Data Mining Diagnostics and Bug MRIs for HW Bug Localization

Monica Farkash
University of Texas
Austin, TX, USA
Monica.Farkash@utexas.edu

Bryan Hickerson
IBM
Austin, TX, USA
bhickers@us.ibm.com

Balavinayagam Samynathan
University of Texas
Austin, TX, USA
balavins@cerc.utexas.edu

***Abstract*—This paper addresses the challenge of minimizing the time and resources required to localize bugs in HW dynamic functional verification. Our diagnostics solution eliminates the need to back trace from point of failure to its origin, decreasing the overall debugging time. The proposed solution dynamically analyses data extracted from sets of passing and failing tests to identify behavior discrepancies, which it expresses as source code lines, coverage events and timing during simulation. It also provides a visual diagnostic support, an image of the behavior discrepancies in time which we call a Machine Reasoning Image (MRI). This paper describes in detail our data mining solution based on coverage data, HDL hierarchies and time analysis of coverage events.**

Our approach brings a data mining solution to the problem of HW bug localization. It defines new concepts, provides in-depth analysis, presents supporting algorithms, and shows actual results on archetypical problems from PowerPC core verification as an industrial application.

***Keywords*— diagnostics; bug localization; debugging; verification; EDA tools;**

I. Introduction

Hardware development is a lengthy and resource intensive process. It requires not only an extensive effort on the design side, but an equivalent endeavor is presented in the functional verification realm of which debugging is the most time consuming activity [1].

There is an unsparing amount of tools that help identify a discrepancy between expected and exhibited design behavior but there is an unpredictable timespan from the origin of a problem to its detected manifestation, therefore identifying the origin of the exposed discrepancy is a major challenge.

While tools support stepwise inspection of the design, back tracing the problem from its manifestation to its origin remains manual labor intensive. Hardware designs are inherently parallel therefore back tracing requires inspecting and vetting a large amount of concurrent behavior which can span over an unbounded number of cycles and all but one are false tracks.

A faster debugging approach starts with an assumption of the bug origin and evaluating the correctness of this assumption. This is significantly cheaper and faster than full back tracing, requiring only forward simulation from a known point in the test. Guessing the faulty behavior origin is a subjective process based on experience and detailed design knowledge. For example a data discrepancy in a memory location can stem from a large variety of problems like coherency, exceptions, arithmetic operations, unexpected order of events, or address translation. Excluding them one by one as a potential cause is more expensive than simply proving an incorrect assumption, for example, a given exception was wrongly triggered.

There is a significant body of work in the area of automatic debugging [2]. In HW a number of automatic error localization methods have been proposed, mainly using formal reasoning [3]. They are generally limited to identifying failures involving a very small number of gates and may return hundreds of candidates, which may not be helpful in manual debug. Our solution approaches the problem from a completely different angle. It works at a high RTL level, inexpensively using already existing coverage data, pointing to differences of behavior distributed in time that involve large parts of the design, (not bound to a single-point of failure).

Data mining solutions were also successfully used previous for related problems [4]. In SW there is a wealth of work in using data mining approach for SW feature identification and bug localization [5]. Our solution follows the lessons learned from the SW analysis, with the necessary creative changes required to apply it in HW.

This paper reveals a data mining process to identify atypical behavior to support the HW bug localization effort, pointing to the probable origin of the behavior discrepancy. The described solution provides the relevant information as functionality defined by the coverage events critical for distinguishing the failing from the passing tests, the source code statements relevant to the behavior difference, and the location within the design (HDL hierarchy) of the exhibited behavior. It also identifies tests which exhibit the faulty behavior early during simulation and provides the exact timing of the events for debugging windows with a visual debugging support.

II. HW VERIFICATION PROCESS

Dynamic functional verification implies running tests on a HW implementation of the design and comparing exhibited to expected behavior. Monitors are implemented Boolean functions added to the execution. Correctness monitors identify and flag discrepancies between the monitored behavior and the expected one. Coverage monitors and events gather information regarding the design areas that were exercised during the simulation run.

Numerous tests are run daily with the majority passing. The few tests failing due to the same monitors, like wrong cache state or incorrect TLB invalidation are considered expressions of the same bug. These fails are grouped together and distributed for debugging. The debugging starts with choosing one test from such a group, identifying the bug manifestation event, and back tracing from there to the potential origin of the problem.

