Comparing Code Coverage Metrics for Analog Behavioral Models

Andreas Fürtig[1], Moritz Paschke[1] and Lars Hedrich[1]

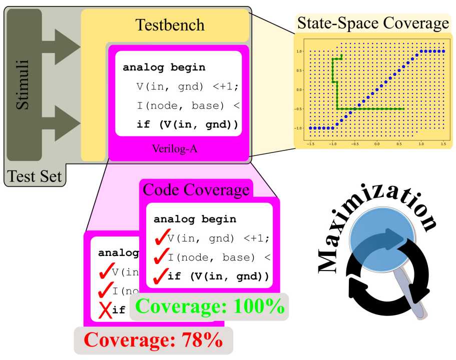
[1]Institute for Computer Science, Goethe Universität Frankfurt a. M., Germany
[1]Email: {fuertig, paschke, hedrich}@em.cs.uni-frankfurt.de

***Abstract*—In current applications Analog/Mixed-Signal (AMS) circuits gets increasingly demanding. To speed up the design process parts of the design were implemented in hardware description languages. Besides positive aspects like simulation processing times these models need to be checked in terms of verification run set completeness, i.e. input stimuli, parameter setting, and test bench circuitry. For this purposes we present a methodology to adopt code coverage metrics on Verilog-A models. A public domain analog circuit simulator automatically instruments and executes the behavioral description. The coverage results are automatically annotated and compared to a coverage metric based on the reachable analog state-space of the circuit. We discuss the methodology on several examples and sketch a path to improve the completeness of a verification run set.**

I. Introduction and State of the Art

Verification is getting more and more demanding due to the increasing complexity of digital and especially analog mixed-signal (AMS) designs. The decision when a design is ready to produce depends on the knowledge of sophisticated engineers who creates test patterns in a diversified way. The amount of tests is only bounded by a full formal verification of the design which does not seem possible due to runtime constraints.

But lets take a step back: Complex AMS designs are implemented in a behavior description language like SystemC-AMS or Verilog-AMS. These systems can be split into different (logic) subsystems, mostly digital and analog parts. Testing digital parts can be done in a suitable time frame while analog parts take much more effort due to the continuous characteristics of the signals. Using Verilog-AMS as a description language has many similarities to the progress of creating software projects. There are a lot of software testing techniques available [1]. One measure to express the process of testing in the software environment is coverage [2]. Today, this measure is available for nearly every available programming language like C, C++, Java, etc. and more or less state of the art. These techniques could be easily adopted to the pure digital design process in the early stages as well. Jou *et al.* [3] give a complete overview over different coverage metrics like statement, decision, event, expression coverage and many more. For more demanding purposes, very complex coverage metrics exists. For example, testing and verification of aviation software products uses modified condition / decision coverage [4] to fulfill the security aspects. The complexity is in general exponential, e.g. in the number of state variables for FSMs of a digital circuit [5] or in the number of decisions in full path coverage. Coverage metrics for analog designs are not available due to the heterogeneous characteristics of the signals and elements in a full system. First attempts were made by discretization of a netlist implementation and the underlying DAE system [6]. Comparing pure digital, binary systems with full continuous ranges of inputs, loads etc. prevents the usage of digital measurements in the analog world. But since it is possible to describe analog circuits with Verilog as well, basic coverage measurements [7] were already adopted with respect to the metric itself and how to taking the continuous value range into account. Due to the enormous range of available functions and the complexity of analog circuits, manually detecting different operating points for nearly every line of code seems not reasonable.



```
analog begin
  V(in, gnd) <+1;
  I(node, base) <
  if (V(in, gnd))
```

Fig. 1. Overview of the presented methodology. The usage of different coverage metrics helps to create a test set for a well tested and bug free behavioral model.

It should be clear that coverage itself is only one type of measurement describing the quality of a set of test benches and input patterns (in the following called *test set*) to a complex system. However a coverage metric does only assess a certain aspect of the test set and not an overall quality metric. A fully covered design (coverage equals to 100%) finally does not ensures a bug free design.

We will focus our contributions in this paper on Verilog-A which is simulated with a public domain analog simulator GNUcap [8]. To simulate the behavioral model, the Automatic Device Model Synthesizer (ADMS) [9] is used to transform Verilog-A (and Verilog-AMS as well) description to a data tree, containing all needed properties and physical informations. As all used software in this contribution is open source, we are able to automatically instrument the compilation process which we will need in the following.

II. Analog Coverage Definitions and Corresponding Methodologies

One main goal of coverage analysis is to give a measure of how much of a design or implementation was inspected by a test set. In this chapter we will firstly

```
1  module rc(in,out,g);
2   parameter real c=1e-6 from [0:inf);
3   parameter real r=1e3 from [0:inf);
4   analog begin
5    I(in,out) <+ V(in,out)/r;
6    I(out,g) <+ c*ddt(V(out,g));
7
8    if(V(out,g) > 1.0)
9      I(out,g) <+ ((V(out,g)-1.0)/10);
10
11   if(V(out,g) < -1.0)
12     I(out,g) <+ ((V(out,g)+1.0)/10);
13  end
14 endmodule
```

Listing 1. Verilog-A model of a limited RC lowpass filter.

