# Observability Analysis of Embedded Software for Coverage-Directed Validation

José C. Costa
**IST/INESC**

Srinivas Devadas
**MIT**

José C. Monteiro
**IST/INESC**

**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. In coverage-directed validation, coverage metrics are defined that quantitatively measure the degree of verification coverage of the design.**

**Motivated by recent work on observability-based coverage metrics for models described in a hardware description language, we develop a method that computes an observability-based code coverage metric for embedded software written in a high-level programming language. Given a set of input vectors, our metric indicates the instructions that had no effect on the output. An assignment that was not relevant to generate the output value cannot be considered as being covered. Results show that our method offers a significantly more accurate assessment of design verification coverage than statement coverage. Existing coverage methods for hardware can be used with our method to build a verification methodology for mixed hardware/software or embedded systems.**

## I. 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 result 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.

Due to their heterogeneity, embedded systems pose several new problems that, only recently, have begun to be tackled. One of them is the specification problem. The specification language has to assume a model of computation for interacting hardware and software components. It is fairly common to support the models of computation with language extensions or entirely new languages. The language used can be specific to embedded systems, such as Esterel [3], Lustre [9], Signal [2], or Argos [13], among others. However, these languages have serious drawbacks. Acceptance is low, platforms are limited, support software is also limited, and legacy code must be translated or entirely rewritten. Currently, multiple general-purpose languages are used, for example, hardware languages such as VHDL, are used to describe hardware models, and software languages such as C, are used to describe embedded software code. The lack of a uniform specification increases the difficulty of the embedded system validation problem.

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. As mentioned above, 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 [8], [10], [12], [15], [16].

Research done in software compilation and validation techniques has been mainly directed at 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 [11]. On the other hand, embedded software is becoming very complex. The demand for more elaborate functionality is making it much more difficult for a single engineer to accomplish the validation task by manually checking tens of kilobytes of assembly code. As a result, efficient automated validation techniques are necessary that give the engineer a measure of confidence in the correctness of the software.

Our focus here is on coverage-directed validation, wherein coverage metrics are defined that quantitatively measure the degree of verification coverage of the design, be it hardware, software or a mixture of both. In this paper we propose a new metric that gives a measure of the instruction coverage in the software portion of the embedded system. Our metric is based on observability, rather than on controllability. Given a set of input vectors, our metric indicates the instructions that had no effect on the output. An assignment that was not relevant to generate the output value cannot be considered as being covered. Then, the designer or programmer, by looking at the statement coverage, can add more tests until all statements have some effect on the program output.

Our work is motivated by recent work on observability-based coverage metrics for hardware models described in a hardware description language [6]. Our results show that our method offers a significantly more accurate assessment of design verification coverage than statement coverage. Existing coverage methods for hardware can be used with our method to build a verification methodology for mixed hardware/software or embedded systems.

In Section II, we give an overview of the software testing field. Our metric for software coverage is presented in Section III. Section IV describes the implementation of our coverage metric. Preliminary results are presented in Section V. Finally, some conclusions and future work are presented in Section VI.

## II. Software testing

Software testing has become more important since the size and complexity of the programs increased in a dramatic way. This importance is even more critical since software programs are error prone. One of the ways to handle this is to test the software. On the other hand, proving that a program does not have bugs is practically and theoretically impossible. It is very difficult to prove that even a section of a given software program works, mainly because every possibility has to be tested in order to guarantee that the software has no errors. Since it is impossible to design a test that exercises every possible path in the program, several metrics have been developed to give a measure of the test thoroughness. But first let us present some concepts on the control flowgraph of

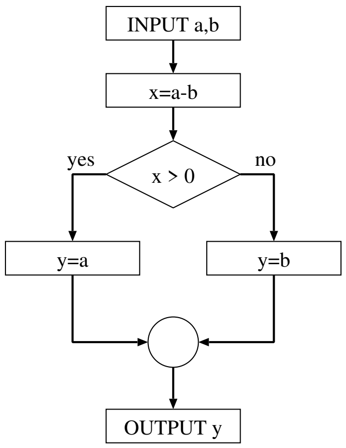


Fig. 1. Example flowgraph for the MAX function.

a program.

