Observability-based Coverage-directed Path Search using PBO for Automatic Test Vector Generation

José C. Costa José C. Monteiro
TU Lisbon, IST / INESC-ID
1000-029 Lisboa, Portugal
Email: {jose.costa, jcm}@inesc-id.pt

***Abstract*—In this paper, we address the problem of finding a minimal set of execution paths that achieve a user-specified level of observability coverage. Under this metric, a program statement is only considered covered if its execution has influence on some output. We use Pseudo-Boolean Optimization (PBO) to model the problem of finding the paths that are most likely to increase code coverage. Generated paths are then validated to check for feasibility. This methodology was implemented into a fully functional tool that is capable of handling real programs specified in the C language.**

I. Introduction

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.

In the case of software, the developed techniques are not directly applicable to embedded software (that interacts with hardware). The main reason is that research done in software compilation and validation techniques has been mainly directed to general-purpose software. The importance of embedded software has been recognized [10], and research done targeting general-purpose software is being retooled to address the problem of embedded software.

Embedded software testing has become more important with the dramatic increase of the size and complexity of the programs. This importance is even more critical since software programs are error prone. Complete path testing, which would give a 100% path coverage, is impractical. Testing only a small set of input values and a small set of paths is the solution. We are left with two problems: decide which set of paths need to be tested while guaranteeing a given confidence level; and determine which inputs need to be applied to the program to activate the selected paths.

In this paper, we address the first of these problems: given an embedded software program, find a minimal set of execution paths that guarantees a user-specified level of statement coverage. The method starts by modeling the problem to obtain the path with the greatest number of statements as a Pseudo-Boolean Optimization (PBO) problem [1]. This path is then validated against an input value generator [6] to test its feasibility, and if feasible its observability-based coverage is computed. The coverage obtained directs the choosing of the next path by changing the PBO problem accordingly. Paths are generated until the specified coverage is achieved.

Many of the co-validation fault models currently applied to hardware/software designs have their origins in either the hardware or the software domains [8]. A number of these models are based on the traversal of paths through a Control-Flow Graph (CFG) representing the system behavior. 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 CFG descriptions. Having the hardware description and the software description both in CFG 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 CFG-based methods can be applied to software and vice-versa. The method proposed is based on traversal of paths on a CFG and thus can be applied to either software or hardware high level languages. Additionally, the coverage metric we use is motivated by work on observability-based coverage metrics for hardware models described in a hardware description language [4] and afterwards by the same metrics applied for embedded software [2].

This paper is organized as follows. In Section II, we give an overview of the field of automated testing of embedded systems. Our method for obtaining the input vectors for observability coverage is presented in Section III. In Section IV, we present how we build the graph that includes the paths to be selected. How we model the problem of finding the longest path into a pseudo-boolean optimization problem is described in Section V. Some results are presented in Section VI. Finally, some conclusions and future work are presented in Section VII.

II. Related work

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 [14] 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 [5] input data for branch coverage, which consists of exercising all alternatives for every branch of the program, 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 [6] 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.

Another dynamic method [9] was proposed for branch coverage. The approach starts by executing a program for an arbitrary program input. The execution flow is monitored as the program is executed. For each executed branch, a search procedure decides whether the execution should continue through the current branch or an alternative branch should be taken if, for instance, the current branch does not lead to the execution of the selected statement. If an undesirable execution flow at the current branch is observed, then a real-valued function is associated with this branch. The function value depends on the branch predicates and the lesser the function value the more likely it is to execute that branch. Afterwards, function minimization search algorithms are used to automatically locate values of input variables which will change the flow of execution at the branch.

A method intended specifically for embedded software was presented in [11]. 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 model of a PowerPC microprocessor.

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. Recently the authors have proposed a method [3] that takes into account whether the statements executed have any influence on the program's output. That previous method uses a tree graph representation to compute the path that most likely would increase previous obtained coverage. In this paper we use a PBO formulation to compute the paths.

III. Proposed methodology

In this section, we make an overview of our method and present tools that are used in different steps.

A. Overview of the method

Our methodology to generate input test vectors is illustrated in Figure 1. The first step of the method is to determine a path which increases the current coverage the most. The path is then tested for its feasibility and, if feasible, we compute the coverage attained by running the program with the inputs determined to exercise the path. If the accumulated level of test coverage reaches the user-specified level, then we stop, having obtained a minimal set of test vectors for this coverage. If the accumulated test coverage was not achieved, or the tested path was infeasible, then we have to compute another path.

