Journal of Computer Science 5 (4):302-310, 2009
ISSN 1549-3636

A Control-Oriented Coverage Metric and its Evaluation for Hardware Designs

[1]Shireesh Verma, [2]Kiran Ramineni and [3]Ian G. Harris
[1]Conexant Systems Inc., Newport Beach, CA 92660
[2]Marvell Semiconductor Inc., Austin, TX 78746
[3]University of California, Irvine CA 92697, USA

Abstract: Problem statement: Dynamic verification, the use of simulation to determine design correctness, is widely used due to its tractability for large hardware designs. A serious limitation of dynamic techniques is the difficulty in determining whether or not a test sequence is sufficient to detect all likely design errors. Coverage metrics are used to address this problem by providing a set of goals to be achieved during the simulation process; if all coverage goals are satisfied then the test sequence is assumed to be complete. Coverage metrics hence evaluate the ability of a test sequence to detect design errors and are essential to the verification process. A key source of difficulty in determining error detection is that the control-flow path traversed in the presence of an error cannot be determined. This problem becomes particularly difficult in case of typical industrial designs involving interaction of control flow paths between concurrent processes. Error detection can only be accurately determined by exploring the set of all control-flow paths, which may be traversed as a result of an error. Also, there is no technique to identify a correlation between coverage metrics and hardware design quality. **Approach:** We present a coverage metric that determined the propagation of error effects along all possible erroneous control-flow paths across processes. The complexity of exploring multiple control-flow paths was greatly alleviated by heuristically pruning infeasible control-flow paths using the algorithm that we present. We also presented a technique to evaluate coverage metric by examining its ability to ensure the detection of real design errors. We injected errors in the design to correlate their detection with the coverage computed by our metric. **Results:** Our coverage metric although analyzed all control-flow paths it pruned the infeasible ones and eliminated them from coverage consideration, hence reducing the complexity of generating tests meant to execute them. The metric also correlated better with detection of design errors than some well-studied metrics do. **Conclusion:** The proposed coverage metric provided high accuracy in measurement of coverage in designs that contain complex control-flow with concurrent processes. It is superior at detecting design error when compared with the metrics it was compared with.

Key words: Verification, simulation, coverage metrics, test sequence, controllability, oberservability

INTRODUCTION

Verification is known to be an expensive and time-consuming part of the design process. Verification approaches can be broadly grouped into two categories, formal verification and dynamic verification. Dynamic verification, the subject of this study, involves the use of simulation to verify design correctness. A design is simulated with a test sequence and the design is assumed to be correct if the test responses match the correct responses. A general framework of the dynamic verification process is shown in Fig. 1 where the process starts on the left side with an executable design description in a Hardware Description Language (HDL). The three main steps of the process are test generation, simulation and response evaluation. Simulation is performed using a software tool but test generation and response evaluation, are often heavily dependent on manual interaction making them costly. The most significant weakness of dynamic verification techniques is the difficulty of determining the error detection ability of a test sequence. We refer to a test sequence as being complete if it detects all potential design errors. The effectiveness of dynamic verification depends on the ability to determine whether or not a test sequence is complete. An incomplete test sequence will lead to the possibility of design errors in the final product and reduced system quality.

Corresponding Author: Shireesh Verma, Conexant Systems Inc., Newport Beach, CA 92660

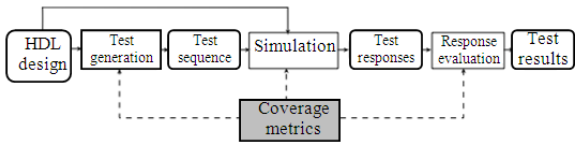

Fig. 1: Dynamic verification flow

The problem of determining the completeness of a test sequence is addressed by using coverage metrics, which abstractly define the coverage goals to be satisfied during simulation. A coverage metric defines a set of criteria that are used to determine which errors are detected by a test sequence. A coverage metric provides an empirical measure of the completeness of a test sequence and the error detection criteria can be used to direct the test generation process. For example, statement coverage metric would define the verification goal as the execution of all HDL statements during simulation. A test sequence is evaluated by determining the fraction of coverage goals, which are satisfied using the test sequence.

