COMPUTATION OF THE MINIMAL SET OF PATHS FOR OBSERVABILITY-BASED STATEMENT COVERAGE

J. COSTA, J. MONTEIRO
IST/INESC-ID, PORTUGAL

KEYWORDS: Embedded software, Validation, Observability, Coverage

ABSTRACT: Existing coverage-based validation methods guarantee the execution of a certain percentage of the program code under test, however they do not generally verify whether the statements executed have any influence on the program's output. Motivated by an observability coverage metric for embedded software we propose a coverage-directed path generation method. In this method, a program statement is considered covered not only if it belongs to the executed path, but also if its execution has influence in some output. The paths are generated by finding the longest path in a tree representing the possible execution paths of the program. Generated paths are then validated to check for feasibility. If a feasible path is found, then we determine and mark the statements actually observed using the computed inputs for exercising the path. If unfeasible, we search for the next longest path. If the desired level of coverage was not obtained yet then a new tree is built. This new tree will reflect the coverage obtained so far and also the information gathered when checking the feasibility of previous paths. We present results that demonstrate the effectiveness of this methodology.

INTRODUCTION

Embedded systems are used in a growing number of diverse applications. Examples include consumer electronics, automotive systems and telecommunications, among others. This prevalence is due to the fact that embedded systems results from a mix of hardware/software systems. The software part, which runs on a processor, gives the system the flexibility, since it can be easily changed depending on the application. The hardware portion, which executes more specialized functions, is used in time critical subsystems.

Techniques for the formal validation of such systems are being developed [5]. Nevertheless, simulation is still the best option when trying to validate a design. Validation of embedded systems is hard because of their heterogeneity. Software and hardware should be simulated simultaneously, and furthermore hardware and software simulations must be kept synchronized, so that they behave as close as possible to the physical implementation. Several methods have been proposed for co-simulation [4],[3],[13].

Many of the co-validation fault models currently applied to hardware/software designs have their origins in either the hardware or the software domains [11]. A number of these models are based on the traversal of paths through a Control Dataflow Graph (CDFG) representing the system behavior (Figure 1). Normally, these systems are described in high level languages such as Verilog or VHDL for hardware, and C or Java for software, among others. Having a system described in a high level language means that its description can be easily converted into CDFG descriptions. Having the hardware description and the software description both in CDFG format means that the entire system can be in the same format. And having the entire system described in the same format means that the same techniques applied to hardware CDFG-based methods can be applied to software and vice-versa.

In this paper, we address the problem of, given an embedded software program, finding a minimal set of execution paths that guarantees a user-specified level of observability coverage. The method starts by building a Directed Acyclic Graph (DAG) from the CDFG. Using the DAG a graph tree is build where each node corresponds to a set of statements in the embedded program. To obtain the longest path that also executes that set of statements is obtained by climbing up the tree starting in the mentioned node. The path is then validated against an input value generator to test its feasibility, and if feasible its observability-based coverage is computed. The coverage obtained directs the choosing of the next path by rearranging the tree accordingly. Paths are generated until the specified coverage is achieved.

The method proposed is based on traversal of paths through a CDFG and thus can be applied to either software or hardware high level languages. Also, the coverage metric we used is motivated by work on observability-based coverage metrics for hardware models described in a hardware description language [7] and afterwards by the same metrics applied for embedded software [2].

In the following sections we give an overview of the field of automated testing of embedded systems and describe our own method to obtain an observability-based coverage. Later we present some examples. Finally, we present some conclusions and future work.

RELATED WORK

Several types of coverage metrics exist that use a CDFG description. The most complete is path coverage where we achieve 100% path coverage when every possible path in the CDFG is executed. This means that from the entry point of the CDFG description all possible ways of getting to an exit point were followed and executed. Reaching

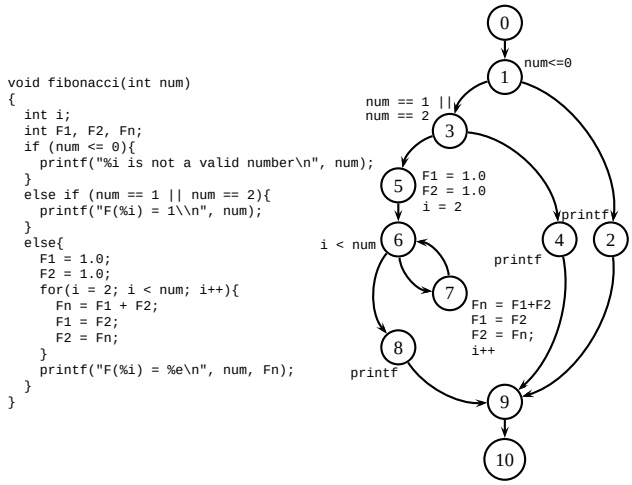

Fig. 1. C *code for the Fibonacci program and its* CDFG.

