes of N, excluding n_e, there is a path to n_e.

Figure 2 presents a simple ESL functionality in C language, i.e., column ESL MODULE, and the corresponding flowgraph H, i.e., column H-GRAPH. In the following, we introduce some definitions in order to explain the slicing process on flowgraph structures.

3.2 Model slicing

We apply dynamic slicing in order to narrow the search of the causes of design errors in algorithmic descriptions. In this Section, we provide a brief introduction to slicing techniques and explain how we implement dynamic slicing in design error localization.

Program slicing [26] is a technique for extracting portions of a program affecting a selected set of variables of interest. By focusing on the computation of only few variables the slicing process can be used to discard portions of the program, which cannot influence these variables, thereby reducing the size of the program. The reduced program is called a slice.

Slices reproduce a projection from the behavior of the initial program. This projection represents the values of certain variables as seen at certain statements.

Definition 2: A *slicing criterion* of a program P is a tuple (x, V), where x is a statement in P and V is a subset of the variables in P.

Informally, given a slicing criterion $C = (x, V)$, a static program slice S consists of all statements in program P that may affect the value of $v \in V$ for a set of all possible inputs at the point of interest, i.e., at the statement x. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data and control dependencies. Unfortunately, the size of the slices so defined may approach that of the original program. Indeed, static slicing preserves the behavior of the original program for *all the possible* input values. In this case, the usefulness of the slices in debugging tends to diminish as the size of the slices increases.

In [27], Korel and Laski introduced a more accurate slicing technique, i.e., *dynamic slicing*. Dynamic slicing provides more narrow slices, preserving the behavior of the original program and consisting of only the statements that influence the value of a variable for *a given input*.

Figure 2 illustrates the concepts of static and dynamic slicing applied to the flowgraph representation of a ESL functionality. In particular, the Figure reports an intuitive correlation between static slicing, execution trace, and dynamic slicing. Let us consider, for example, the slicing criterion $C = (n_6, \{out\})$. In this case the n_6 is the end node n_e of the hammock graph. The black dots in the column STATIC SLICING indicate the statements included into the slice in case of static slicing. These mark the statements that are needed in order to calculate the value of variable a at node n_6. As we can see the node n_2 is excluded from the slice because the statement b=0 is not necessary for calculating the value of variable *out* at node n_6.

The column DYNAMIC SLICING refines that slice according to the execution trace obtained with actual value assignments. Assuming that variables get assignments a=2, b=4 and c=7, we obtain the slice shown in the last column of Figure 2. The *else* branch of the condition is not activated by these input values and therefore the respective statement are not included into the slice. The column EXECUTED STATEMENTS shows all the statements that were executed in current trace with the given input assignments. As we can see, the statements occurring in the dynamically-computed slice are a proper subset of the statements in the statically-computed slice and execution trace. This narrows the search space of the following step for ranking the error locations.

3.3 Slicing-based error localization

In current paper, we consider a design-error-localization approach, where ESL implementations fail on some of the given test cases. The error localization relies on error detection results. The mechanisms of the latter are out of scope of this paper and may involve for instance the golden output responses specified by the test cases, assertions supplied with the test environment or results obtained from a analyzing the specification (e.g. UML, SW program, etc.).

The error localization proposed in current paper is based on calculating the dynamic slices for all the observable outputs of the system with all the test cases. Depending on whether an output response obtained by a given slice is correct or not, the slice is marked as a passed or failed one, respectively. Then, a statistical and coverage-based approach is implemented assigning score to flowgraph nodes based on the number of times they were included into failed slices with respect to the number of times they occur in the previous executions. Finally, the flowgraph nodes are ranked according to this score, referred to as the *suspiciousness score*.

In details, the error ranking and localization takes place as follows. Let T be a test suite consisting of test cases t_i for verifying the functionality of the ESL description. Let H be the flowgraph associated with the description. Let y_j be the observable output variables of the design. Finally, let the nodes n_j of H be the respective nodes were value assignments to y_j are made. Over each test case t_i and, in turn, over each observable output variable y_j we generate a dynamic slice d_{ij} according to the values of current test case t_i and a slicing criterion $C = (x_j, \{y_j\})$, where x_j is the statement at the flowgraph node n_j.