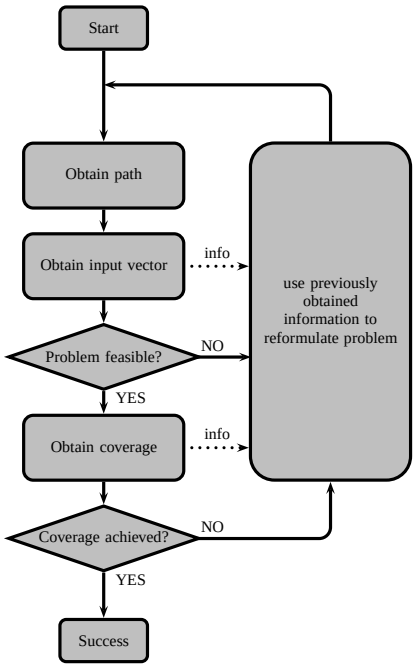

Fig. 1. Input test vectors generation methodology.

The focus of this paper is on determining the path that potentially increases the already calculated coverage. For that we use information obtained from the previous steps to reformulate the problem of finding the path. In the remaining steps (namely, computing the input test vector and evaluating its coverage) we use methods available in the literature.

In the rest of this section we state the methods we use in the problem we focus on, determining the next path. In the next section, we present how we integrate everything to obtain an observability-based coverage-directed method for software.

B. Pseudo-boolean optimization problems

Linear pseudo-boolean optimization (PBO) problems, also known as 0-1 integer linear programming problems, can be defined as follows,

$$\text{minimize} \quad \sum_{j \in N} c_j \cdot x_j \tag{1}$$

$$\text{subject to} \quad \sum_{j \in N} a_{ij} l_j \geq b_i, \tag{2}$$

$$x_j, l_j \in \{0,1\}, a_{ij}, b_i \in \mathbb{N}_0^+, i \in M,$$
$$N = \{1, \ldots, n\}, M = \{1, \ldots, m\}$$

where c_j is a non-negative integer cost associated with variable x_j, $j \in N$ and a_{ij} denote the coefficients of the literals l_j in the set of m linear constraints. A literal l_j denotes either a variable x_j or its complement $\bar{x}_j$. A literal will have value 1 if $l_j = x_j$ and $x_j = 1$ or $l_j = \bar{x}_j$ and $x_j = 0$. Otherwise it will have value 0.

In the case of non-linear pseudo-boolean expressions, they can be easily reduced to linear ones, since the literals have

value 0 or 1. In our method we use non-linear pseudo-boolean expressions, thus through out the rest of the paper we will be referring to the non-linear pseudo-boolean optimization problem as simply Pseudo-Boolean Optimization (PBO) problem.

Using PBO to model the problem of finding the path we can:

- specify a cost function that states the best statements that we want our path to execute in order to increase accumulated coverage;
- specify which statements that we want our path to have;
- specify which paths we do not want to execute (because they are infeasible or were already tested).

To obtain a PBO problem from the test program we first obtain its Directed Acyclic Graph (DAG) representation IV. The vertices of this DAG are then mapped into PBO variables.

The reason why we model the problem of finding the longest path with PBO is that while the first longest path can be obtained in linear time, the same is not true for the consequent longest paths. Because by inserting into the problem the fact that there are several path that already have been tested and we do not want to obtain them again, brings extra complexity to the problem, and thus a linear time solution is no longer possible.

C. Input vector generation

The solution to the PBO problem will give us a path. 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 [6]. 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 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.

If it is infeasible then the information of where in the path it became infeasible is added to the PBO problem (Section V).

D. Software observability coverage metric

In order to know if the desired coverage 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 and also of the conditions on which the branches depend. For that purpose, for each variable in the program there is a list of statements the variable depends on. When the execution

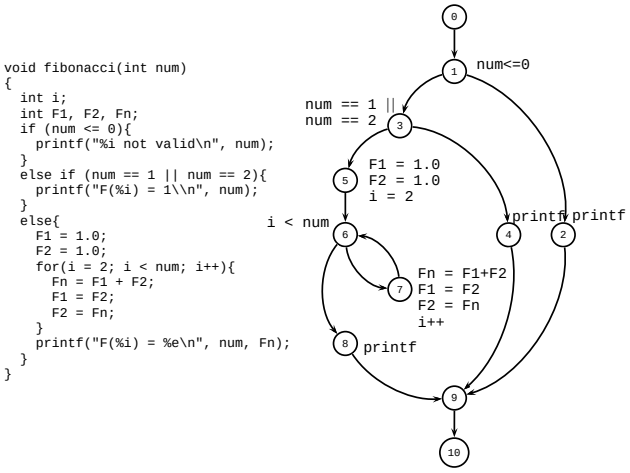

Fig. 2. C code for the Fibonacci function and its CFG.