Tests consist of both initial conditions for the environment and instruction steams (supported ISA instructions). Instruction streams can be specifically chosen for specific hardware stimulation or randomly chosen. Automatic test case generators are used to generate deliberately dissimilar tests within a given verification plan [6].

A test can contain numerous instructions which, while running, can interleave in time while sharing resources. They change not only data but also the conditions in which the next instruction finds a resource, like control registers, cache line state, or buffer state. Due to these internal interactions the testing conditions while running a test are not the sum of the testing conditions for each instruction separately.

To compare with Software, each instruction could be considered as a different SW thread that runs in parallel with all the others, in a highly concurrent, resource sharing and critically time sensitive SW system [7]. The number of potential order of interleaving concurrent activities increases with the test size and renders existing SW solutions which require an analysis of all possible interleaving alternatives.

III. HW DYNAMIC ANALYSIS

The general idea in SW dynamic feature analysis is to compare the code coverage of a test that exhibits the targeted behavior to the one that does not. Then remove from the analysis the code lines covered in the same sequence, rendering them irrelevant, and identify the code lines that differ, the origin of the difference, and the conditions that trigger it.

We offer a HW version of the above analysis for bug localization and diagnostics which we call *Applied HW Dynamic Behavior Analysis* (ADBA), which is based on three creative decisions:

A. *Utilizing event coverage*

We replace the use of code coverage with event coverage because we look to expose the functionality that triggered the difference in behavior, and coverage events represent exercised functionality.

B. *Keeping irrelevant code*

A SW problem can be reduced by removing irrelevant code from its analysis. In HW a slight change in the test or in its initial conditions can trigger an avalanche of discrepancies due to the natural HW reaction to input (e.g. cache instructions, bus transaction interleaving, timing of servicing exceptions). This means that all the code up to the manifestation of the problem is to be considered relevant or one has to invest expensive effort into identifying an eluding smaller sub-test. Our solution allows the tests as-is, due to the next decision.

C. *Using statistical approach*

The available tests, failing or passing, being diverse by design, challenge us with intentionally dissimilar coverage results from which we need to select the coverage events that best describe the behavior discrepancy. Hence we turn to using a statistical approach to comparing tests. We can learn by comparing the regressive components of the passing and failing tests groups.

Thus, following the decisions above, ADBA compares the event coverage of failing and passing tests using data mining to identify the coverage events that define the difference in behavior as well as to provide visual debugging support.

IV. APPLIED HW DYNAMIC BEHAVIOR ANALYSIS

Our Applied HW Dynamic Behavior Analysis approach provides a bug origin assumption based on HW dynamic analysis. The debugging process consists therefore of validating the assumption, a fast and easy process compared to back tracing all possible reasons for the exhibited faulty behavior.

A. *ADBA usage*

ADBA is not meant to completely replace traditional debugging but to be used as a supporting method alongside it. Traditionally, the verification engineer would start with back tracing on subjective assumptions. We replace those with objectively extracted information based on the data mining of the dynamically gathered data. Our approach guides the verification engineer to the most likely origin of the difference in behavior, the coverage events that best express the functionality of the difference, and the exact timing within the test. This provides a head-start in the debugging challenge with disregard to the experience and detailed knowledge of the design presented by the verification engineer.

For example back tracing to reach the conclusion that an asynchronous external interrupt is not being gated properly is very difficult, while, if suspected, vetting it is easy and fast by comparison.

B. *Deviant Behavior*

We call the behavior discrepancy, the difference between the expected and the exhibited behavior, deviant behavior. Only if the deviant behavior is ruled as being faulty, we will consider it as faulty behavior.

C. *Coverage as Bug Indicators*

Coverage events by their nature do not represent bugs, nor do deviant behaviors. For example reaching a buffer full state is not an indication of a bug, nor is a data storage interrupt. They can happen in both failing and passing tests. They become such indicators only in a given context. This is why they can point to the most likely origin (as in functionality, code source statement, and timing within test) of a deviant behavior, not immediately define it.

D. *ADBA Assumptions*

Our problem of finding the relevant coverage events that characterize the deviant behavior can be re-defined as the problem of stepwise removing the coverage events which are not relevant, to expose only the relevant functionality. We base this process on three basic assumptions:

Origin: We assume that all failing tests have in common the origin of the deviant behavior, the functionality that takes all the failing tests on a different path than expected. If there are coverage events in or very close to that decision, we expect all failing tests to contain them.