If y_j resulted in a correct value at test case t_i, then the dynamic slice d_{ij} is included into the set of passed slices D_{PASSED}. Otherwise, it is included to the failed slices, i.e. $d_{ij} \in D_{\text{FAILED}}$. Each node n_k of flowgraph H gets a score according to the number of times c_{FAILED} it is included into the set of failed slices D_{FAILED} and the number of times c_{PASSED} it is included into the set of passed ones, i.e. D_{PASSED}. This score of suspiciousness is calculated as follows:

$$suspiciousness(n_k) = \frac{c_{FAILED}}{c_{FAILED} + c_{PASSED}}$$

The nodes n_k are ranked according to the suspiciousness score with more probable candidates for error correction having higher score values. This ranking is used for selecting statements to be corrected by the mutation-based methodology presented in the following section.

Table 1. List of mutation operators for correction

MUTATION OPERATOR	C OPERATORS/EXAMPLES
AOR (arithmetic operator replacement)	+, -, *, /, %
ROR (relational operator replacement)	==, !=, >, <, >=, <=
LCR (logical connector replacement)	&&, ||
ASOR (assignment operator replacement)	+=, -=, *=, /=, %=, =
UOR (unary operator replacement)	+, -, ~, !
Bitwise operator replacement	<<, >>, &, |, ^
Bitwise assignment operator replacement	<<=, >>=, &=, |=, ^=
Increment/decrement operator replacement	*x*++, ++*x*, *x*--, --*x*
Number mutation (decimal digit replacement in integers, floats and array indexes)	0...9
Constant replacement unary minus/ unary plus/ zero	+*C*, 0, -*C*

4 Mutation-Based Error Correction

Mutation is a process, where syntactically-correct functional changes are inserted into the program [28]. Traditionally mutations are performed by perturbing the behavior of the program in order to see if the test suite is able to detect the difference between the original program and the mutated versions. The effectiveness of the test suite is then measured by computing the percentage of detected, or *killed*, mutations.

In this paper, we apply mutation operators for correcting erroneous circuits. The goal is to develop an error-matching based correction approach, which would be capable of modeling realistic design errors. Moreover, it is crucial to select a limited number of mutation operators, because the perturbation and simulation of erroneous design implementations with a large number of error locations and mutant operators would become prohibitively time-consuming.

Table 1 presents the set of ESL-mutation operators which were implemented in the error-matching based correction method developed in this paper. Since we target ESL descriptions in C language, we only focus on algorithmic aspects of the description and do not consider software-specific constructs and related errors, such as dynamic-memory allocation, pointer arithmetic, and file

Table 2. Characteristics of the Siemens benchmarks.

DESIGN	LOC	TEST-CASE #	FAULTY-VERSION #
replace	507	5542	32
schedule	397	2650	9
schedule2	299	2710	9
tcas	174	1608	41
tot_info	398	1052	23
print_tokens	539	4130	7
print_tokens2	489	4115	10

I/O. This permits to reduce the overhead of the code-mutation phase and address only system-level issues.

In particular, the mutation operators include replacement of C language operators, which have been divided into several groups: arithmetic operators, relational operators, assignment operators, unary operators, etc. In addition, number mutations are performed by replacing each decimal digit in the numeric values one-by-one with other decimal values. This includes both, integer and floating point numbers and it covers also the array indexes. Also, constants are mutated by inserting unary operators + and – as well as replaced by zero.

Figure 3 explains the mutation-based correction process. Subsequent to the error location step described in Section 3, which ranks the statements of the program, the suspected error locations are iteratively tried according to their rank. The operators in the statements are, in turn, iteratively substituted by mutation operators, i.e., valid operators from the same category. In other words, replacing arithmetic operators by arithmetic operators, relational operators by relational ones etc. These iterations stop when the simulation result confirms that the mutated program provides output responses equal to the golden output responses, in other words, a correction has been found. Otherwise the process continues until there exist untried error locations and/or mutant operators, or when a user-specified time limit is reached.

The mutation-based correction method proposed in this paper is an error-matching approach. Error-matching is known to have the limitation that it is generally not capable of fixing errors that are not included to the model. On the other hand, the mutation-based error-matching provides easy-to-read corrections of system-level descriptions. Moreover, our experiments show that the mutation-based approach can fix some of the not modeled errors by proposing alternative but equivalent fixes.

