Input Generation for Path Coverage in Software Testing

José C. Costa
IST/INESC
jccc@algos.inesc-id.pt

José C. Monteiro
IST/INESC
jcm@inesc-id.pt

ABSTRACT

The most common approach to checking correctness of a hardware or software design is to verify that a description of the design has the proper behavior as elicited by a series of input stimuli. In the case of software, the program is simply run with the appropriate inputs, and in the case of hardware, its description written in a hardware description language (HDL) is simulated with the appropriate input vectors.

Complete software functional testing in the sense of subjecting the program to all possible input values is impractical. Path testing corresponds to the input stimuli of the program exercising a selected set of paths through it. But total path testing is also impractical. Testing only a small set of input values and a small set of paths is the solution. The problem is to know which set of paths need to be tested and which inputs need to be applied to the program.

In our work, we develop a method for obtaining the inputs that allow an embedded software program written in a high-level language to execute a specified path. Given the path to be executed, our method extracts from statements in that path a set of linear programming (LP) constraints. The solution of the LP problem gives us the necessary inputs or sequence of inputs to exercise that path.

The integration of our method with a HDL functional vector generation will give us a complete input vector generation methodology applicable to embedded systems.

1. 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.

Research done in software compilation and validation techniques has been mainly directed to general-purpose software, and in most cases the developed techniques are not directly applicable to embedded software (that interacts with hardware). The importance of embedded software has now been recognized, and research done targeting general-purpose software is being retooled to address the problem of embedded software [10]. Testing is an integral part of software development. It needs to be an integral part of programming. Completing a program should consist of an iterative process of programming, testing, correcting, testing, and signing off the program as having met predetermined test criteria.

If it were practical, or even possible, to test exhaustively, we would stop when all possible tests had been executed and passed. But testing is expensive, and in all but the simplest systems, exhaustive testing is impossible in a finite time. Given that we can never prove perfection, we want to avoid testing beyond the point of significantly diminished returns - while still achieving the desired level of confidence in our product. So we need to choose test cases carefully, to achieve the necessary coverage while avoiding replication.

In path testing, a selected set of paths of the program is exercised through a set of input stimuli. Complete path testing, which would give a 100% coverage, is also impractical. Testing only a small set of input values and a small set of paths is the solution. The problem is to decide which set of paths need to be tested, and which inputs need to be applied to the program to activate those paths.

Several approaches exist for obtaining test cases. Dynamic test data generation [9] is one. In dynamic test data generation an instrumented version of the program is executed. The execution flow is thus monitored. This information helps in the guidance of the search for new and better inputs to cover a specified program path. Another approach is evolutionary testing [16, 13] where the test cases are sought by formulating an optimization problem.

Our work is motivated by recent work on functional vector generation for an hardware description language (HDL) [4]. In this work an algorithm is presented that is an integration of linear programming (LP) techniques and 3-satisfiability (3-SAT) checking [6]. Given a path in the HDL model, an input stimuli that exercises that path is obtained.

In our work, we develop a method for obtaining the inputs that allow an embedded software program written in a high-level language to execute a specified path. Given the path to be executed, our method extracts from statements in that path a set of LP constraints. The solution of the LP problem gives us the necessary inputs or sequence of inputs to exercise that path.

This process can be repeated until all the paths selected for testing have been exercised. The paths to be executed depend on the test thoroughness defined by the designer and on the type of metric to measure that thoroughness. There are several metrics that can be used, such as path coverage, branch coverage, statement coverage [1], PIE analysis [15], impact analysis [5] or code coverage [2] to name a few.

In the case of embedded systems its validation 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 [7, 8, 11, 12, 14].

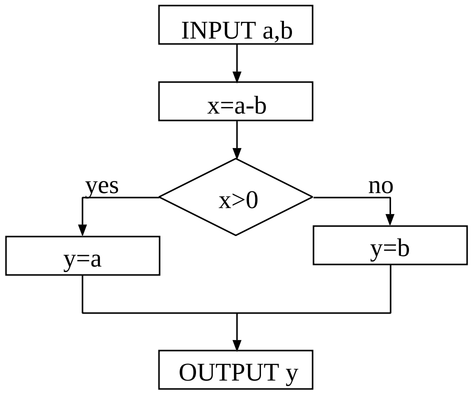

Figure 1: Example flowgraph for the MAX function.