### A. 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.

### B. Software path testing

The most commonly used methods for software testing are based on 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. Further, some paths may be false, i.e., can never be executed.

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.

All these coverage metrics only take into account the activation of some path, i.e., controllability. They do not say anything about observing on the outputs the effects of those activations. Hence, observability is neglected in these path coverage metrics. We say that we have observability when, besides executing some statement, the result of that execution reaches some output. Controllability without observability is not very useful. Activating a statement does not mean that it has been verified if it is not observed at some output.

The path coverage metric will satisfy observability requirements if paths from program inputs to program outputs are exercised and the values of variables are such that the erroneous value is not masked. However, the path coverage metric does not explicitly evaluate whether the effect of an error is observable at an output.

### C. Observability metrics

Observability is taken into account in metrics such as sensitivity and impact analysis.

PIE analysis was proposed by Voas [17] to predict a statement's ability to cause program failure if the statement were to contain a fault. The program inputs are selected at random consistent with an assumed input distribution. The analysis gives as results, an estimation of the frequency with which inputs execute the statement, an estimation of the frequency with which mutants of the statement created altered data states and an estimation of the frequency with which altered data states cause a change on the program's output.

Goradia [7] introduced impact analysis. Impact analysis estimates impact strengths of all entity instances in an execution, in a time proportional to the execution time. The impact strength of a statement or variable $x$ serves as a quantitative measure of the error-sensitivity of the paths from $x$ to the output.

These two metrics both deal with the introduction of some error in some part of the program and compute the probability of that error reaching the output. The metric we propose does not need to inject errors in the program. For an input vector it gives us the statements that had no effect on the output, thus forcing the designer to get more/better test vectors.

## III. Proposed coverage metric

The metric we propose is based on observability coverage. This metric addresses not only which statements are executed by an input vector (controllability) but also if the statements have any influence on the output result (observability). This metric was motivated by recent work in observability analysis for hardware models [6].

### A. Hardware observability

The computation of the Observability-based Code COverage Metric (OCCOM) [6] can be done while simulating an HDL design. This computation is done with the help of tags. A tag signifies a possible change in the value of a variable due to an error. If a tag injected at a statement gets propagated to an output node then that statement is relevant to the output. This method consists of two steps:

1. The given HDL model is modified by adding new variables and moving statements out of conditional clauses. The modified HDL model is then simulated using a commercial simulator. This allows the extraction of information than by just using the original model.
2. A flowgraph is created from the modified HDL model and the results of the simulation are used to determine the coverage under OCCOM. In this step tag injection is used. Tags are inserted in the edges of the graph and propagation of the tags is done by selectively traversing paths from the edge to the output nodes.

The computed coverage information serves as a diagnostic aid to the designer. It helps to debug and design, and/or create better functional tests.

### B. Software observability

In our method to compute the software observability we do not inject errors or tags into the statements of the programs. The software program is modified to give us more information on the program. Then the modified program is executed and an observability coverage is computed.

In order to achieve the observability target, we need to keep track of all the statements that assign a variable. For that purpose, for each variable in the program we have a list of statements the variable depends on. When we arrive at a statement we save the variable that will be assigned. For that variable we build a list of dependencies which is the set union of the dependency lists of the variables that are at the right hand side (RHS) of the assignment.

When we reach an observable statement (in our case either a write to screen or to file) where the content of some variable is passed to the exterior of the program we check the statements in its list of dependencies. The statements that are not on that list are not observable from that output.

After every input vector is tested we end up with a set of observable statements. We can then give a measure of the coverage thoroughness of the input vectors on the program. Along with this measure we also know which statements were not relevant to the output. This can give the designer of the program sufficient information on how to design more/better test vectors.

The construction of the dependency lists is done dynamically while the program is running. Thus, the original code has to be modified. For each statement, the information regarding the dependencies that exist in it are saved.

The coverage analysis procedure starts with a program in the C language and a set of input stimuli. When we run the program with input stimuli and monitor its behavior using our analysis tool, our tool detects, for each stimulus, the statements in the program that did not have any control in the output result for that stimulus.