5 Experimental Results

The proposed debugging approach has been implemented as a module of a larger tool, i.e., FoREnSiC [24], which also features formal and semi-formal approaches for debugging of ESL design [10]. Our framework supports debugging of algorithmic descriptions of hardware in C language. In order to evaluate the proposed method, experiments on Siemens

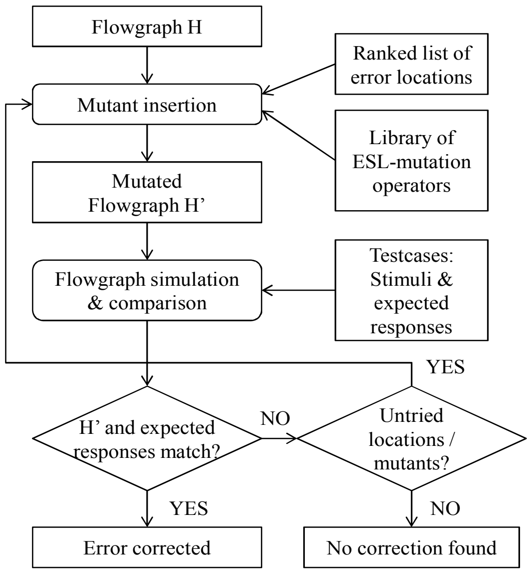

Flowgraph H
Ranked list of error locations
Mutant insertion
Library of ESL-mutation operators
Mutated Flowgraph H'
Flowgraph simulation & comparison
Testcases: Stimuli & expected responses
YES
H' and expected responses match?
NO
Untried locations / mutants?
YES
NO
Error corrected
No correction found

Figure 3. The mutation-based error correction flow.

benchmark suite [23] comparing it to a recently published formal [10] and dynamic [19] technique were carried out. We applied the front-end of FoREnSiC for generating the flowgraph models for the C language designs.

In Table 2, the main characteristics of the benchmark circuits are presented. Column LOC shows the number of lines of code for the corresponding C designs; column TEST-CASE # shows the number of test cases for the design, which include both failing test stimuli and passing stimuli; finally, column FAULTY-VERSION # shows the number of faulty versions of the benchmark programs. We excluded one faulty version from benchmark *schedule2* because the design error did not result in any test case failure making the correction process meaningless.

In Table 3, the results of the design error correction experiments are presented. Current method is compared to two recently published methods: a symbolic-simulation-based method [10] and a mutation-based method [19]. For each methodology, columns # FIXED show the number of corrected faulty model versions and Columns % FIXED show the percentage of corrected models from the total number of faulty model versions.

As it can be seen from the table, the proposed approach clearly outperforms [10], where only 8 faulty versions (out of 41) of *tcas* design are analyzed. The approach in [10] is able to correct 7 out of these 8 faulty versions, whereas our approach corrects all 8. Furthermore, due to the underlying solver, the formal approach [10] is only able to model the designs whose bit-width is reduced from 32 to 8 bits.

With respect to [19], current method increases the percentage of successful corrections from 16.0% to 50.3%. Thus, the rate of corrections is increased by the factor of three.

It is important to stress that the increase in successful fixes does not come at the expense of more mutants to be

Table 3. Design error repair experiments.

Design	FMCAD'11 [10]		STVV'10 [19]		Current method		
	# fixed	% fixed	# fixed	% fixed	# fixed	% fixed	mutants examined
replace	-	-	3	9.4	**12**	**37.5**	855.2
schedule	-	-	0	0.0	**2**	**22.2**	188.0
schedule2	-	-	1	11.1	**3**	**33.3**	460.7
tcas	7	17.1	9	22.0	**26**	**63.4**	131.1
tot_info	-	-	8	34.8	**15**	**65.2**	781.3
print_tokens	-	-	0	0.0	**1**	**14.3**	825.0
print_tokens2	-	-	0	0.0	**7**	**70.0**	952.3
Total:		*N/A*		*16.0*		***50.4***	*599.1*

considered. The last column of Table 3 shows the localization accuracy in terms of the average number of examined mutants per design error. In fact, this number is 599.1, which is even slightly less than 642 mutants in average obtained in [19].

The significant increase in successful corrections with respect to [19] is due to the selection of mutation operators, which are not limited to control flow errors. The run-time advantages in terms of the number of mutants examined comes partly from the more accurate diagnosis method based on dynamic slicing and location ranking.