Some metrics targeted towards errors in Finite State Machines (FSMs)[3,11] have been developed; state coverage model, which requires that all states be reached and transition coverage, which requires that all transitions be traversed. Metrics based on the traversal of paths through the Control Dataflow Graph (CDFG) have been used; branch coverage[4], which requires that the set of CDFG paths executed include all conditional branches and path coverage[1], which requires that all control-flow paths be executed. Researchers have studied techniques to reduce the complexity of path coverage metric[9].

Many control-dataflow coverage metrics consider the requirements for error activation without explicitly considering error effect observability. Researchers have developed observability-based behavioral error models to alleviate this weakness. The OCCOM approach[5] inserts errors, called tags, at each variable assignment to represent a positive or negative offset from the correct signal value. The propagation of these tags to observable variables is determined using a set of propagation rules for behavioral operations. In[10], the software dataflow testing technique is enhanced to identify chains of variable definitions and uses which extend to observable variables. Both of these observability-oriented metrics have difficulty considering the multitude of possible control flow paths, which may be executed in the presence of an error. Either complex control flow has not been considered or only a small subset of possible control flow paths is considered.

A significant difficulty in predicting error propagation is the ambiguity in the control flow path executed in the presence of an error. For example, consider the evaluation of the conditional statement “if (x>5) then...” in the presence of an unknown error effect on variable x. If the correct value of x is 0 then the correct value of the conditional predicate is FALSE, but the incorrect predicate value is unknown. The control flow choices at branch points have a major impact on the sequence of instructions executed and the propagation of error effects. In the simple conditional predicate example above, if the predicate evaluates to FALSE in the erroneous design, the error effect may not propagate and the error may remain undetected as a result. The problem of determining error propagation in the presence of ambiguity in control flow quickly becomes severe as control flow becomes more complex particularly with interacting control flow paths across processes.

Our approach explores all control flow paths across processes, which could be executed as a result of an error. Error detection is estimated along each control flow path and the detection probabilities along each path are combined to compute total error detection probability. To alleviate the computational complexity of exploring multiple control flow paths, we use knowledge of the sign of an error effect to prune infeasible paths, which could never be executed as a result of an error. Our approach provides an accurate estimate of error detection probabilities by considering the full range of erroneous behaviors. We call this metric a Control-Oriented Coverage Metric (COCM)[12]. In case of designs consisting of interacting processes a set consisting of one path from each process is considered.

Very little work has been done on the evaluation of existing metrics. In the hardware domain researchers have performed a limited evaluation of the statement and branch coverage metrics with two small examples[6]. We present a technique to estimate the correlation between coverage metrics and design error detection. Test sequences are generated for a set of benchmark examples and coverage is computed for each test sequence using the coverage metric under evaluation. We also compute the error coverage, the fraction of design errors detected, for each test sequence by injecting a large number of errors into each benchmark. The coverage metric is evaluated by examining the difference between the metric coverage and the error coverage values for the same benchmarks. We employed this technique on the statement, branch and state coverage metrics to serve as a baseline for the evaluation of our control-oriented coverage metric.

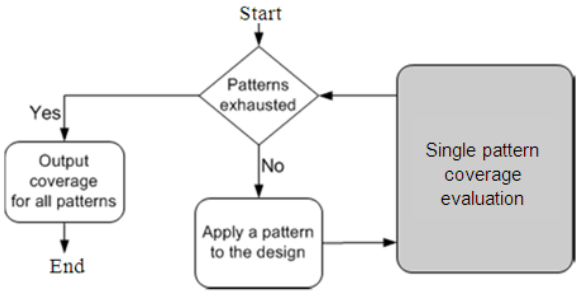

Fig. 2: System structure

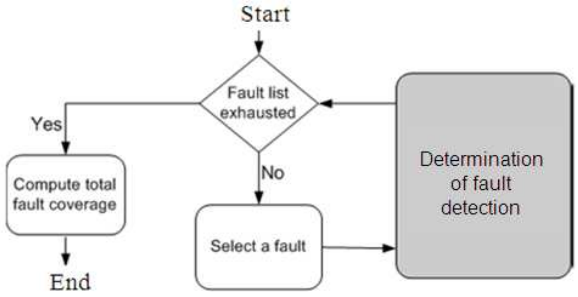

Fig. 3: Single Pattern Coverage Evaluation

Overview: The inputs to our method are a behavioral HDL description of the design and a set of test patterns and the output is the set of errors that propagate to an observable point for every input pattern. An observable point in a simulation is a variable that may get an incorrect value because of a design error.