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. In the case of branches that depend on conditions (e.g., if statement), all the statements in that branch will depend also on the variables of the condition.

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 the ones observable from that output.

IV. Program DAG

From the source program code, we extract multiple control-flow graphs (CFG), one for each function in the program (in Figure 2 we have the CFG of the Fibonacci function). The CFGs obtained are directed graphs, which can be cyclic if the functions have loops. The vertices in the CFGs 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 connect the CFGs as in the program, when we have a function call vertex we mark on the CFG vertex which CFG the function call corresponds to.

In order to simplify the computation of the PBO expressions, we build a new graph that starts in the CFG that corresponds to the main function. We traverse the CFGs, expanding function calls and loops along the way. In the end we get a DAG of the program (Figure 3). Note that we do not expand function calls and loops indefinitely. We just expand each once. 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

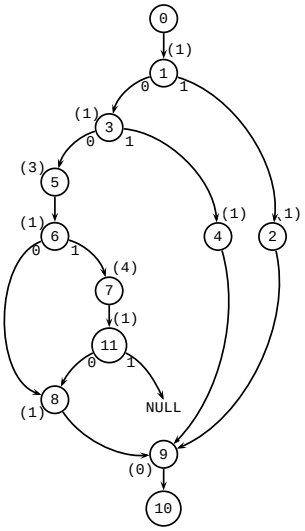

Fig. 3. Loop unrolling of the Fibonacci graph.

is not further expanded.

We refer to [3] for more details of function expansion and loop unrolling.

V. Pseudo Boolean Optimization

We model the problem of finding a path to increase our observability coverage as a pseudo-boolean optimization problem. 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.

All of these items can be modeled by PBO expressions. The PBO problem solution will give us the values of the boolean variables that indicate what the value of each program condition should be. Afterwards, we test that solution in the input vector generator. In the process of finding an input test vector we get information about the feasibility of the path or which was the vertex where the path became infeasible. This information will be later used in subsequent PBO problems.

A. Path constraints

In Section IV we described how we obtained a DAG from the CFGs of the functions. In order to obtain the PBO expressions we attach for each decision vertex in the DAG a boolean variable. Thus, if in the PBO solution we get a boolean variable with value 1, that means that we will follow the branch whose respective vertex condition is true. If we obtain two boolean variables which have values 1 and 0, then we will follow the path whose first condition is 1 and the second condition is 0. This means that we can specify a path as a product of literals.

Using the example on Figure 3 if we have the product of literals:

$$\overline{x_1}\,\overline{x_3}x_6\overline{x_{11}}. \tag{3}$$

that defines the path that goes through vertices 0,1,3,5,6,7, 11,8,9,10. This indicates that in vertices 1, 3 and 11 the boolean variables must have value 0 and in vertex 6 the boolean variable must have value 1.

In this notation if we want to explicitly not to go through some path then we equal its expression to 0. Similarly, if we want to always execute a certain path then we equal its expression to 1. If we want to go through vertices 1,3 and 5 we have,

$$\overline{x_1}\,\overline{x_3} = 1. \tag{4}$$

Solving this expression, and since the variables are boolean, we get as the only solution $x_1 = 0$ and $x_3 = 0$. As expected, this means that in order for the path to be executed we must have the conditions in vertices 1 and 3 equal to false.

In order to specify in the PBO problem which of the vertices that we want to execute, depending if they are already covered or not, we have for each vertex a PBO expression. This PBO expression represents the conditions necessary for its execution.

The algorithm to compute the boolean expressions for each vertex starts by ordering the vertices in a topological manner, to a list, such that when we reach a vertex, all of its ingoing vertices have already been processed. We go through that list when propagating the PBO variables. For each vertex we first test if the vertex is a condition vertex. If it is then we assign a PBO variable to it. Next, we check its ingoing vertices. If it is only one then we just copy the PBO expression to the vertex and append its PBO variable if the ingoing vertex is a condition. If there are more than one ingoing vertices then we must merge their PBO expressions. The rationale here is that if we have the same variable merging but in one of the ingoing vertices it is x and on the other it is $\overline{x}$ then this means that we are merging two paths that were disjoined in the vertex corresponding to the PBO variable. In that case that variable does not matter anymore. So we do not propagate it. Otherwise we propagate the variable. We do this until all the vertices are processed.