Weight: We argue that the deviant behavior should be less exhibited by one group of tests than the other. For example, if the bug is related to a buffer full state, then the failing tests will all contain the event regarding the buffer full, while the passing might or might not contain it.

The common behavior of failing tests can be the lack of a given behavior. For example, the failing tests are less likely to exhibit a given flush. The lower likelihood of such a flush is the deviant behavior. The behavior itself (the flush) can be identified as a characteristic of the passing tests group.

Locality: We assume that the clustering of coverage events in time and location within the design help and support the understanding of the deviant behavior.

Based on the above assumptions, the basic idea behind our approach is to identify the coverage events (or lack of) which are a characteristic of failing tests and are more preponderant in failing than passing tests.

E. *Stepwise Processing*

The process based on the above assumptions reduces the initial set of coverage events until it reaches a subset which represents the likely origin of the deviant behavior, Fig. 1.

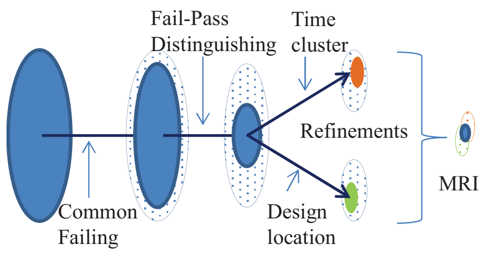

Figure 1. Data Mining Stepwise Process

Step1. Remove coverage events that are not common to all failing tests. (*Assumption1*)

Step2. Solve the combinatorial problem of deciding which coverage events are more likely to be seen in a group of tests than the other. ADBA uses an already existing algorithm, developed to build decision trees, for the purpose of solving our problem. While building a decision tree that is meant to identify the data attributes that most efficiently differentiate between failing and the passing tests we identify the critical coverage events, which we call the *distinguishing* coverage events. (*Assumption2*)

Step3. Provide a visual diagnostic support. Our Machine Reasoning Image (MRI) is a visual representation of the deviant behavior in time and location within the design. (*Assumption3*)

A small number of distinguishing coverage events would point directly to the origin of the bug. If there is a large amount of such coverage events, it is difficult to abstract the deviant behavior they characterize. We refine the solution by using the clustering of coverage events in time and location within the HDL hierarchy, and build MRIs. The MRIs exhibit the behavior pattern and can be used to extract the exact timing and location within the design of the deviant behavior.

V. ADBD Implementation

We implemented and indiscriminately used our algorithm on nine HW bugs identified during IBM's PowerPC pre-silicon core verification process. The large variation in the type and number of tests available, varying from 100 failing to only one, reflects the real problems and data availability conditions during the industrial verification process. The sample results are given in the table below (table 0) for the 9 cases analyzed (B1..B9):

Group of Tests	**Table 0. Number Tests Used Per Analyzed Bug**								
	B1	***B2***	***B3***	***B4***	***B5***	***B6***	***B7***	***B8***	***B9***
Failing Tests	100	46	43	10	6	10	12	1	9
Passing Tests	100	49	50	10	12	10	14	13	10

A. *Data Processing Algorithm*

The data mining problem considers each test as a record, and each coverage event as an attribute. For each set of tests representing a failing monitor (bug set) the algorithm consists of:

1. *Construct data mining problem*
2. *Add fail or pass* attribute
3. *Clean data*
4. *Keep only attributes common to failing*
5. *Build decision tree*
6. *Separate classifying attributes for failing/passing*
7. *Build MRI (optional)*

[1] *Construct Data Mining Problem*. Even though the running HW contains a very large number of coverage events (150k+), the bug sets would have triggered only a subset, reducing the problem space to the following set of initial attributes:

Table1. Total Number Attributes								
B1	***B2***	***B3***	***B4***	***B5***	***B6***	***B7***	***B8***	***B9***
13071	25806	45645	14878	32014	13758	28932	10247	57318

[2] *Add Pass Fail Attribute.* We set the attribute for passing or failing for each test for differentiation later

[3] *Clean Data*. We use basic data filters implemented in Matlab [8, 9] to remove tests which would provide no new information, or are outliers, hence less probable to bring value to this problem

[4] *Keep only attributes common to failing.* The number of coverage events common to failing tests is influenced by the number of available tests and varies among bug sets. For our samples, we see the number of failing attributes (table 2)

Table2. Number Attributes Common to All Failing

B1	***B2***	***B3***	***B4***	***B5***	***B6***	***B7***	***B8***	***B9***
1102	10085	1456	6631	7481	4970	9676	10247	12960

[5] *Build decision tree.* We decided to use an already implemented version (J48-Weka) of the C4.5 algorithm, a statistical classifier, to identify the classifying attributes. C4.5 does the following:

```
For all attributes
        Find the normalized information gain ratio from
        splitting on one of its values
        Choose the attribute providing the highest gain
        Add it to the decision tree
Do it recursively on the remaining attributes
Prune the tree
```

C4.5 picks the set of attributes out of which to build the tree such that for each tree node it selects those that most effectively split the set of records into subsets based on normalized information gain [8]. The resulting classifying attributes best distinguish between the passing and failing tests.