100% path coverage is very often impractical due to the large number of possible paths, or even impossible as some of the paths may be infeasible. Branch coverage, on the other hand, is not such a strong measure of the system correctness, but good enough in practice for many instances, and is relatively easy to achieve. Branch coverage consists of exercising all alternatives for every branch of the CDFG.

Embedded systems coverage

To obtain a set of input vectors that give a certain coverage level of an embedded system several algorithms can be used [1],[12],[18]. These algorithms heuristically modify an existing test set to improve total coverage, and then evaluate the fault coverage produced by the modified test set. If the modified test set corresponds to an improvement in fault coverage then the modification is accepted. Otherwise the modification is either rejected or another heuristic is used to determine the acceptability of the modification.

Most of the covered directed algorithms are based on fault activation without taking into account observability. Observability consists of not only activating some fault, but also in verifying explicitly its effect on some observable point. Methods to measure observability coverage were presented for both hardware and software. An observability coverage-directed method for input vector generation for hardware was introduced [6]. In this paper we propose a method that achieves a similar goal for software, and which can be adapted in order to integrate the hardware and software methods.

Coverage-directed software path generation

Several methods have been proposed for coverage-directed software path generation. Some of those methods were intended for general software, while others were intended specifically for embedded software.

Evolutionary testing searches test data that fulfill a given structural test criteria by means of evolutionary computation. In general it starts with an initial test vector that is generated at random. Afterwards, the test vectors are evaluated to determine their fitness value. The test vectors are then subject to mutations and/or combinations in order to obtain new test vectors that try to fulfill the test criteria. In [17] an evolutionary test method was presented that could be applied to statement tests, branch tests, condition tests and segment tests.

Dynamic methods generate input data by running the program and gathering information along its execution. In [8] input data for branch coverage is generated by dynamically selecting a path in an attempt to exercise a test branch in a given program. It uses the approach presented in a test generation relaxation technique [9] to guide the path selection. The path selection is done by dynamically switching execution to a path that offers less resistance in order to force execution to reach the given branch. The resistance of a branch tries to measure how difficult it is for that branch to be executed.

A method intended specifically for embedded software was presented in [14]. This method is based on a coverage-driven validation approach in order to stress and cover variables and function calls in embedded software, running on a SystemC PowerPC microprocessor model.

While the methods mentioned here are representative of the area, there are many other variations of coverage-directed methods for software. These methods are simply concerned with executing a certain percentage of the program code under test. Yet, knowing which executed percentage has some influence on the program outputs is even a more relevant measure. To our knowledge none of the existing coverage-directed methods verify whether the statements executed have any influence on the program's output. In this paper we propose a coverage-directed method based on an observability metric.

PROPOSED METHODOLOGY

In the previous section we presented an observability coverage-directed input vector generator for hardware. To the best of our knowledge, there is no equivalent for software. In this paper we propose an observability coverage-directed method for input vector generation for software. Our work is motivated by the observability coverage metric for embedded software of [2]. In this method, the software program is modified to give more information on the original program. Then the modified program is executed and an observability coverage is computed.

Overview of our method

For every function in the program under test we first obtain its CDFG representation, as described in the next section. In these CDFG we clearly mark the vertices that are decision points and the vertices that are function calls. From the CDFG representations we obtain a single direct acyclic graph (DAG) where the start vertex correspond to the start of the program execution and the last vertex is the exit of the program. In this graph the loops are unrolled (see next section) and the function call vertices are expanded with the CDFG of that function. When expanding function calls we avoid at first doing recursion more than once.

From this DAG we obtain a tree representation where for each node we can easily obtain the longest path. The path

obtained is then submitted to an input vector generation tool in order to assess its feasibility. If it is not feasible then the information obtained from that path is used to modify the existing tree or build another one.
If it is feasible then we check its coverage using a software coverage observability meter. If the added coverage is equal or greater then the one we aimed to obtain, then we exit the program. Otherwise we mark those observed statements and build another tree accordingly.
In the rest of this section we state the methods we use in our own method. Later, in the next section, we present how we integrate everything to obtain an observability-based coverage-directed method for software.