Figure 2 shows the way COCM coverage value is computed for test patterns, which constitute a test sequence. We automatically generate a list of potential errors in the design. We then simulate the HDL description with each test pattern one at a time until all the generated patterns are exhausted. The coverage value for each pattern is computed at the end of its simulation.

Figure 3 shows a sequence of steps we follow to compute the COCM coverage for each test pattern. We select an arbitrary error from the generated list and then simulate the HDL description to determine the propagation of the error. We evaluate its detection probability at the end of the simulation. This sequence of steps is repeated until all the errors in the list are exhausted. The COCM coverage value for the given test pattern is computed on the basis of the detection probabilities of the individual errors.

Error list generation: We use the tag model[5] for injecting and propagating design errors. We consider every assignment statement and input variable in the HDL description as potential error injection points.

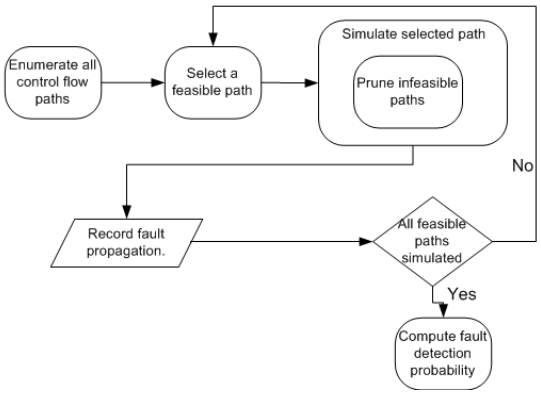

Fig. 4: Determination of error detection

The tags are injected on to the Left Hand Side (LHS) variable of each assignment statement and on each of the input variables. We parse the HDL description to obtain a list of all the assignment statements and input variables. There are following three types of tags defined:

- Positive tags: These cause the LHS variable to have a greater value than the correct one
- Negative tags: These cause the LHS variable to have a lesser value than the correct one
- Unknown tags: These cause the LHS variable to have a greater or lesser value than the correct one

Since there are three types of tags, our error list may consist of a number of errors that is three times the number of assignment statements and input variables in the HDL description.

Evaluation of error tag propagation: The simulation and evaluation of a single error is depicted in the Fig. 4. We compute all the possible control flow paths and maintain a list of these pre-determined paths. We select an arbitrary path from the list and then the simulation is guided through that path. During each simulation, an error is injected by inducing a tag using the simulator's directives. During the simulation, we monitor the observable points where an injected tag could manifest itself. This evaluation is repeated at every time step of the simulation in order to propagate an error. The error propagation is updated after each statement simulation. Each injected error propagates in one of the following two ways through the behavior.

Data flow propagation an error can propagate from one variable to another via a direct data dependency between the two variables created by a variable assignment. For example, in the code shown in Fig. 5 a

tag on variable a would propagate to variable x as a result of the assignment on line number 6. We use a calculus to perform tag propagation through each type of behavioral operation[5]. The propagation table used for an addition operation is shown in Table 1. The top most row in the Table 1 represents the tags on b, while the left most column represents the tags on a. "+", "-" and "?" signify a positive, a negative and an unknown tag respectively and a "0" signifies absence of a tag. Variables a or b followed by "0", "+", "-" or "?" signify the presence of tags on these variables.

Control flow propagation it is possible for an error to change the executed control flow path by changing the result of a conditional. Changing the control flow path can have drastic effects on error propagation by introducing indirect variable dependencies and by changing the sequence of dataflow operations. An indirect data dependency can be seen in Fig. 6 between variables cond and x. Although there is no dataflow dependency between the two variables, it is clear that the value of cond determines the value of x. If the correct value of the cond variable is 1 and it has a positive tag, then either branch 2 or 3 could be taken. The sign of tag on x depends on the magnitude of the error on cond. cond = 2 creates a negative tag on x and cond = 3 creates a positive tag on x.

```
1   Begin
2   Reg x;
3   Integer a;
4   x = 5;
5   a = 2;
6   x = x + a;
7   end
```

Fig. 5: A simple data flow example

```
1   Case (cond)
2   1: x = 5;
3   2: x = 1;
4   3: x = 10;
5   End case
```

Fig. 6: A simple control flow example

Table 1: Tag calculus for addition

a/b	b	b-	b+	b?
a	0	-	+	?
a-	-	-	?	?
a+	+	?	+	?
a?	?	?	?	?