Armed with the PBO expressions for the execution of each vertex and the boolean variables of each condition vertex we can specify in our PBO problem the longest path, the paths that we do not want to execute and the vertices that we want to execute.

B. Cost function computation

In trying to find the longest path, we want to maximize the number of statements executed, thus defining the cost function of the PBO problem. Thus, we give weights to all the vertices in the graph (see Figure 3). The vertex weight multiplied by its PBO expression will give us the number of statements if that vertex PBO expression is equal to 1. Therefore, the PBO expression that we want to maximize is the sum of the products of the vertices weights with their PBO expression. In the case of the graph in Figure 3 we have:

$$\begin{aligned} max : 1 + 1x_1 + 1\overline{x_1} + 1\overline{x_1}x_3 + 3\overline{x_1}\,\overline{x_3} + 1\overline{x_1}\,\overline{x_3} + \\ 4\overline{x_1}\,\overline{x_3}x_6 + 1\overline{x_1}\,\overline{x_3}x_6 + 1\overline{x_1}\,\overline{x_3}, \end{aligned} \tag{5}$$

where the literals indexes correspond to the vertices numbers of the graph.

This expression alone can define the problem of finding the longest path in our DAG. Note that the solution to this problem will not give us a path that goes through all the vertices. That is impossible in this graph and we turn it impossible

in our problem when we propagate the PBO expressions the way we described in Section V-A. For instance, if the best option is to pass through edge 5 (with expression $\overline{x_1}\,\overline{x_3}$) we are automatically excluding vertices 2 (x_1) and 4 ($\overline{x_1}x_3$).

C. Avoiding certain paths

As stated before, when we build the DAG for the first time we only do one unrolling of the loop in order to have all vertices in the DAG. Thus, there are certain paths in the program that are not represented in the DAG. Therefore, when we extract the PBO problem from the DAG we have to take into account those paths. In the example of Figure 3, vertex 11 has one edge that can not be followed. So, in the PBO problem we have:

$$x_{11} = 0. \tag{6}$$

This guarantees that x_{11} will not be 1 and the solution given by the PBO problem will not contain that path. The same applies to function expansion.

Once we have obtained a path, we do not want to obtain that path again as a solution of the PBO problem. Thus, in the subsequent PBO problems we equal the PBO expression of that path to zero, thus, guaranteeing that the next solutions of the PBO problem will not contain that path.

D. Observability vertices

Since we are interested in observability coverage we want to execute at least one of the vertices that have observability points. Thus, when building the CFG and later the DAG we mark those vertices as observable. Later, when building the PBO problem we force the problem to give as solution one path that goes through at least one of those vertices. In Figure 3 we would have the expression:

$$x_1 + \overline{x_1}x_3 + \overline{x_1}\,\overline{x_3} >= 1, \tag{7}$$

where each product of literals correspond to a vertex.

Note that expressions 5, 6 and 7 form the complete initial PBO problem for the Fibonacci program.

E. Loop unrolling / function expansion

When the PBO problem is infeasible that means that there no more unexplored paths in the DAG. Thus, and assuming the specified coverage was not achieved, that means that some loop must be unrolled or some function must be expanded. 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.

F. Input vector generator

The input vector generator [6] is run to obtain the input vector that allow execution of the path given by the PBO solution. 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 reformulating the PBO problem. This reformulation involves turning the vertices weight of the covered vertices into zero. This will force the solution of the PBO problem to give a path that goes through some uncovered edges, since the PBO problem tries to maximize a cost function.

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 PBO problem gave a path that is not feasible has to do with the fact that the PBO problem 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 reformulate the PBO problem (Section V-C). If the vertex is not known then we make the whole path infeasible. This later solution will increase the number of PBO problems to solve and consequently the number of runs of the input vector generator.

TABLE I
PROGRAM STATISTICS.

Program	lines	input	decisions	statements
`fibonacci`	24	integer	3	13
`dijkstra`	141	integer	15	99
`huffman`	203	text	17	193

VI. 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:

- `fibonacci`, which, given an integer, computes the corresponding Fibonacci number;
- `dijkstra`, which computes the shortest path in a graph;
- `huffman`, which gives an Huffman coding given a string of characters.