Integrating our method with a vector generator for hardware will give us a complete vector generation methodology applicable to embedded systems. This should not be too hard since our method is already based on a functional vector generator for hardware.

This paper is organized as follows. In Section 2, we give an overview of the software testing field and the simulation vector generation from an HDL description. Our method for obtaining the input vectors for path coverage is presented in Section 3. Some examples are presented in Section 4. Finally, some conclusions and future work are presented in Section 5.

2. RELATED WORK

2.1 Software testing

The most commonly used methods for software testing are based on path testing. To give a measure of the test thoroughness using path testing, several metrics have been developed. These metrics make use of the concept of control flowgraph.

2.1.1 Control flowgraph

A control flowgraph is a graphical representation of a program's control structure [1]. A control flowgraph consists of processes, decisions, and junctions. A process is a sequence of statements such that if any statement is executed, then all other statements are executed. Thus, a process block is a sequence of statements uninterrupted by either decisions or junctions. A decision is a program point at which the control flow can diverge. A junction is a point in the program where the control flow can merge. Figure 1 shows the flowgraph of a program.

A path in the program is a sequence of statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, once or more than once.

2.1.2 Software path testing

Path testing corresponds to the input stimuli of the program exercising a selected set of paths through it. There are several metrics that can give us a measure of the test thoroughness for some input stimuli [1]. The most important ones are path, statement and branch coverage.

Path coverage is the most complete of all the path testing methods. We achieve 100% path coverage when every possible path in the program is executed. This means that from the beginning of the program all possible ways of getting to the end were followed and executed. Reaching 100% path coverage is very often impractical due to the great number of possible paths.

Statement coverage targets the execution of every statement in the program. Although this metric is easily achieved, it is a very weak one. Many possible buggy conditions are not tested.

Between the two, in terms of test thoroughness, we have branch coverage. Branch coverage consists of exercising all the alternatives of every branch. This metric is only a little better than statement coverage. It executes every statement and also tests every branch in each condition, including those branches that do not have any statement.

Variants of branch coverage such as multicondition coverage and loop coverage are also used as coverage metrics. In multicondition coverage every condition is required to take every possible value. Loop coverage requires that every loop is executed zero, one or two times.

2.2 HDL functional vector generation

Sensitization of a program path is not very different from that of a circuit path. In hardware, sensitizing a path implies that the value at the input of the path should affect the value at the output of the path. In software, sensitizing a path means that the value at the input of the path will permit the execution of every statement in that path.

Sensitization of a circuit path corresponds to sensitize every single module in that path. In the case of software path sensitization it corresponds to sensitize every single statement in that path.

The algorithm to circuit path sensitization proposed by Fallah et al [4] is an hybrid algorithm for satisfiability checking that seamlessly integrates LP feasibility and 3-satisfiability checking. This integration is necessary due to the correlation between word-level variables and boolean variables.

The word-level variables and its constrains are specified as a linear programming problem. A linear program (LP) is a problem that can be expressed as:

$$\begin{array}{llll} \text{minimize} & cx & \text{subject to} & Ax = b \\ & & & x >= 0 \end{array}$$

where x is the vector of variables to be solved for, A is a matrix of known coefficients, and c and b are vectors of known coefficients. The expression cx is called the objective function, and the equations $Ax = b$ are called the constraints. The constrains can also be expressed as $b1 <= Ax <= b2$ in most LP solvers.

The 3-SAT checking consists in solving a boolean formula which is in the 3-conjuntive normal form (3-CNF). A boolean formula is in 3-CNF if it is the AND of clauses of ORs of exactly 3 variables or their negations. This boolean formula is satisfiable if there is some assigment of the values 0 and 1 to its variables that causes it to evaluate to 1.

The algorithm that can be applied to integrated hardware and software testing consists in:

1. write sensitization requirements on intermediate signal values. For example in the case of an AND gate, the side inputs of the path must be set to 1.

2. for every module in the circuit a set of module-input module-output relationships is written. In the case of logical gates, 2-SAT or 3-SAT clauses are written. In the case of word-level operators, LP constraints are written.