When a tag changes the result of a conditional predicate, it alters the sequence of dataflow operations executed. This consequently changes all subsequent dataflow propagation. Error propagation depends on the sequence of dataflow operations, so any change to that sequence will also alter the dataflow error propagation.

At the end of simulation of each feasible control flow path the tag propagation data is recorded and the tags on the variables are stored. The propagation data is the set of variables that contain tags at the end of simulation. The stored tag values at the end of each simulated feasible control flow path are used to determine the tags on the variables at the start of simulation of next feasible control flow path. This process is repeated until there are no feasible paths left to simulate in the control flow path list. Once all the paths in the list are simulated, the error detection probability for the injected error is computed using the error propagation data recorded for individual paths.

Pruning infeasible paths: The number of simulations required to determine the impact of tags can become intractable as the number of control flow paths may be exponential in the number of conditionals. If a tag is present on a variable involved in a conditional predicate, then we determine whether or not the outcome of the predicate depends on the magnitude of the tag. If it does, the path is pruned thereby reducing the time complexity of this approach. The following steps are involved in pruning.

Generating decision tree: We generate a decision tree from the HDL description at the parsing step. A path through the decision tree from root to leaf corresponds to a control flow path in the HDL description. The tree consists of following two types of nodes.

Condition node each condition node corresponds to a control flow decision that is made during simulation. A single conditional predicate in the HDL may map to many decision nodes since each predicate may be evaluated on many control flow paths. Each condition node has a number of children corresponding to the number of outcomes of the conditional predicate. A condition node corresponding to an IF Else statement will have two children, TRUE and FALSE. A condition node corresponding to a case statement will have as many children as the case has branches. These nodes are represented as white nodes.

A sequence of nodes starting with the top most condition node and ending at a leaf node of the tree constitutes a control flow path. This signifies, there is a one-to-one mapping between the leaf nodes and the

control flow paths. In this case, we have six control flow paths each corresponding to a leaf node. These control flow paths are used to guide the simulation.

Pruning algorithm: It is possible to reduce the number of control flow possibilities to be considered by using dynamic tag information computed during simulation. A child c of a conditional node n node can be pruned if the predicate P (n) associated with node n can never evaluate to the value required to lead to child c in the presence of the error. For example, the root node in Fig. 7 has two children, one associated with a TRUE predicate value and one associated with a FALSE predicate value. If during simulation the variable b has the value 11 and b has a positive tag, then it is not possible for the predicate to evaluate to FALSE in the presence of the error. In this case, the child in the FALSE direction can be pruned from consideration without reducing the accuracy of the approach. When simulation encounters a conditional predicate, the following steps are performed:

- Determine the set of possible erroneous predicate values, V_f-this step is performed by limiting the range of the variables in accordance with the signs of their tags and determining if the predicate is satisfiable or not. This problem in the worst case is a version of the SATISFIABILITY problem, but in practice it is trivial given the monotonic nature of the vast majority of predicates in real examples
- Prune children corresponding to impossible predicate values-once the possible predicate values are known, all children of the decision node that can be reached only by impossible predicate values can be safely pruned.

The decision tree for an HDL description in Fig. 7 is shown in Fig. 8. We will use the example used in Fig. 7 in order to illustrate the pruning of the decision tree in Fig. 8. Let us consider the following cases of tag injection on variable a.

Negative tag at line number 2: This results in a negative tag for b at the assignment statement b = a + 5 at line number 3. The condition node b>10 evaluates to FALSE, as there is a negative tag on b and its value is 10. This allows us to prune TRUE branch, which would never be taken in presence of the error. On the FALSE branch, assignment statement out = 1 -a at line number 12 is evaluated. Out gets a value -4 and a positive tag since a, which has negative tag, is being subtracted. The value of the next condition out < a can not be uniquely determined because it depends on the relative magnitudes of tags out and a. For example, out < a would evaluate to TRUE when magnitudes of tags on out and a are 1 and 1 respectively in which case the resultant expression becomes ((-4 + 1)<(5 - 1)), which evaluates to TRUE. However, when the magnitudes of the tags are 5 and 6 respectively for out and a, the resultant expression (-4 + 5)<(5 - 6)) evaluates to FALSE. Pruning cannot occur at the decision node out < an on account of ambiguity resulting from the magnitudes of the tags. The pruned tree is depicted in Fig. 9. The dashed lines represent the pruned part of the tree. About 66.66% of the total number of control flow paths is pruned in this case.

```
1  begin
2  a = 5;
3  b = a+5;
4  if (b > 10)
5     begin
6     if (a < 5)
7           out = 1 + a;
8     else
9           out = 1;
10    end
11 else
12    out = 1 - a;
13 if (out < a)
14    out = 0;
15 else
16    out = 1;
17 end
```