Input vector generation

When we obtain a path, at first we do not know if the path is feasible or not. In order to test its feasibility, and if feasible to measure its coverage, we must use an input vector generation method. We use a dynamic method based on relaxation techniques proposed by Gupta et al [9]. In this method test data generation is initiated with an arbitrarily chosen input from a given domain. This input is then iteratively refined to obtain an input on which all the branch conditions on the given path evaluate to the desired outcome. In each iteration the program statements relevant to the evaluation of each branch condition on the path are executed, and a set of linear constraints is derived. The constraints are then solved to obtain the increments for the input values. These increments are added to the current input values to obtain the input for the next iteration. The relaxation technique used in deriving the constraints provides feedback on the amount by which each input variable should be adjusted for the branches on the path to evaluate to the desired outcome. When the branch conditions on a path are linear functions of input variables, this technique either finds a solution for such paths in one iteration or it guarantees that the path is infeasible.

Software observability coverage metric

In order to know if the desired level has been attained, we must measure the observability coverage for each input data obtained from the input vector generator. We use the observability coverage metric for embedded software described in [2].
In that method, in order to achieve the observability target, one keeps track of all the statements that assign a variable. For that purpose, for each variable in the program there is a list of statements the variable depends on. When the execution arrives at a statement the variable that will be assigned is stored. For that variable a list of dependencies is built which is the set union of the dependency lists of the variables that are at the right hand side (RHS) of the assignment.
When an observable statement is reached, where the content of some variable is passed to the exterior of the program, the statements in its list of dependencies are checked. The statements that are not on that list are not observable from that output.

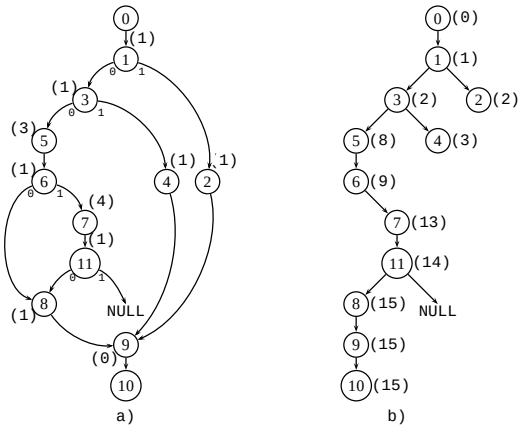

Fig. 2. DAG of Fibonacci example and corresponding tree.

PROGRAM DAG

From the source program code, we extract multiple control dataflow graphs (CDFG), one for each function in the program. The CDFGs obtained are directed graphs, which can be cyclic if the functions have loops. The vertices in the CDFGs correspond to the program statements and each vertex can have more than one statement. That is represented by its weight. Also, the vertices symbolize blocks of code that if one statement in that block is executed all other are also executed. The vertices can also represent conditions and in that case the vertex will have two outgoing edges that correspond to the branches. Also, in order to interconnect the CDFGs as in the program, when we have a function call vertex we mark on the CDFG vertex which CDFG the function call corresponds to.
From the CDFGs we build a new graph that starts in the CDFG that corresponds to the main function. We traverse the CDFGs, expanding function calls and loops along the way. In the end we get a directed acyclic graph, DAG, of the program. Note that we do not expand function calls and loops indefinitely. The first time the loops are expanded just once. In Figure 2a) we show the unrolling of a loop. Also, the first time the function calls are expanded until we detect that the calling of some function will enter into a recursion. The rest of the expansion is done on demand:

- when all the paths of the current expanded graph are categorized as either feasible or infeasible and the coverage metric was not yet achieved, or
- the input vector generator returns the path as infeasible in a vertex that is the start of the loop because the loop is not further expanded.

FINDING THE LONGEST PATH

In order to increase our coverage we are interested in getting a path that:

- was not yet executed or was not found to be infeasible;
- has the greatest number of statements in order to improve our chances of covering the greatest number possible;
- has statements that were not yet observable by previous paths.

We model this problem by representing it using a tree. From the tree we get a path and test its feasibility using an input vector generator. The information extracted from the input vector generator will be used to rebuild the tree and thus get another path as explained below.