To illustrate and motivate the proposed metric, consider the simple program given in Figure 2. If we apply the test vectors ($a = 1$, $b = 1$) and ($a = -1$, $b = -1$) we get 100% statement coverage and 100% branch coverage. With those input stimuli every statement is executed and every branch is taken. In Figure 3 we verify the observability for the same input vectors. Starting with the vector ($a = 1$, $b = 1$) we get that $a$ depends on statement (1), then $b$ depends on statement (2). Since $x = b$, $x$ depends on the statements which $b$ depended and also on its own statement (3). In line (4) we have that $a$ depends not only on $b$ but also on $a$ since executing this branch side of the condition depends on $a$. So $a$ depends on statements (1), (2) and (4). The *else* branch of the condition is not executed. Nevertheless, since the value of $x$ depends on which branch is taken, $x$ will depend not only on statements (2) and (3) but also on variable $a$ and consequently on statement (1). Applying the same procedure to the rest of the program we get that the output will depend on statements (1), (2), (3) and (6). Using the other vector ($a = -1$, $b = -1$) we get that the statements (1), (2), (7) are the ones on which the output depends. When we make the set union of this two results we see that statements (4) and (5) have no control on the output. Thus, despite the fact that these vectors get 100% statement and branch coverage, in terms of observability we only get 75% coverage. The solution is to use more test vectors or use different ones. In Figure 4 we have the same analysis of the program but with test vectors ($a = 1$, $b = -1$) and ($a = -1$, $b = 1$). We can see that we get not only 100% statement and branch coverage, but also 100% of the statements are relevant to the output.

```
(1)   INPUT a;
(2)   INPUT b;
(3)   x = b;
      if (a > 0)
(4)     a = b;
      else
(5)     x = x+a;
      if (b > 0)
(6)     z = x;
      else
(7)     z = a;
(8)   OUTPUT z;
```

Fig. 2. Example of a simple program.

$$a = 1, b = 1$$

$a \rightarrow$ (1)
$b \rightarrow$ (2)
$x \rightarrow$ (2), (3)
$a \rightarrow$ (1), (2), (4)
$x \rightarrow$ (1), (2), (3)
$z \rightarrow$ (1), (2), (3), (6)
$OUT \rightarrow$ (1), (2), (3), (6)

$$a = -1, b = -1$$

$a \rightarrow$ (1)
$b \rightarrow$ (2)
$x \rightarrow$ (2), (3)
$x \rightarrow$ (1), (2), (3), (5)
$z \rightarrow$ (1), (2), (7)
$OUT \rightarrow$ (1), (2), (7)

Fig. 3. Observability coverage using ($a = 1$, $b = 1$) and ($a = -1$, $b = -1$) as input vectors.

When parsing the source program we are only interested in assignments, conditions and statements that generate some output. Included in the assignments we have the relation that exists between arguments in the call to a function and the parameters of the function. We can say that the former are assigned to the latter. The conditions in every control structure are taken into account. The statements that generate some output are the ones that write some information to file or to screen, or in an embedded system, to an observable register.

This method we propose will help the designer in catching bugs and designing more/better test vectors. For catching bugs, this coverage is more reliable than the ones based on controllability only. By designing test vectors based on observability, the designer can guarantee that the statements tested are relevant to the output, thus, enabling the bugs to reach an output where they can be detected. In the case of test vectors based on controllability, we do not have the guarantee that an error in a statement will show up in an output.

## IV. Implementation

The coverage metric being proposed was implemented to handle programs in the C language. The algorithm was implemented in a two step process. In the first step we transform the source program by adding for each statement a call to a function. This function will process the information extracted from the statement. Then, in the second step we compile the transformed program inside a framework that will allow

$a = 1, b = -1$

$a \rightarrow$ (1)
$b \rightarrow$ (2)
$x \rightarrow$ (2), (3)
$a \rightarrow$ (1), (2), (4)
$x \rightarrow$ (1), (2), (3)
$z \rightarrow$ (1), (2), (4), (7)
$OUT \rightarrow$ (1), (2), (4), (7)

$a = -1, b = 1$

$a \rightarrow$ (1)
$b \rightarrow$ (2)
$x \rightarrow$ (2), (3)
$x \rightarrow$ (1), (2), (3), (5)
$z \rightarrow$ (1), (2), (3), (5), (6)
$OUT \rightarrow$ (1), (2), (3), (5), (6)

Fig. 4. Observability coverage using ($a = 1$, $b = -1$) and ($a = -1$, $b = 1$) as input vectors.

several input vectors to be run and obtain an overall estimate of the observability coverage for these vectors.