Fig. 7: An HDL example for illustration

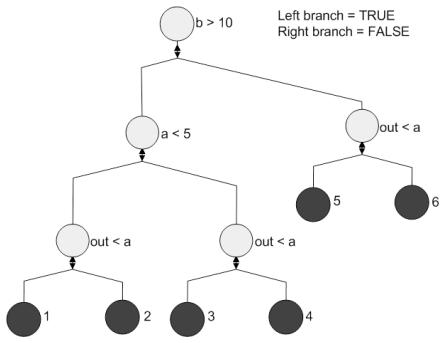

Fig. 8: The decision tree for HDL example

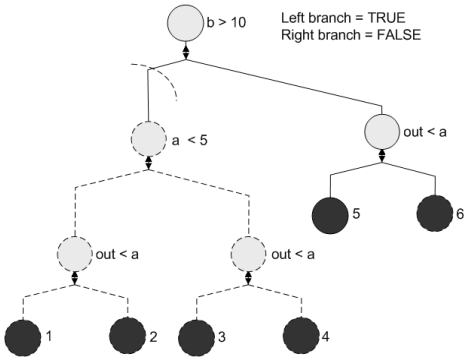

Fig. 9: Pruning case with negative tag on a

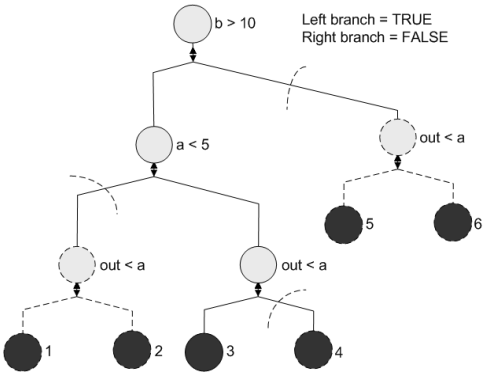

Fig. 10: Pruning case with positive tag on a

Positive tag at line number 2: This results in a positive tag for b at the next assignment statement b = a +5 at line number 3. The condition node b>10 evaluates to TRUE as there is a positive tag on b and its value is 10. This allows pruning FALSE branch that would never be taken. The next condition node a <5 evaluates to FALSE as a has positive tag and its value is 5. The TRUE branch is pruned at the decision node a <5. The assignment statement out = 1 at line number 9 is evaluated. The next decision node out < a evaluates to TRUE as there is positive tag on a and its value is greater than that of out. So, the FALSE branch is pruned. The final value of out is 0. The pruned tree is shown in Fig. 10. The dashed lines represent the pruned part of the tree. 83.33% of the total number of control flow paths is pruned in this case.

Computing coverage: The COCM coverage is based on the computation of Tag Detection Probability (TDP). An error tag is considered detected if any of observable points obtains a tag at the end of the simulation. TDP of a variable v is the probability that v gets a tag at the end of simulation. The number of simulations run for an error e tag is represented by S (e). D (V, e) represents number of simulations where the tag propagated to an observable point and V is the set of observable points where the error is detected. TDP (V, e) is the probability of a single error tag e being detected at any of the observable points in V. TC (t) is the tag coverage value for a test sequence t:

$$TDP(V, e) = D(V, e)/S(e) \quad (1)$$

$$TC(t) = 1 - \prod_{i=1}^{e} (1 - TDP(V, i)) \quad (2)$$

Metric evaluation: In our description of the evaluation approach we will refer to the fraction of coverage goals satisfied for a benchmark b when simulated with test sequence t as the metric coverage, $MC_{b,t}$.

```
Foreach b € Benchmark set
        Generate test sequence t for benchmark b;
        Compute error coverage, EC_b,t;
        Compute metric coverage, MC_b,t;
        Record | EC_b,t - MC_b,t |;
```