Computation of the path

Computation of the longest path is done by building a tree graph (Figure 2b). The nodes of this tree will correspond to vertices on the DAG (numbers inside the nodes in Figure 2b). Adjacent nodes correspond to adjacent vertices. Traversing the tree upwards, from a given node, will give us a sequence of vertices. Those vertices define a path in the DAG. In the tree, only a small set of path are defined at one time. Except for one special case, the leaves of the tree will correspond to paths that are infeasible (leave with NULL in Figure 2b), that were already tested (there are none yet in this example) or that will not give the longest path from start to finish of the DAG (all nodes except 10). The special case leaf (node 10), corresponds to the last vertex in the DAG and thus gives us the longest path. Also, for each node we have the number of statements that correspond to the execution of the path (numbers in parenthesis in Figure 2b).
To obtain this tree we start by obtaining a list of vertices in topological order. Thus, by going through the list we guarantee that when we process a vertex v we have already processed the vertices whose edges are directed to vertex v. For each vertex we build a new tree node. That tree node will be connected to the node of the ingoing vertex with the greatest distance. Afterwards, we update the distance of the vertex and its node with the sum of the obtained greatest distance and the number of statements of the vertex. In Figure 2b, we connect the node 9 with node 8 and not with nodes 2 or 4 because that way we can achieve the greatest distance to node 9. By doing this in a topological manner we end up with a tree where each node gives us the maximum distance from the start vertex and its corresponding path.

Avoiding infeasible paths

When we get a path that is infeasible we go through the tree nodes until we get to the node corresponding to the last executed vertex. That vertex will have two outgoing vertices. One of them will correspond to the infeasible path. Thus, all the nodes that are below the tree node corresponding to that infeasible vertex will be removed. Then, the node corresponding to the infeasible path is marked as infeasible. Then we go up one node in the tree. If that current node as all the nodes below it marked as infeasible then those nodes are deleted. Then the current node is marked as infeasible and again we repeat the process of going up one node. We stop when one of the nodes below the current node is not marked as infeasible. This is done in order to reduce the size of the tree.
After we deleted the path from the tree we go through the infeasible path and for each vertex in that path we try to build an alternative path. The rationale here is that we try in each decision vertex to choose a different path of the infeasible one. So, for each vertex in that path we check to see if there are any ingoing vertex whose corresponding tree node is a leaf (if it not a leaf then it means that through that node already exists a path and the alternative will not be a better one). If there are then we choose the leaf with the greatest distance value and connect the new node to it. For this new node we compute the distance value. Doing this for all the vertices in the infeasible path we end up with an alternative path. When we reach the end of the path we end up with a leaf that most likely will not correspond to the last vertex of the DAG. Thus we must compute the rest of the tree by doing what we described in the previous subsection. But instead of starting in the beginning of the DAG we start with the last vertex of the infeasible path.

Observability vertices

Since we are interested in observability coverage we want to execute at least one of the vertices that have observability points. Also, when choosing the path we want the longest one. Therefore, we go through the list of tree nodes that correspond to observability vertices and get the one with the greatest distance from the root of the tree. Traversing the tree upwards from that node will give us the longest path in the DAG already taking into account all previous tried paths.

Loop unrolling / function expansion

When all paths in the DAG were already tested, then the resulting tree is a one node tree where that node is marked as infeasible. Thus, we must make further loop unrolling of the loops and/or increase the depth of the function recursions. This loop unrolling/function expansion is done on the least possible number of loops/functions in order not to increase greatly the size of the DAG.
Each loop/function that needs to be unrolled/expanded have a vertex where the unrolling/expansion has stopped (in the case of Figure 2a is vertex 11). If that vertex was executed in any of the previous input vector generations then the corresponding loop/function will be unrolled/expanded. This avoids expanding the graph in vertices where execution has not reached yet. If execution of the program has reached some vertices and the paths are infeasible then that means that those vertices must be further expanded.

Input vector generator

The input vector generator is run to obtain the input vector that allow execution of the path given by the path in the tree. However, the path may be feasible or not.
If the path is feasible we obtain an input vector for that path. Then, we run the program to obtain the coverage. The coverage obtained will accumulate with the coverage of previous input vectors. If the accumulated coverage is greater or equal to the specified goal coverage then the algorithm ends and we have a set of input vectors that allow for a certain observability coverage. In case the specified coverage is not reached then we get the next path by rebuilding the tree. This rebuilding of the tree involves turning the vertices weight of the covered vertices into zero. This will force the tree to give us a new path that goes through some uncovered edges, since the tree always has the longest path.