### A. Parser

The parser used was c2c which is a public-domain software program. c2c works by making 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 changing the AST, the c2c tool produces the C program for that new AST. In our case, the modifications made are, for each statement, adding one of several functions to the code. In the case of an assignment, a *control* function is added after the assignment. When we have a call to a function that will send something to the output, an *observe* function is added.

The information regarding the dependencies between variables is saved in a list of dependencies. This list of dependencies contains the list of statements on which the variable depends. When building the list, we are interested in the address of the variables and not in its value. What we want to know is if some position in memory was modified and which statements that modification depends on. This allows us to work with dynamically allocated objects from simple integers to complex structures.

#### A.1 Assignments

Take, for example, the simple statement $x = a + b$. In this case the control function is called specifying that the variable $x$ will depend on variables $a$ and $b$,

x=a+b; ⇒ x=a+b; *control*(&x,&a,&b);

This means that if $x$ arrives at an output statement (e.g., *printf*), the statement $x = a + b$ and the statements where $a$ and $b$ were assigned are recursively covered.

If a variable that was already assigned is reassigned, a new list of dependencies is built as if it was the variable's first assignment. This new list will then replace the older list. This allow us to handle cases such as $x = x + a$.

In more complex statements where we have more than one operation in the right hand side (RHS) of the statement, the statement is divided into its parts so that we get simple statements. For each of these simple statements the *control* function is called. For example,

x=a+b+c; ⇒ temp=a+b; *control*(&temp,&a,&b);
x=temp+c; *control*(&x,&temp,&c);

The *control* function takes into account what kind of operation is done on the RHS. In the case of the multiplication the operands are checked to see if one of them is zero. In that case the assigned variable will only depend on the operand that has value zero, since the other operand does not have any effect on the result. This procedure is similar to all operations that have this property, where a particular value in an operand defines the result of the operation, without need to take into account the value of the other operand.

#### A.2 Conditions

All the conditions are reduced to a single variable. So,

if (x>a)... ⇒ temp=x>a; *control*(&temp,&x,&a);
if (temp)..

This way we treat the conditions as just any simple statement. Since every statement in the block inside the *if* condition will depend on the newly created temporary variable in the condition, all assign variables in the block will depend on the statements where that new variable was assigned. The fact that only one variable remains in the condition will simplify the building of the dependencies. In all other control structures, such as *while*, *do*, *for* and *switch*, the same procedure is applied.

#### A.3 Appending dependencies

Every statement in a conditional block depends on the condition. This applies even to those statements that are in a branch that is not taken when the program is executed. Thus, at the end of the conditional block, to every variable that was inside the block we append to its list of dependencies the variables that were on the condition statement.

#### A.4 Function Calls

We have to make the correspondence between function arguments in the function call and the parameters in the actual function. To enable this, for each argument we add another argument that is the address of this argument. That way we do not lose the address of the variable when inside the function. In the case of arguments passed as reference or constants the same procedure is applied but since we cannot pass the address of a constant we pass instead a flag indicating that we are not passing the address of a variable. Besides the addition of one new argument for each original argument, if the function returns some value, we also add another variable to the list of arguments. This variable points to the list of dependencies of the returned variable when the function ends.

Since we are working with variable addresses, we need to delete, from the list of dependencies, the variables that were created in the function. This is because those variables are automatically freed when the function ends and the addresses can be re-utilized in the future by other variables. Therefore we need to clean all references to these addresses. Conflicts could arise when the same address is used in more than one function when computing the lists of dependencies.