Fig. 11: Metric evaluation algorithm

We will refer to the fraction of potential design errors detected in a benchmark b when simulated with test sequence t as the error coverage, $EC_{b,t}$. The algorithm in Fig. 11 computes $EC_{b,t}$ and $MC_{b,t}$ whose difference for a set of benchmarks reveals how closely error detection is modeled by the coverage metric for the test sequences used. The lesser the difference, the better the metric under evaluation will be at detecting design errors, hence better its quality.

Test sequence generation: The difference between error coverage and metric coverage depends on the test generation technique. Hence sequences for evaluation purpose should be generated as it would be in practice. We identify some guidelines which any reasonable test sequence should reflect

- Succinctness-a test sequence should be just long enough to attain 100% coverage using the metric
- Randomness-inputs are randomized to maximize coverage except where it would make coverage of part of state space difficult
- "Special" control inputs-some input control signals have a drastic impact on the behavior and they should be assigned in a special way. For example, assigning some reset randomly would probabilistically keep the system in reset half the time, making it hard to explore the entire state space

Error coverage computation: Computation of the $EC_{b,t}$ requires that each potential design error be inserted into the design individually and that the erroneous designs be simulated with the test sequence. Inserting errors into a design description requires the use of an error model, which describes the set of design errors to be considered. The wide variety of potential design errors makes it impossible to capture all of these errors at this time. Instead, we restrict our investigation to a subset of design errors, which has been found to be most common in hardware design[2]. These errors include simple typographical mistakes and accounted for 12.7% of the design errors found in the Pentium 4[2]. In order to inject errors we use the mutation analysis technique studied previously in software testing and

hardware validation[7,8]. Following mutation operators describe expected design errors:

- Arithmetic Operator Replacement (AOR)-each occurrence of one of the arithmetic operators (+, -, * and /) is replaced by each of the other operators
- Relational Operator Replacement (ROR)-each occurrence of one of the relational operators (<, >, ≤, ≤, = and ≠) is replaced by each of the other operators
- Logical Operator Replacement (LOR)-each occurrence of one of the logical operators (&, &&, | and ||) is replaced by each of the other operators

MATERIALS AND METHODS

We used a set of nine ITC'99 benchmarks in order to demonstrate the formulation of the proposed metric and to evaluate its quality. We used Cadence Verilog-XL simulator for our experiments. We developed a C application that interacts with the simulator while running a simulation. First, we parsed a Verilog design descriptions and constructed its decision tree. Then we assigned simulation callbacks for each of the assignment statements so that we could interact with the internal data structures of the Verilog simulator. A callback associated with a statement forces the simulator to acquiesce control to Verilog Procedural Interface (VPI) when the statement is reached while simulating. At that point, we applied our tag calculus for each of the assignments and store subsequent changes in the tag values for each of the variables affected.

All the test sequences were generated in a random fashion. However, two different random test generation setups were used. First setup aimed at achieving high COCM coverage for each design in order to demonstrate the efficiency of pruning obtained by our algorithm. We generated test sequences consisting of 20 random patterns for each design. The second setup generated test sequences with enough random patterns to achieve 100% coverage by a given metric, which in this case was chosen to be state coverage. State coverage was chosen, as the benchmark examples were mostly control-flow driven. The number of test patterns in the test sequence is different for each benchmark as it varied based on the difficulty of "hitting" the states in each design. All the experiments were run in Unix environment on a Sun SPARC 5 machine with 1.6 GHz speed and 2 GB RAM.

RESULTS

The results are divided into two parts. In the first part, we demonstrate the performance of the proposed control-oriented coverage metric with the benchmark designs and the efficiency achieved by our pruning algorithm. In the second part, we compare its quality with that of state, statement and branch coverage metrics. The same set of test sequences is used for evaluating all the three coverage metrics. For the first part, each row in Table 2 shows the results for benchmark design whose name is listed in the first column. The second column lists the total number of tags injected for each design. The third column shows percentage coverage values computed. The next two columns list average number of simulations run and pruned, respectively for each tag. The last column shows the percentage of pruned simulations.