3. solve the satisfiability problem that corresponds to the conditions obtained in 1 and 2.

For every module in the circuit there is a corresponding set of LP constraints.

The modules corresponding to addition or subtraction are translated into LP constraints like in these examples:

$$\begin{array}{rr} C = A + B: & A + B - C \leq 0 \ \wedge \ A + B - C \geq 0 \\ C = A + k: & C - A \leq k \ \wedge \ C - A \geq k \end{array} \tag{1}$$

In the case of scalar multiplication:

$$C = A \times k: \quad C - kA \leq 0 \ \wedge \ C - kA \geq 0 \tag{2}$$

In the case of $c = (A > B)$ where c is a boolean variable we have:

$$A - B + U(1 - c) \geq 1 \ \wedge \ A - B - Uc \leq 0 \tag{3}$$

with U is equal to 2^n, where n is the maximum number of bits in A or B. The LP constraints for the $\geq$, $<$ and $\leq$ are similar.

The integer multiplication can be decomposed into linear operators. $Z = X \times Y$ is decomposed into:

$$Z \geq \sum_{i=0}^{n-1} 2^i p_i \qquad Z \leq \sum_{i=0}^{n-1} 2^i p_i \tag{4}$$

$$p_i - Ux_i \leq 0, \qquad 0 \leq i < n \tag{5}$$

$$p_i + U(1 - x_i) - Y \geq 0, \qquad 0 \leq i < n \tag{6}$$

$$0 \leq p_i < Y, \qquad 0 \leq i < n \tag{7}$$

LP expressions for the modulus operator and integer division can also be obtained.

The algorithm then performs a satisfiability search on the 3-SAT clauses where boolean variables are set to $0, 1$. A search is also done on the LP constraints. The constraints and the 3-SAT clauses are then modified accordingly and a new search begins.

Since our method is applied to software, we will have very few boolean variables, and in most cases, most of these are conditions obtained from comparisons between other types of variables. Thus, we do not have 3-SAT problem to solve. All we have are LP constraints, where the boolean variables are treated as LP variables restricted to the values 0 or 1.

3. INPUT GENERATION FOR PATH COVERAGE

3.1 Overview

The proposed method of input generation for path coverage is done in three steps. First, we obtain all the LP constraints for the source program, with additional information on the control structure of the program. Each statement in the source program corresponds to one or more constraints. Second, we specify the path that we want to exercise. Third, with the path specification and the set of all constraints for the program, we filter only the constraints relevant for the execution of the selected path. Then, this subset of constraints is applied to a LP solver to obtain the value of all the input variables in the program.

3.2 LP constraints of the program

To obtain the LP constrains from the source program we parse it extracting the constrains from all the source program lines. Along with these constrains we maintain the entire control structure of the source program.

This extraction, despite using a different syntax, is a mapping, line by line, of the program source file. For each line in the source file, we end up with one or more LP conditions. These gives us, in this first step, the conditions for all the lines of the source file plus additional information regarding functions, if-then-else conditions and loops. All this information is then used by a parser to extract only the conditions necessary to execute specified branches in a path. This necessary conditions depend on the path specified by the designer.

The parser used was c2c, which is a public-domain software program. c2c works by constructing an Abstract Syntax Tree (AST) of a C program. The AST can then be manipulated in several ways, such as adding or deleting nodes in it. Finally, after the AST has been modified, the c2c tool produces a C program for that new AST.

3.2.1 Expressions

Before we extract the LP constraints from the source program, we parse the source file and substitute every complex expression by simpler ones, *e.g.*,

```
a = b + c + d; ⇒ temp1 = b + c;
                 a = temp1 + d;
```

and,

```
if (a < b + c) ... ⇒ temp1 = b + c;
                     temp2 = a < temp1;
                     if (temp2) ...
```

In the second example, we could leave the condition

```
if (a < temp1)
```

but using the method shown above we guarantee the same treatment for all expressions regardless of the fact that they are assignments, conditions, arithmetic operations, function calls, etc.

At the same time that we are obtaining the simpler expressions, we extract all the LP constraints.