TABLE 1. Program statistics.

Program	lines	input	decision points	statements
bitcount	12	integer	1	6
fibonacci	24	integer	3	13
strmatch	39	text	4	35
dijkstra	141	integer	15	99
huffman	203	text	17	193

If the path is infeasible then it means that at some decision point, the path that we want to follow diverges from a feasible path. The fact that the tree gave a path that is not feasible has to do with the fact that the building of the tree does not take into account the values of the condition variables. If it is possible to know the vertex where the paths diverge then we assume as infeasible the path just until that vertex (including the vertex). This information is used to rebuild the tree (as described above). If the vertex is not known then we make the whole path infeasible. This later solution will decrease the number of tree paths that we must obtain and consequently the number of runs of the input vector generator.

RESULTS

Our method to generate the minimum number of paths that achieve a specified observability coverage was implemented into a framework. The framework uses the methods described in the previous sections to fully automate the process of finding the input vectors given the program and the coverage we want to achieve. To demonstrate the feasibility of the method we used several example programs:

- **Bit counting,** which, given an integer, computes the number of bits that are one;
- **Fibonacci number,** which, given an integer, computes the corresponding Fibonacci number;
- **String matching,** which detects if a substring is present in a string;
- **Dijkstra,** which computes the shortest path in a graph;
- **Huffman coding,** which gives an Huffman coding given a string of characters.

Dijkstra and **Bit counting** belong to MiBench [10], a commercially representative embedded benchmark suite. The implementation of **Huffman coding** and **Fibonacci number** are found in Numerical Recipes in C [16]. **String matching** was implemented using the shift-and algorithm [15].

In Table 1 we have the statistics of the programs that we used as examples. In it we show the size of the programs in number of lines, the type of input, the number of decision points and the number of statements.

The examples were submitted to our framework to obtain the input vectors. The machine where we run the tests was an Intel(R) Pentium(R) 4 running at 3.2GHz with 1GB of physical memory.

In Table 2 we show the results we obtained. For each program tested we have the feasible paths that increased the observability coverage. For each of the feasible paths obtained we show its observability coverage and also the accumulated observability coverage. The size of the tree is also shown and the number of paths extracted from the tree before obtaining a feasible one. Also, for each path we show the accumulated CPU time to obtain the path.

In **bitcount**, we have a 100% observability coverage using just one path. That has to do with the fact that the program consists of a loop where by executing it once we get that all statements have some influence on the output.

In **fibonacci** (represented in Figure 1), the longest path is by executing the loop once. This path will give us 69% coverage. The other two paths will be for executing the conditions when we have the input value equal to 0 or 1. But this does not give us 100% observability coverage. The fourth path gives us an accumulated observability coverage of 100%. Note that the coverage for that path is greater than the one for the other paths. Since our method tries to find the longest path, that can be explained by the fact that when obtaining paths 1-3 we only unrolled the loop once. Looking at the example in Figure 1 and Figure 2 we can see that there are two statements inside the loop that will only be observable if the loop is executed once more.

In the **strmatch** program, the observability point is only executed when there is a match between the strings. The results we obtained are, in the first path, the coverage that corresponds to the matching of a string in its first character. In order to obtain 100% coverage we have to test the second character. Testing the string second character means that we expand the loop once more and end up obtaining 100% coverage with that path.

In **dijkstra**, we get 100% observability coverage with the first path. That can be explained by mention that this program finds a fixed set of shortest paths in a graph. Thus, the longest path will have several passages through the shortest path computation. By combining the coverage of those passages we obtain 100% observability coverage in the first path.

In **huffman**, we achieved 94% with an input vector that had a one character input string. We achieve 99% with two characters which caused the unrolling of some loops and expansion of some recursive function. To obtain 100% coverage we have to further unroll/expand the loops/functions. With that done we can obtain a new path whose coverage combined with the first or second path coverage give us 100% observability coverage. Note that the number of paths that are obtained from the tree before getting the feasible paths is larger than in the other examples. That has to do, as mentioned before, with the necessity of unrolling/expanding the loops/functions.

The results shown here confirm the feasibility of this methodology. The fact that the tested programs chosen are small allow us to know exactly why and how the results presented are what they are. The results also show small amount of CPU time which seems to indicates that this methodology can be scalable to larger real life embedded software programs.