We do not present direct performance results, however the performance overhead of applying our methodology can be approximated as the average number of simulations per tag as shown in Table 2. There is an additional overhead associated with pruning but it is small compared to the cost of performing multiple simulations required by this methodology.

For the second part, each row in the Table 3 shows the name of the benchmarks listed in the first column and the corresponding results for that benchmark. The second column lists the total error coverage achieved by performing mutation analysis for every design. The third column shows the percentage coverage computed by COCM metric. Percentage state, statement and branch coverage numbers are shown in the fourth, 5th and 6th columns respectively. The total numbers of patterns run on each benchmark and the numbers of mutations used for error coverage are shown in seventh and eighth columns respectively. Execution times (CPU) for COCM, the numbers of tags induced for obtaining coverage by COCM are depicted in the subsequent columns.

Table 2: COCM results

Bench mark	Tags	Cvg. (%)	Avg. no per tag		Pruned (%)
			Sims	Pruned	
b01	195	100.0	11.00	6.20	36.6
b02	66	100.0	9.00	8.00	47.1
b03	156	100.0	3.40	13.60	79.7
b06	168	100.0	7.66	6.33	37.3
b07	90	100.0	1.66	11.30	86.9
b08	60	100.0	5.00	4.00	44.8
b09	111	67.3	3.10	5.90	65.4
b10	279	50.1	10.00	15.00	60.0
b12	711	87.1	21.00	27.00	74.2

Table 3: Metric evaluation results

Bench mark	Error Cvg. (%)	Control oriented Cvg. (%)	State Cvg. (%)	Statement Cvg. (%)	Branch Cvg. (%)	No. of patterns	No. of mutants	COCM CPU time (sec)	No. of tags
b01	25	35	100	100	100	9	57	127	51
b02	44	98	100	100	100	9	8	176	72
b03	23	45	100	100	100	5	43	270	122
b06	57	61	100	100	100	10	28	304	144
b07	0	47	100	100	100	9	32	215	99
b08	0	15	100	100	100	5	26	154	66
b09	32	40	100	84	78	8	23	192	78
b10	26	38	100	89	63	12	123	327	189
b12	39	47	100	91	67	19	327	729	523

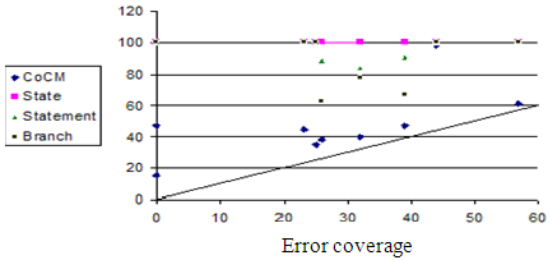

Fig. 12: Comparison of coverage metrics

For example, error coverage for b06 is 57% while COCM coverage is 61% when simulated for 10 patterns. The error coverage for b06 is obtained by inducing 28 mutants. The COCM coverage is computed by introducing 144 tags, simulating one tag at a time in a total of 304 sec. No error coverage is observed in b07 and b08 because of low observability. These designs have only one output variable, which gets updated only after applying a test sequence with a certain number of patterns (b07 needs 20 patterns). We use only enough patterns to obtain 100% state coverage. The COCM coverage is observed as the induced tags force a wrong path.

Figure 12 shows a graphical description of the evaluation results. It shows four sets of points, (1) COCM Coverage, (2) FSM (state) Coverage, (3) Statement Coverage and (4) Branch coverage. Each point in a set corresponds to a benchmark design. Error coverage is depicted on X-axis while percentage coverage numbers from various metrics are plotted on Y-axis. The broken line joining coordinates (0, 0) and (60, 60) represents the ideal case where all points would lie on it for a coverage metric which perfectly tracks the error coverage. So the closer a point is to the line better is the correlation between corresponding coverage and the detection of errors under consideration. It is clear at a glance that the set of points corresponding to COCM is closer to the perfect line than the other metrics. The difference in the quality can be observed quantitatively by computing the average coverage difference, the difference between COCM/state/statement/branch coverage and error coverage. A small difference indicates a more accurate metric. The average coverage difference for our COCM coverage is 21.5% as compared to 74.13% for state coverage, 70.75% for statement coverage and 66.75% for branch coverage. We also compute the standard deviation for coverage difference for all four coverage metrics. It is 18.75% for COCM coverage as compared to 19.58% for state coverage, 21.27% for statement coverage and 25.01% for branch coverage.