3.2.2 Assignments

Each time a variable is assigned in the input program, it must be a different variable in the LP program. Despite being the same variable in the source program, the fact is that it has different values when considering the different time instants of its assignment. So, each time an assignment is made a LP variable is needed. The variable will correspond to the program source variable and to the time instant of its assignment. Thus, in the LP solver we have different variables that correspond to the same variable in the source program only for different time instants.

If we have,

```
a = b + c;
a = a + 1;
```

we get,

```
a_1 - b_1 - c_1 = 0;
a_2 - a_1 = 1;
```

When we solve this problem we get in a_1 the initial value of a and in a_2 the final value of a.

3.2.3 Conditions

As shown before, conditions are transformed in this way:

```
if (a < b) ...  ⇒ temp1 = a < b;
                   if (temp1) ...
```

Thus, for this branch to be executed we have the following LP constraints:

```
a_1 - b_1 + (INT_MAX+1)×temp1 <= INT_MAX;
a_1 - b_1 + (INT_MAX+1)×temp1 >= 0;
```

plus the conditions that specify that the LP variables a_1 and b_1 are integers,

```
int a_1;
int b_1;
```

and the conditions that specify that the LP variable $temp1$ is boolean,

```
temp1 >= 0;
temp1 <= 1;
int temp1;
```

3.2.4 *Control expressions*

The true and false conditions from the if-then-else expressions or from the loops must also be translated into LP constraints. Both true and false constraints are specified in the first extraction of the LP conditions along with the control structure of the program. Choosing which one will be solved when we solve the LP problem is done when we specify the path to be executed.

If the path chosen has several executions of a loop then that loop is unfold as many times as it is executed. When the loop is unfold the LP conditions are extracted as explained before.

3.2.5 *Functions*

Since different functions can have variables with the same name, we have to guarantee that the name of the LP variable includes the information regarding the function. Thus, in the extraction we guarantee that it is divided in functions and inside each function we have the LP conditions for all the branches of that function. As stated earlier, this extraction has all the information regarding the control structure of the source program.

3.3 Specifying the execution path

In this second step, the path to be exercised is specified. This specification is done by defining all the branches to be executed. The specified paths are chosen depending on the type of coverage the designer wants. For every path specified, we obtain a LP problem and consequently a set of input values that allow the program to execute the path. The process of specifying the execution path is done manually. In the future we plan to do this automatically by specifying the desired coverage level [2].

3.4 Obtaining the input test vectors

With the path specification and all the LP constraints for the input program, we get the LP constraints for that specific path. Those LP conditions that we obtained in the extraction are saved into a file. Next, those conditions are applied to a LP solver and the input values for the program that allow the execution of the selected path are obtained.

3.4.1 *Parsing the LP file*

The file with the constraints is parsed according to the branch specification. For each branch, all the LP constraints for that branch are extracted along with the constraints corresponding to the condition that allows the branch to be executed. The constraints in the file are almost ready to be submitted to a LP solver. Two issues need to be taken into account before the set of LP constraints becomes ready to be sent to a LP solver:

- for every variable, an index is added. This index corresponds to the instant of assignment of the variable. When a variable is assigned, its index is incremented;
- for every variable, the name of the function to which it corresponds is added to its name. This allows for the same variable name in different functions;
- whenever a function is called, LP conditions are added to reflect the substituting of actual for formal parameters.

When the parsing of the file is complete, we end up with the LP constraints. Next we submit those constraints to a LP solver.

3.4.2 *Solving the LP conditions*

For solving the LP conditions, we are using `lp_solve` which is a Mixed Integer Linear Program solver based on the Simplex algorithm [3].

If the problem is feasible, we obtain all the values of all variables in all functions and in all time instants of the source program execution. We can then select the variables that depend only on external values from the source program. Those are the input values for the program that allow the execution of the selected path.

4. RESULTS

To show in more detail how our method works we present an example using a string match program. At the end of this section we present some statistics for some of the examples tested.

4.1 String matching

The string match program reads a stream of characters from a file and detects the occurrence of a specific string. The program activates the output only when there is a match. In the example that follows, whose code is depicted in Figure 2, the string that we are after is 'ack'.