#### A.5 Memory allocation

When the program allocates memory dynamically, some caution is necessary. The same procedure described in the previous paragraph is needed. This way, when the program frees a previously allocated block of memory we delete from our structure all the variables that are in that block.

#### A.6 Structures, arrays and pointers

Structures, arrays and pointers are not very different from simple variables since we always manipulate the address of the variable. The difference from simple variables is that the field that is being accessed depends also on the beginning of the structure. So, we have,

x->a=b; ⇒ x->a+b; *control*(&(x->a),&x,&b);

A.7 Observable statements

When the program outputs the value of some variable, an observe function is called.

*printf*(``%i'',x); $\Rightarrow$ *printf*(``%i'',x); *observe*(&x);

This function checks the list of dependencies for the output variable and marks the statements that are in the list of dependencies as covered according to the observability metric. This means that the statements that are in the list of dependencies are the statements upon which the value of the output variable depended.

A.8 Current Limitations

The major limitation in this implementation is in the handling of the standard C library functions. This presents a problem since we do not know which arguments are altered. For those functions that are not supported, we assume that the arguments of the function call will not change. This is not always true. However, this will change as we add support for those functions.

Another limitation is in the *goto* statement. We allow jumps to any part of the software program, if the jump is semantically correct. However we can only guarantee meaningful results if the jump is made inside the same block or if the jump is made to the end of the block from where it originated.

### B. Calculating the coverage

The second step in the process is to compile the modified program and link it with the functions that will execute the program, process the information, and display that information.

The main function of the modified program is called from within our framework. Special care must be taken to pass the proper arguments to that function. After the call to the main function, the program we wish to test runs as if it was running by itself. But now, for each statement executed a function is called.

These functions process the information on the statement and depending on the type of statement they can add elements to a list of dependencies or mark the statement as observable.

After all the input vectors have been run, statistics on the percentage of observed statements are given, together with information about the non-observed statements.

## V. Results

In this section we show four examples we used to test the observability based metric being proposed. One of the program computes Fibonacci numbers, one matches a stream of characters against a string, one computes the Huffman code and the last one implements the Fast Fourier Transform (FFT). All four were implemented using the C language. As explained in Section IV, the first step is parsing the source code so that the functions for controllability and observability are added to the program. Then the modified code runs in a support program which gives us the measure of the coverage. We present results for statement coverage and observability based coverage and we compare the two for the different programs.

### A. Fibonacci numbers

The program to calculate Fibonacci numbers is a very simple one implemented without using recursion. The program takes as input a positive integer $n$, and gives as result the Fibonacci number of $n$, $F[n]$. Table I shows the measure of coverage for several input values. The first set of input values is the minimum necessary for achieving 100% statement coverage. However, it is not sufficient in terms of observability based coverage. 100% observability coverage is obtained by adding another value to the set of input values. This example shows that observability based coverage is a stronger metric than statement coverage. Although with the first sequence of three input vectors all statements are exercised, not all are observed at the output. With one more input vector we can guarantee that all statements are observable.

| Input Values | Statement Coverage | Observability Coverage |
|---|---|---|
| 0, 1, 3 | 100% | 87% |
| 0, 1, 3, 4 | 100% | 100% |

TABLE I. Coverage for the Fibonacci procedure.

| Input Values | Statement Coverage | Observability Coverage |
|---|---|---|
| input1 | 88% | 0% |
| input2 | 92% | 0% |
| input3 | 96% | 68% |
| input4 | 96% | 84% |
| input5 | 100% | 100% |

TABLE II. Coverage for the string match procedure.

### B. String match

The string match program reads a stream of characters and detects the occurrence of a specific string. The program activates the output only when there is a match. Table II shows the results for different input vectors. Each input corresponds to a stream of characters being feed to the program. The first two inputs are streams of characters that do not have the matching string in it. So, despite the fact that they can give a high percentage of statements covered, they do not produce any output. Thus no statement is observable from the output. The last three inputs are inputs where the stream of characters has the matching string in it. In these two cases we can see that some of the statements executed influence the output. The inputs are ordered such that *input5* is a stream of characters longer than *input1*. As can also be seen from the results, to reach a certain percentage of coverage, if we want to use the observability coverage we need longer input vectors. Thus making the test more complete.