[6] *Analyze failing/passing*. Translate the attributes back into coverage events, and separate them into two sets, the set of events found in the passing tests (*SetPass)* along with those failing tests (*SetFail)*. The resulting sets are provided to the user. Table 3 shows the large variation in the number of distinguishing coverage events.

	Table 3. Number of Distinguishing Coverage Events								
	B1	***B2***	***B3***	***B4***	***B5***	***B6***	***B7***	***B8***	***B9***
SetFail	1	1	21	1	1	2	33	36	1
SetPass	-	-	-	-	-	527	4	-	-

If *SetPass* is empty, the deviant behavior, which represents the failure, can be observed only in the failing tests. If *SetFail* is empty, the lack of that behavior described by *SetPass* is characteristic of the failure! If none of them is empty, *SetPass* shows the behavior of the passing, while the failing seem to have replaced it with the behavior characterized by *SetFail*.

[7a]*Building MRIs*. An MRI is a visual representation of the deviant behavior for a given test and it is built as follows:

a. The distinguishing coverage events are correlated to representative signals we wish to trace during simulation.
b. Extract the timing of these corresponding signals during simulation.
c. The location of the distinguishing coverage events is extracted from the compiled design. The coverage events are grouped according to their location in the design, based on hierarchical HDL structure.
d. Each such group is colored distinctly.
e. The representation is plotted with the x axis representing the simulation cycle of when a coverage event is being triggered. The y axis represents the location in the compiled design.

[7b]*Reading MRIs*. Case B6 has the highest number of distinguishing coverage events (Table 3.). Figure2 shows the MRI of a passing test defined by 527 coverage events.

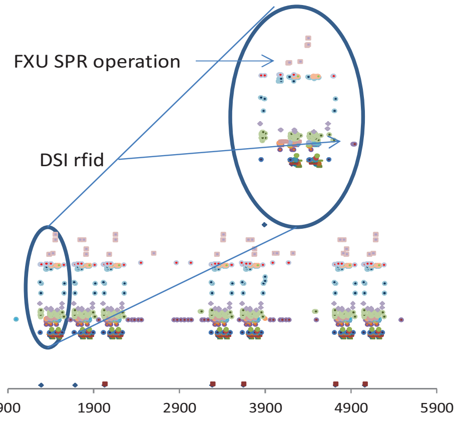

Figure 2. MRI for B6 – DSI exception

This is an example where the lack of behavior represents the deviant behavior. The pattern shows a repetitive action. Each color represents a different location in the design. Familiarity with the data flow in the design would immediately point to servicing an interrupt. The names of the coverage events are by themselves explanatory. We notice an *sprg0_mtspr_access* in *FXU SPR* (Fix Point Execution Unit - Special Purpose Register) operation typical for entering the exception handler and *spr_msr_mux_select_rfid* for a *DSI* (the return from interrupt). This is the MRI of an erroneous triggering of an exception.

Case B3, shown in Figure 3, represents the MRI of an interrupt caused by a trap instruction. ADBD succeeds in identifying a smaller subset of coverage events that are typical to this type of interrupt, hence easier to recognize.

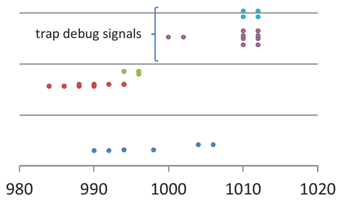

Figure 3. MRI of B3. Trap Interrupt

The MRI for case B7 is shown in Figure 4. A repetitive coverage event *count_ucode* is shown, which is located in the instruction fetch unit (IFU) and represents the routine which breaks up an operation into microcode. We identify the operation as store, which immediately points us to problems with atomicity of a store operation.