6 Conclusions

The paper presents a method for correcting design errors in algorithmic descriptions of system-level hardware. The method applies dynamic slicing and location ranking to accurately pinpoint the error locations and combines it with a dedicated set of ESL-mutation operators for automatically proposing fixes to the errors. In order to validate the approach, experiments on the Siemens benchmarks have been carried out. The experiments show that the proposed method is able to repair three times more errors than previously achievable by mutation-based repair while examining fewer mutants. In addition, the method clearly outperforms a recent formal correction approach.

Acknowledgements

The work has been partly supported by European Commission FP7-ICT-2009-4-248613 DIAMOND project, by Research Centre CEBE funded by European Union through the European Structural Funds, by Estonian Science Foundation grants 9429 and 8478 and by Estonian Academy of Security Sciences.

References

[1] B. Bailey, G. E. Martin, and A. Piziali, "ESL design and verification: a prescription for electronic system-level methodology," Morgan Kaufmann Publisher, 2007.

[2] F. Rogin and R. Drechsler, "Debugging at the electronic system level," Springer Verlag, 2010.

[3] R. S. Pressman, "Software Engineering – A Practitioner's Approach," McGraw Hill, 1992.

[4] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15-26, 2005.

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

[6] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, vol.83, no.2, pp.188-208, 2010.

[7] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 4, pp. 573-597, 2009.

[8] H. Cleve and A. Zeller, "Locating causes of program failures," *Proc. of Int. Conf. on Software Engineering*, pp. 342-351, 2005.

[9] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," *Proc. of Conference on Correct Hardware Design and Verification Methods*, pp. 35-49, 2005.

[10] R. Konighofer and R. Bloem, "Automated error localization and correction for imperative programs," *Proc. of Formal Methods in Computer Aided Design*, pp. 91-100, 2011.

[11] L. De Moura and N, Bjorner, "Satisfiability modulo theories: An appetizer," *Formal Methods: Foundations and Applications*, pp. 23-36, 2009.

[12] J. C. Madre, O. Coudert, and J. P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM," *Proc. of International Conference on CAD*, 1989, pp. 30-33.

[13] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic design verification via test generation," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 1, 1988.

[14] Ali, M. F., Safarpour, S., Veneris, A., Abadir, M. S., and Drechsler, R. Post-verification debugging of hierarchical designs. *In Proceedings of ICCAD*, pp. 871-876. 2005.

[15] A. Wahba, D. Borrione. Design error diagnosis in sequential circuits. *Lecture Notes In Computer Science*; Vol. 987, pp. 171 – 188, Springer, 1995.

[16] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability", *Proc. Asia and South Pacific Design Automation Conference (ASPDAC)*, 2004, pp. 218-223.

[17] Kai-hui Chang, et al., "Automatic Error Diagnosis and Correction for RTL Designs", *Proc. High-Level Design and Validation Workshop (HLDVT)*, Irvine, CA, November 2007.

[18] Kai-hui Chang; Markov, I.L.; Bertacco, V.; , "Fixing Design Errors With Counterexamples and Resynthesis," *IEEE Trans. on CAD of ICs and Systems,* , vol.27, no.1, pp.184-188, Jan. 2008

[19] V. Debroy, W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs", *Proc. of Int. Conf. on Software Testing, Verification and Validation*, 2010, pp. 65-74.

[20] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," *Proc. Int. Conf. on Automated Software Engineering*, pp. 273-283, 2005.

[21] I. Vessey, "Expertise in debugging computer programs," International *Journal of Man-Machine Studies: A Process Analysis*, vol. 23, no. 5, pp. 459-494, 1985.

[22] R. M. Stallman and R. H. Pesch, "Using GDB: A guide to the GNU source-level debugger," Free Software Foundation, 1991.

[23] Siemens benchmark suite: http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/

[24] DIAMOND project website: http://www.fp7-diamond.eu/

[25] V.N. Kas"janov, "Distinguishing Hammocks in a Directed Graph", *Soviet Math. Doklady*, vol. 16, no. 5, pp. 448-450, 1975.

[26] M. Weiser, Program slicing, *IEEE Transactions on Software Engineering*, vol. 10, no. 4, 1984, pp. 352–357.

[27] B. Korel and J. Laski, Dynamic program slicing. *Information Processing Letters*, vol. 29, no. 3, 1988, pp. 155-163.

[28] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". *IEEE Computer*, pp. vol. 11, Issue 4, 34–41, 1978.