`dijkstra` belongs to MiBench [7], a commercially representative embedded benchmark suite. The implementation of `huffman` and `fibonacci` are found in Numerical Recipes in C [13]. In Table I we have the statistics of these programs. 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 submited to our framework to obtain the input vectors. The machine where we run the tests was an Intel® Pentium®4 running at 3.2GHz with 1GB of physical memory. The PBO solver we used was `bsolo` [12].

In Table II 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 PBO problem necessary to achieve various coverages is presented. We show the number of its variables, constraints and literals. We also show the number of PBO problems necessary to find a feasible path. Also, for each path we show the accumulated CPU time to obtain the path.

TABLE II
RESULTS OF THE TESTS.

Program	path	PBO problem #	variables	constraints	literals	% coverage	accumulated % coverage	accumulated CPU time
`fibonacci`	1st	1	14	24	84	69%	69%	0.002s
	2nd	2	10	18	56	23%	77%	0.004s
	3rd	3	10	20	57	15%	85%	0.005s
	4th	5	11	22	65	85%	100%	0.007s
`dijkstra`	1st	1	87	146	1306	100%	100%	0.002s
`huffman`	1st	12	440	745	6675	93%	93%	39.644s
	2nd	74	1087	2019	24157	88%	100%	1688.3s

In `fibonacci` (represented in Figure 2), 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 2 and Figure 3 we can see that there are two statements inside the loop that will only be observable if the loop is executed once more. In fact, we do not show it in the table, but the PBO problem #4 results in infeasible and thus the loop must be expanded once more.

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 93% with an input vector that had only one value. To obtain 100% coverage we have to further unroll the loops and also expand some recursive functions. With that done we can obtain a new path whose coverage combined with the first path coverage give us 100% observability coverage. Note that the number of PBO problems 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 obtained confirm the validity of this methodology. The small size of the benchmark programs allow us to understand and confirm the results obtained. One concern is that, although the size of PBO problems remain fairly small, indicating that this methodology is scalable to larger real life embedded software programs, the solvers seem to have a harder time obtaining a solution as we add paths to avoid (either because they have been covered or have been found to be infeasible). We are currently investigating why these few additional clauses may cause this effect.

VII. 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. We model this problem as a PBO problem where we try to find the longest feasible path. Thus in our greedy algorithm we get the longest feasible path and, if the coverage was not achieved, we refine our path search by inserting additional constraints in our PBO problem. Initial results demonstrate the validity of the method. While the sizes of the PBO remain well within reach of existing solvers, the solvers we have tried have some difficulty computing some of the solutions. This is an issue we are focusing our attention on.

Compared to the authors' previously mentioned work [3], this method has the potential to have the input vector generation phase integrated with the PBO formulation since both methods are based on solving a linear constraints problem.

Also, in the future, we will be applying this method to hardware descriptions in high-level description languages with the objective of obtaining a coverage-directed co-validation method for embedded systems.

REFERENCES

[1] E. Boros and P. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.

[2] J. Costa, S. Devadas, and J. Monteiro. Observability analysis of embedded software for coverage-directed validation. In *Proceedings of the Intl. Conference on Computer Aided Design*, pages 27–32, 2000.

[3] J. Costa and J. Monteiro. Computation of the minimal set of paths for observability-based statement coverage. In *15th International Conference on Mixed Design of Integrated Circuits and Systems, MIXDES 2008*, pages 587–592, 2008.

[4] F. Fallah, S. Devadas, and K. Keutzer. OCCOM: Efficient Computationa of Observability-Based Code Coverage Metrics for Functional Simulation. In *Proceedings of the* 35^{th} *Design Automation Conference*, pages 152–157, June 1998.

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

[6] 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*, pages 231–244, Lake Buena Vista, Florida, United States, November 1998.

[7] 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.

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

[9] B. Korel. A dynamic approach of test data generation. In *Proceedings of the Conference on Software Maintenance*, pages 311–317, 1990.

[10] E. A. Lee. Embedded software. *Advances in Computers*, 56:56–97, 2002.

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

[12] V. Manquinho and J. Marques-Silva. Effective lower bounding techniques for pseudo-Boolean optimization. In *Proceedings of the Design, Automation & Test in Europe Conference*, pages 660–665, 2005.

[13] 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.

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