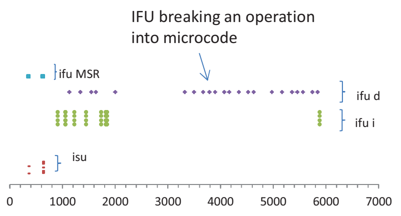

Figure 4. MRI for B7 – Atomic Operation

VI. Analysis and Results

We used our method of identifying the bug origin on nine real world cases. These cases used raw data without pre-selection to fully represent actual use cases.

A. *Problem Types*

We encountered a variety of bug types.

Cache coherency: Cases B4 and B5 exposed different aspects of cache coherency problems, where B4 was a typical load hit store to the same address sequence. Timing and the build-up of internal conditions is very important for hitting these problems. The ADBD distinguishing events pointed to the exact origin of the bug without requiring any additional information about the test instruction stream.

Memory consistency: Case B7 was a memory consistency problem. Under certain conditions an operation, which was supposed to be atomic, didn't behave as expected. Atomic operations are always tricky to verify given the interactions with other processors/threads. ADBD was able to point to the core of the problem, the functionality that breaks up the instruction in multiple (microcode) segments, which needs special handling under the constraints of an atomic operation. The cycle at which it happened helps identify the exact instruction at fault.

Exception related: Cases B1, B6 and B9 expose different aspects related to exceptions, being wrongly triggered, or serviced. Case B9 is particularly difficult because it was an asynchronous external interrupt not being gated properly by the HW, and asynchronous events are more difficult to monitor.
Case B3, even though related to a synchronous interrupt, pointed correctly to a taken trap instruction as being the differentiating factor.

Address translation: Case B2 exposed a potential PTE (Page Table Entry) problem in the address translation mechanism. Address translation problems can take hours even for the most trained expert to diagnose, due to the inherent complexity of the address translation mechanism. Pointing directly to the problem saves not only time to diagnose but also requires less expertise.

Live lock: Case B8 shows a live lock problem, Instruction Fetch related hang, which kept the system working without advancing. Live lock problems are particularly hard, because they keep the system busy, show a lot of activity, without advancing, and sifting through huge amount of information to identify the origin of the hang, then its cause, can take a very long time.

B. *Dependency on Events Distribution*

Our method relies on the existence of coverage events in the functional area of interest. For cases B1, B2, B4 and B5 and B9 the solution pointed to a very small amount of coverage event and they pointed directly to the exact functionality which characterized the bug. The events not only were perfect descriptors of the bug, but, after inspecting the available coverage events, they were also the best choices possibly made to describe that particular behavior.

C. *No Events in the Problematic Area*

The main concern is when there are no events that in any way characterize the origin of the deviation. This was the case in B3, the trap instruction decoding. Puzzling at first sight, the distinguishing coverage events seemed to be totally disconnected. They pointed to two different macros with no immediate connection. At closer investigation, those were the events of entering and exiting the area of interest, which had no independent events. In this situation our approach was less effective than in the other, but was helpful by pointing out the timing in the test, as a time window, and a further inspection of the events in that window pointed to the origin of the bug. These missing coverage events would be indicative of areas of the design which need to re-assess for possibly adding extra coverage events.

D. *Too Many Events*

The example cases B6, B7, B8 are representative of bugs where the difference in behavior is significant in time and activity. For this, the number of distinguishing coverage events is large. If we look at the classifying ranking among them as provided by the decisional tree, the coverage events ranked as most important would not represent the quintessence of the deviant behavior, but rather whatever coverage event is mostly exercised while exhibiting it. The MRIs represent how events are clustered together in simulation cycles and location, which enables the interpretation of the behavior pattern.

While each coverage event by itself doesn't present value, the MRIs show the evolution of the exception, which, knowing the meaning of the points, is easy to distinguish and hence to point the debugger into questioning the reason why the failing do not present the same exception.

MRIs can become intuitive for users that understand the architecture, and the meaning of the events.

E. *Overall Result*

In our nine examples, which represent consecutive, non-selected, industrial real world examples our solution proved to always find the origin of the deviant behavior to the extent of available coverage events. Most of the time it pointed directly to the coverage event that best described the functionality exposing the problem. When there was none, it pointed to the entrance and exit from that functionality. In case the problem was not punctual but rather covered an extended behavior, the MRIs expose it, with clear indications and when and where events happen that characterize the problem.