Suppose that we want to execute line 13. There is a large number of paths that include line 13, but to make things simple we choose the shorter path. We want the path that executes lines 7, 8, 10, 11, 12 and 13. We do not take into account variable f since it is only an index to a file. The relevant value is the value read from the file, which in this case is the variable c. So, the following conditions must be met:
line 8: i_0 = 0;
line 10: c_0 != -1;
line 11: i_0 = 0;
line 12: c_0 = 97;
where the index of the variables corresponds to the index of the assignment, the value -1 corresponds to the `EOF` and the value 97 corresponds to the character 'a'. The fact that we have the condition in line 8 will become clear in the next example. The LP constraints for these conditions are (see Sections 2 and 3):

line 8:

```
i_0 = 0;
```

line 10: since we do not have the inequality operator in the LP constraints accepted by the LP solver, we have to transform this into:

```
 1: main()
 2: {
 3:   FILE *f;
 4:   char c;
 5:   int i;
 6:
 7:   f = fopen("string.dat", "r");
 8:   i = 0;
 9:
10:   while((c = fgetc(f)) != EOF ){
11:     if (i == 0){
12:       if (c == 'a')
13:         i = 1;
14:     }
15:     else if (i == 1){
16:       if (c == 'c')
17:         i = 2;
18:       else if (c == 'a')
19:         i = 1;
20:       else
21:         i = 0;
22:     }
23:     else if (i == 2){
24:       if (c == 'k')
25:         printf("Found String\n");
26:       else if (c == 'a')
27:         i = 1;
28:       else
29:         i = 0;
30:     }
31:   }
32: }
```

Figure 2: C code for the string matching program.

```
tmp1 = (c_0 > -1);
tmp2 = (c_0 < -1);
tmp3 = tmp1 || tmp2;
tmp3 = 1;
```

In LP constraints we get:

```
c_0-(INT_MAX+1)temp1 >= -INT_MAX-1;
c_0-(INT_MAX+1)temp1 <= -1;

c_0+(INT_MAX+1)temp2 <= INT_MAX-1;
c_0+(INT_MAX+1)temp2 >= -1;

tmp3-tmp1 >= 0;
tmp3-tmp2 >= 0;
-tmp3+tmp1+tmp2 >= 0;

tmp3 = 1;
```

and finally, we have to state that tmp1 and tmp2 are boolean:

```
tmp1 >= 0;
tmp1 <= 1;
tmp2 >= 0;
tmp2 <= 1;
int tmp1;
int tmp2;
```

For lines 11 and 12 we have:

```
i_0 = 0;
c_0 = 97;
```

Solving this LP problem we get:

```
i_0 = 0
c_0 = 97
tmp1 = 1
tmp2 = 0
tmp3 = 1
```

Examining the results, we see that for the execution of the specified path we have to have as an initial condition the value of c to be 'a'. The remaining results are of no interest.

Let us consider another example. Assume we want to execute line 17. We could choose the path that executes lines 7, 8, 10, 15, 16, 17. The following conditions are necessary to execute that path:
line 8: i_0 = 0;
line 10: c_0 != -1;
line 11: i_0 != 0;
line 15: i_0 = 1;
line 16: c_0 = 99;

This gives us the following LP constraints corresponding to the conditions above:

```
i_0 = 0;

c_0-(INT_MAX+1)temp1 >= -INT_MAX-1;
c_0-(INT_MAX+1)temp1 <= -1;
c_0+(INT_MAX+1)temp2 <= INT_MAX-1;
c_0+(INT_MAX+1)temp2 >= -1;
tmp3-tmp1 >= 0;
tmp3-tmp2 >= 0;
-tmp3+tmp1+tmp2 >= 0;
tmp3 = 1;
tmp1 >= 0;
tmp1 <= 1;
tmp2 >= 0;
tmp2 <= 1;
int tmp1;
int tmp2;

i_0-(INT_MAX+1)temp4 >= -INT_MAX;
i_0-(INT_MAX+1)temp4 <= 0;
i_0+(INT_MAX+1)temp5 <= INT_MAX;
i_0+(INT_MAX+1)temp5 >= 0;
tmp6-tmp4 >= 0;
tmp6-tmp5 >= 0;
-tmp6+tmp4+tmp5 >= 0;
tmp6 = 1;
tmp4 >= 0;
tmp4 <= 1;
tmp5 >= 0;
tmp5 <= 1;
int tmp4;
int tmp5;

i_0 = 1;

c_0 = 99;
```

Solving this LP problem we get an answer indicating that this is an infeasible problem. We could see that just by looking at the expressions for each line. The value i could not be equal to 0, different from 0 and equal to 1 at the same time. From this example it becomes clear that the condition for line 8 needs to be there to ensure that some invalid branch is not executed.