### C. Huffman code

The Huffman algorithm takes as input a set of characters and gives for each character the binary character code. That code, which depends on the frequency each character appears in the input, can then be used to compress that set of characters. The implementation used was based on [4]. This example uses dynamically allocated structures linked in binary trees. Furthermore, it uses recursive procedure calls extensively. In Table III we present the results for statement and observability coverage for two input vectors. Vector *input1* has only two characters. Despite the fact that it was a very small set, it still gave us a 99.4% statement coverage. The observability coverage was a little smaller but still close to the statement coverage. This means that almost all executed statements had an effect on the output. *input2* is a longer vector and in this case both statement and observability coverage reached 100% percent.

### D. Fast Fourier Transform

The Fast Fourier Transform algorithm was implemented as it appears in [14]. This algorithm computes the Fourier Transform of a vector whose size is a power of 2. The vectors used to measure the coverage

| Input Values | Statement Coverage | Observability Coverage |
|---|---|---|
| input1 | 99,4% | 98,8% |
| input2 | 100% | 100% |

TABLE III. Coverage for the Huffman code procedure.

| Input Values | Statement Coverage | Observability Coverage |
|---|---|---|
| dirac | 100% | 81% |
| constant | 100% | 82% |
| sine | 100% | 100% |

TABLE IV. Coverage for the Fast Fourier Transform procedure.

were the *dirac* function which consists of a single pulse, a *constant* function and the *sine* function. The results are presented in Table IV. As it can be seen, for the first two vectors, *dirac* and *constant* functions, despite the fact that both execute every single statement in the program, almost 20% of the statements do not have any influence in the output. Only when we use as input the *sine* function can we get 100% statement coverage and 100% observability based coverage.

This happens mainly because, as stated in Section IV, when propagating the list of dependencies we take into account if we are multiplying some value by zero. The FFT algorithm has numerous multiplications. This way, the first two vectors used, which have a great number of zeros and are very regular, do not give a good observability coverage. So, they should not be used alone in testing this program, despite the fact that they achieve 100% statement coverage.

### E. Overhead

Since we are calling a function for each statement executed our method has significant CPU time and memory overhead. That can be seen in Table V where we present the CPU time overhead. We only present results for the Huffman and the FFT procedures since they are the more computation intensive procedures. These results were obtained in a Sparc Ultra I running at 170MHz with 384M of main memory. In Table V we show the time in seconds to execute the procedure with and without computing the coverage. As it can be seen the time overhead can be very high.

## VI. Conclusions

We presented a new software coverage metric based on observability rather than simply on controllability. It gives us not only which statements were executed, but also if they have any effect on the output. We have shown that statement coverage alone may not give an accurate measure of the test thoroughness. The results show that this metric is stronger than statement coverage. So, this metric has great potential to be used in embedded software testing. There is significant overhead due to the fact that for each statement, a function call is made. Nevertheless, in embedded systems where software is not very large this should not be a major limitation. Furthermore, it can be used in a co-simulation environment with hardware coverage metrics such as OCCOM [6] which uses also an observability coverage metric.

| Procedure | Without Coverage | With Coverage |
|---|---|---|
| Huffman (file size = 300k) | 0.23s | 203s |
| FFT (#vectors = 10000) | 0.55s | 147s |

TABLE V. Time overhead for the Huffman and FFT procedures.

In the future we will be extending this work for not only giving the coverage metric but also to generate automatically the test vectors to increase the observability coverage. We are also investigating methods to reduce the overhead in coverage computation for embedded software.

## References


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

[2] A. Benveniste and P. Le Guernic. Hybrid Dynamical Systems Theory and the SIGNAL Language. *IEEE Transactions on Automatic Control*, 35(5):525–546, May 1990.

[3] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill e MIT Press, 1990.

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

[6] 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.

[7] 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.

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

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

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

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

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

[13] F. Maraninchi. The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[14] William H. Press, Saul A. Taukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition edition, 1992.

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

[16] 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.

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