CONCLUSIONS

We presented an observability coverage-directed vector generation method. In this method we address the problem of finding a minimal set of execution paths that achieve a user-specified level of observability coverage.

TABLE 2. Results of the tests.

Program	path	tree paths tried #	tree paths tried nodes	% coverage	accumulated % coverage	accumulated CPU time
bitcount	1st	1	7	100%	100%	0.00s
fibonacci	1st	1	15	69%	69%	0.00s
	2nd	3	21	23%	77%	0.00s
	3rd	4	22	15%	85%	0.00s
	4th	6	28	85%	100%	0.00s
strmatch	1st	1	36	87%	87%	0.00s
	2nd	6	222	100%	100%	0.06s
dijkstra	1st	1	88	100%	100%	0.00s
huffman	1st	12	3576	94%	94%	13.71s
	2nd	33	15240	99%	99%	152.93s
	3nd	84	32111	87%	100%	1415.08s

We model this problem building a tree representing the longest not yet tried paths. Thus in our greedy algorithm we get the longest feasible path and, if the coverage was not achieved, we refine our path search by modifying the existing tree or building a new one. We presented results that show the feasibility of our method. In any coverage-directed method, improving the so far obtained coverage can become a hard problem when we are trying to obtain a test to cover the last statements and reach 100% coverage. Our method guarantees the best possible convergence rate, as the next path to be computed to improve coverage is the one that, among the feasible, will maximize the coverage.

Our final goal is to implement an embedded systems observability coverage-directed co-validation method. Therefore, in the future we will integrate this method with a hardware observability coverage-directed method such as the one proposed by Fallah et al. [6].

REFERENCES

[1] F. Corno, P. Prinetto, M. Sonza Reorda, and D. e Inf. Testability analysis and ATPG on behavioral RT-level VHDL. *Procs. of the International Test Conference*, 753–759, 1997.

[2] J. Costa, S. Devadas, and J. Monteiro. Observability analysis of embedded software for coverage-directed validation. In *Procs. of the ICCAD*, 27–32, 2000.

[3] J. Davis et al. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*. Electronics Research Laboratory, College of Engineering, University of California, 2001.

[4] G. De Michell and R. Gupta. *Hardware/Software Co-Design*. Morgan Kaufmann, 2001.

[5] S. Edwards, L. Lavagno, E. Lee and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation and Synthesis. *Procs. of the IEEE*, 85(3):336–390, 1997.

[6] F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from HDL descriptions for observability-enhanced statement coverage. *Procs. of the 36th DAC*, 666–671, 1999.

[7] F. Fallah, S. Devadas, and K. Keutzer. OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Simulation. In *Procs. of the* 35^{th} *DAC*, 152–157, June 1998.

[8] N. Gupta, A. Mathur, and M. Soffa. Generating test data for branch coverage. *Procs. of the 15th IEEE International Conference on Automated Software Engineering*, 219–227, 2000.

[9] N. Gupta, A. P. Mathur, and M.Soffa. Automated test data generation using an iterative relaxation method. In *Procs. of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, 231–244, November 1998.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[11] I. Harris. Hardware/software covalidation. *Procs. of IEE Computers and Digital Techniques*, 152(3):380–392, 2005.

[12] M. Lajolo, M. Rebaudengo, M. Reorda, M. Violante, and L. Lavagno. Behavioral-level test vector generation for system-on-chip designs. In *Procs. of the IEEE International High-Level Design Validation and Test Workshop*, 21–26, 2000.

[13] S. Lee and J. M. Rabaey. A Hardware-Software Co-simulation Environment. In *Procs. of the International Workshop on Hardware-Software Codesign*, October 1993.

[14] D. Lettnin, M. Winterholer, A. Braun, J. Gerlach, J. Ruf, T. Kropf, and W. Rosenstiel. Coverage Driven Verification applied to Embedded Software. *VLSI, 2007. ISVLSI'07. IEEE Computer Society Annual Symposium on*, 159–164, 2007.

[15] U. Manber and S. Wu. Fast text search allowing errors. *Communications of the ACM*, 35(10), 1992.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1993.

[17] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Tech.*, 43(14):841–854, 2001.

[18] J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz, M. Inc, and T. Austin. Modeling design constraints and biasing in simulation using BDDs. *IEEE/ACM ICCAD. Digest of Tech. Papers*, 584–589, 1999.