VII. Overview and Conclusion

HW bug localization is known to be the most expensive activity in the verification process. Our work shows a diagnostics data mining solution based on a generic functional verification via simulation methodology conditioned only by coverage recording of passing and failing tests.

Automatic solutions to this problem were proposed before, with interesting results though limited results depending on the size of the problem or the languages used [10]. There is extensive and successful work done in the area of localizing features in SW [5] and post-silicon [11, 12].

Generally based on code coverage, SW dynamic analysis compares traces of tests with and without a particular behavior and identifies the code which is responsible for that exhibited behavior [13, 14]. Alternatively, the frequency of execution portions of code can be analyzed to locate the implementation of a specific behavior [15] [16].

We re-invent the approach for HW debugging and use data mining to extract the differences between sets of failing and sets passing tests. We express the problem of identifying the difference between the characteristics of failing tests and the normal behavior as defined by passing test, as the problem of building a decision tree. It results in deciding the number and the ranking of coverage events that best differentiate among the two sets hence are best descriptors of the bug.

We introduce a novel source of information, plotting the coverage events in time, and use it together with information regarding the physical location of the coverage event in our design, to build the MRIs of a bug. The MRIs provide an easy to read, visual representation of a bug.

We ran the algorithm on subsequent bugs, presenting a large variety of problems, and available data. In spite of this variation, our solution consistently provided us with the right answer, adding value to the understanding of the bug. It also helped us choose which from the failing tests to debug (as the test which exhibited the problem earlier in time and with the easiest to read visible pattern).

As an automatic aid to manual debugging, this solution proves to be extremely valuable on a daily basis, and invaluable for failures received from the lab.

VIII. Future Work

ADBD was consistently accurate on an industrial processor core design on consecutive bug sets. It wasn't tested on different unit designs or on systems.

The method relies on the existence of coverage events throughout the design. We met a single case where there was no coverage event in the area that contained the bug. We would need to see more such cases to understand if it is generally the case that the results ADBD can provide continue to bring value.

References

[1] Wilson Research Group, "The 2012 Wilson Research Group Functional Verification Report," 2012. [Online]. Available: http://www.mentor.com/products/fv/multimedia/the-2012-wilson-research-group-functional-verification-studyview. [Accessed 20 05 2014].

[2] A. G. Veneris, B. Keng and S. Safarpour, "From RTL to silicon: The case for automated debug.," in *ASP-DAC*, 2011.

[3] B. Keng and A. G. Veneris, "Automated debugging of missing input constraints in a formal verification environment.," in *FMCAD*, 2012.

[4] Z. Poulos and A. G. Veneris, "Clustering-based Failure Triage for," in *International Test Conference*, Seatlle, Washington, 2014.

[5] B. Dit, M. Revelle, M. Gethers and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process,* vol. 25, no. 1, pp. 53-59, 2013.

[6] A. Adir, E. Almog, L. Fournier and M. Eitan, "Genesys-Pro: innovations in test program generation for functional processor verification," *Design & Test of Computers, IEEE,* vol. 21, no. 2, pp. 84 - 93, 2004.

[7] D. Edwards, N. Wilde, S. Simmons and E. Golden, "Instrumenting Time-Sensitive Software for Feature Location," in *International Conference on Program Comprehension*, Vancouver, 2009.

[8] I. H. Witten, E. Frank and M. A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, 2011.

[9] MathWorks, "Matlab," 2014. [Online]. Available: http://www.mathworks.com/products/matlab/. [Accessed 20 05 2014].

[10] H. M. Le, D. Grosse and R. Drechsler, "Automatic TLM Fault Localization for SystemC," in *Computer-Aided Design of Integrated Circuits and Systems, Volume:31, Issue:8*, 2012.

[11] S.-B. Park and S. Mitra, "IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors," in *Design Automation Conference*, Anheim, California, 2008.

[12] A. Tepurov, V. Tihhomirov, M. Jenihhin and J. Raik, "Localization of Bugs in Processor Designs Using zamiaCAD Framework," in *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on*, Austin, TX, 2012.

[13] T. Eisenbarth, R. Koschke and D. Simon, "Locating Features in Source Code," *IEEE Transactions on Software Engineering,* pp. 210-224, March 2003.

[14] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice,* vol. 7, no. 1, pp. 49-62, 1995.

[15] A. D. Eisenberg and K. De Voler, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," in *Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest. Hungary, 2005.

[16] H. Safyallah and K. Sartipi, "Dynamic Analysis of Software," in *Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06)*, Athens, Greece, 2006.