DISCUSSION

The proposed coverage metric manages ambiguity in control-flow that arises in presence of an error. Our algorithm identifies a subset of control-flow paths, which may be executed due to an error and determines error propagation for each path. Accuracy is gained in error propagation by considering all possible control flow paths. Our algorithm reduces the complexity significantly (up to 86.9%) pruning infeasible control-flow paths using error propagation information. Our technique handles designs with concurrent processes and its accuracy can be further improved by enhancing design errors models.

We can deduce from the results that COCM coverage is closer to error coverage than the coverage obtained by state, statement and branch coverage metrics. The standard deviation between COCM and error coverage is smaller as compared to deviation from the other coverage metrics. This shows that COCM is a better metric, as it is not over estimating detection of errors for control flow oriented designs.

CONCLUSION

We have presented a control-flow oriented coverage metric to measure coverage in multi-process designs with complex control-flow with high accuracy. Our metric analyzes the meaningful control flow paths, while ignoring the infeasible ones. We also presented a

methodology to evaluate the metric by analyzing its ability to detect design errors. The research presented in this study can be applied to any complex design with a bounded number of concurrent processes.

REFERENCES

1. Beizer, B., 1990. Software Testing Techniques. 2nd Edn., Van Nostrand Reinhold, New York, ISBN: 0-442-20672-0, pp: 550.
2. Bentley, B., 2001. Validating the Intel Pentium 4 microprocessor. Proceedings of the 38th Conference on Design automation, (CDA'01), Las Vegas, Nevada, United States, pp: 244-248. http://portal.acm.org/citation.cfm?id=378473
3. Cheng, K.T. and J.Y. Jou, 1992. A functional fault model for sequential machines. Trans. Comput. Aided Des. Integ. Circ. Syst., 11: 1065-1073. DOI: 10.1109/43.159992
4. Corno, F., M.S. Reorda, G. Squillero, A. Manzone and A. Pincetti, 2000. Automatic test bench generation for validation of RT-level descriptions: An industrial experience. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, Mar. 27-30, IEEE Xplore Press, Paris, France, pp: 385-389. DOI: 10.1145/343647.343802
5. Fallah, F., S. Devadas and K. Keutzer, 1998. Occom: Efficient computation of observability-based code coverage metrics for functional verification. Proceedings of the Design Automation Conference, June 15-19, IEEE Xplore Press, USA., pp: 152-157. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=724457&isnumber=15604
6. Gaudette, E., M. Moussa and I. G. Harris, 2003. A method for the evaluation of behavioral fault models. Proceedings of the 8th IEEE International Workshop on High-Level Design Validation and Test, Nov. 12-14, IEEE Computer Society, Washington, DC., USA., pp: 169-172. http://portal.acm.org/citation.cfm?id=1114537
7. Hayek, G.A. and C. Robach, 1996. From specification validation to hardware testing: A unified method. Proceedings of the International Test Conference, Oct. 20-25, IEEE Xplore Press, Washington, DC., USA., pp: 885-893. DOI: 10.1109/TEST.1996.557150
8. King, K.N. and A.J. Offutt, 1991. A Fortran language system for mutation-based software testing. Software Pract. Eng., 21: 685-718. DOI: 10.1002/spe.4380210704
9. Laski, J. and B. Korel, 1983. A data flow oriented program testing strategy. IEEE Trans. Software Eng., SE-9: 33-43. DOI: 10.1145/947955.947963
10. Lv, T., J. Fan and X. Li, 2003. An efficient observability evaluation algorithm based on factored use-def chains. Proceedings of the 12th Asian Test Symposium, Nov. 16-19, IEEE Xplore Press, USA., pp: 161-166. DOI: 10.1109/ATS.2006.260998
11. Moundanos, D. *et al*., 1998. Abstraction techniques for validation coverage analysis and test generation. Trans. Comput., 47: 2-14. DOI: 10.1109/12.656068
12. Verma, S., K. Ramineni and I.G. Harris, 2005. An efficient control-oriented coverage metric. Proceedings of the Asia South Pacific Design Automation Conference, Jan 18-21, IEEE Xplore Press, USA., pp: 317-322. DOI: 10.1109/ASPDAC.2005.1466181