Thus, a good path to execute line 17 would be path 7, 8, 10, 11, 12, 13, 10, 15, 16, 17. In this case we would get two different LP variables for the variable i. One with the value 0, corresponding to the first run of the loop and another with value 1 corresponding to the second run of the loop. Also in this case we would have two different values to c. The first one corresponding to the character 'a' and the second one corresponding to 'c'.

4.2 Statistics for some examples

We present results of our method using three examples. One of the programs is the string matching shown before, one computes Fibonacci numbers without using recursion and one is a program that simulates an elevator with five floors. All three were implemented using the C language.

In Table 1 we present some statistics for the three programs. It gives us the number of lines of the source program and the number of LP constraints after parsing the source program.

Table 1: Size of the programs in number of lines and number of constraints.

Program	# lines	# LP constraints
String match	42	216
Fibonacci	28	72
Elevator	298	1377

In Table 2 we present the complexity of the LP problem and the CPU time to solve it. All reported CPU times are in seconds and were obtained on a Sparc Ultra 60 with 1G of main memory. In the case of the program for string matching the results correspond to a 100% statement coverage. In the case of the Fibonacci program the results shown correspond to executing the main loop an hundred times.

Table 2: Size of the LP problems.

Program	# LP constraints	# LP variables	CPU (s)
String match	602	114	0.51
Fibonacci	1261	612	7.13

5. CONCLUSIONS

We developed a method for obtaining the inputs that allow an embedded software program written in a high-level language to execute a specified path. This is accomplished by constructing a set of LP constraints with all the conditions that are necessary for the different branches is that path to be exercised. The solution to this LP problem gives us the input variables, or sequence of variables, we need.

In this version, the specification of the paths to be executed is still performed manually. The aim of this research is to have in the future a methodology for automatic path selection, guaranteeing some specified code coverage based on a given metric.

The final objective is to integrate this software validation method with HDL functional vector generation to give us a complete input vector generation methodology applicable to embedded systems.

6. REFERENCES

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinhold, New York, second edition, 1990.

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

[3] G. B. Dantzig. Application of the Simplex Method to a Transportation Problem. *Activity Analysis and Production and Allocation*, pages 359–373, 1951.

[4] F. Fallah, S. Devadas, and K. Keutzer. Functional vector generation for HDL models using linear programming and 3-satisfiability. In *Design Automation Conference*, pages 528–533, 1998.

[5] T. Goradia. Dynamic Impact Analysis: A Cost Effective Technique to Enforce Error Propagation. In *Proceedings of Int'l Symposium on Software Testing and Applications*, March 1993.

[6] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for Satisfiability (SAT) Problem: A Survey. In *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem*, volume 35, pages 19–152. American Mathematical Society, 1997.

[7] R. K. Gupta, C. N. C. Jr, and G. D. Micheli. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Proceedings of the Design Automation Conference*, June 1992.

[8] A. Kalavade and E. A. Lee. Hardware/Software Co-design Using Ptolemy - a Case Study. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1992.

[9] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[10] E. A. Lee. Embedded Software - An Agenda for Research. ERL Technical Report UCB/ERL M99/63, University of California, Berkeley, CA, USA 94720, December 1999.

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

[12] J. Rowson. Hardware/Software Co-simulation. In *Proceedings of the Design Automation Conference*, pages 439–440, 1994.

[13] R. Sagarna, J. A. Lozano, R. Murga, and L. M. Gonzlez. Dealing with software testing via estimation of distribution algorithms: a preliminary research. In *Proceedings of the Second Spanish Conference on Metaheuristics, Evolutive and Bioinspired Algorithms*, pages 70–77, Spain, 2003.

[14] K. ten Hagen and H. Meyr. Timed and Untimed Hardware/Software Cosimulation: Application and Efficient Implementation. In *Proceedings of the International Workshop on Hardware-Software Codesign*, October 1993.

[15] J. M. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[16] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.