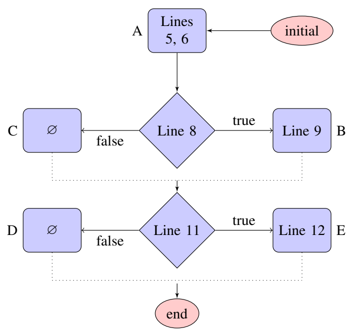


Fig. 2. Control flow graph of the RC lowpass filter in Verilog-A.

give a short overview of existing coverage metrics and adopt them to the concept of Verilog-A behavioral models and secondly revise an analog coverage to combine both metrics. Fig. 1 illustrates the presented methodology. To measure the executed lines of the behavioral model we instrument the ADMS and GNUcap code. Depending on the granularity of the wanted coverage metric, different type of instrumentation is needed, but if there is a chance to record the full path of execution of a Verilog-A model at each simulation step, all thinkable coverage metrics can be computed offline afterwards. Details on the runtime of this method are presented in Section III.

Listing 1 shows the codelines of Verilog-A model for a very simple RC lowpass filter which is limited to the range $[-1V, 1V]$ consisting of four statements, two *if*-branches, two parameters and as well some Verilog-A specific code. We will demonstrate our methods in this section on this small example.

A. Analog Code Coverage

The simplest form of coverage in the digital domain is the statement or line coverage. This very straight forward method points a designer to parts of the implementation which were not executed by a test set. Missing lines of code can be based on some misleading input stimuli or a wrong test bench. A direct adaption of this coverage metric fails at a very basic mechanism of analog circuit simulators. These simulators solve complex differential equations by several numerical methods, for example a Newton iteration. It is very important not to count every iteration step rather to include the final timestep into concideration only. Codelines which can be exectued in a simulation step are counted in this metric only, skipping comments, parameter definitions, control statements, etc. For the given example this reduces the number of codelines to $|\text{all statements}| = 4$. The statement coverage $\zeta_{\text{statement}}$ is expressed as the ratio

$$\zeta_{\text{statement}} = \frac{|\text{executed statements}|}{|\text{all statements}|}. \quad (1)$$

In addition to this simple form we can define a branch coverage. It does not count simple code lines but has an more abstract view on the code, counting which branches are visited and which not. If an implementation deals with a given range of values in the enabling condition of the *if* statements, this coverage metric forces the designer to think about the complementary values as well. In contrary to $\zeta_{\text{statement}}$ the nominator of this coverage is not only the sum of all available branches in the implementation. Moreover missing *else* branches need to be counted as well even if missing in the code. Illustrated at our example in Listing 1 the sum of branches evalutated to 4 due to missing *else* branches. A branch itself was executed, if all statements in this branch were covered by a simulation. The branch coverage ζ_{branch} is expressed as the ratio

$$\zeta_{\text{branch}} = \frac{|\text{executed branches}|}{|\text{all branches}|}. \quad (2)$$

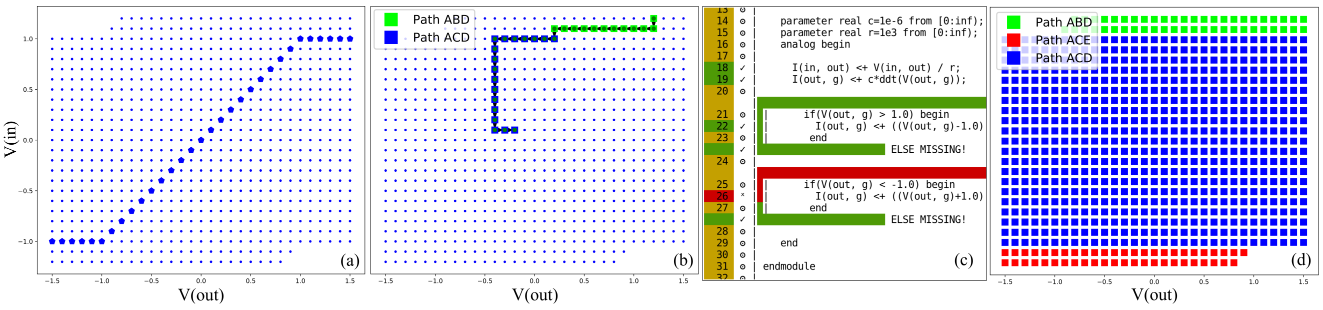
A coverage metric which depends much more on the actual value of variables and depending on what path is evaluated through the model is called path coverage. The path coverage ζ_{path} is expressed as

$$\zeta_{\text{path}} = \frac{|\text{executed paths}|}{|\text{all paths}|}. \quad (3)$$

Obviously, computation of Equation 3 is very complex, since the sum of paths is increasing exponentially with every *if* statement. In [10] a so called *Control Flow Graph (CFG)* for Verilog models are introduced which helps to detect all possible paths in a behavioral model. Fig. 2 is the CFG for the RC lowpass filter described before. As block A is executed in any case, the set of possible paths is $\{ACD, ACE, ABD, ABE\}$. Having a closer look on the result, we will later on see that reaching a full path coverage is often not possible. For example, contrary conditions on *if* statements will prevent that all paths are reachable. In our example (see Fig. 2) it is not possible to reach more than 3 paths. Path ABE is not executable.

B. State-Space Coverage

In addition to code based coverage metrics, a state-space coverage metric can be defined and may lead to a different point of view on the quality of a test set. In [11] a discretization of the full state space of an analog circuit is presented. Using an analog circuit simulator with full SPICE accuracy, a discrete state space model $M_{ATS} = (\Sigma, R, L_V, T)$ can be computed, including a finite set of states (Σ), transitions between those states (R), labeled with a positive or zero valued time for that transition (T) as well as a labeling functions L_V storing all state space variables and inputs of the DAE system. In the same way, the simulator can create such a state space for Verilog-A models. Fig. 3 (left) shows a discrete state space for the RC lowpass filter described previously, including the DC operating points of the filter. A stimulus created for a test bench now leads to a transient response which can directly be mapped to the discrete state



```
14 ⊙ |    parameter real c=1e-6 from [0:inf);
15 ⊙ |    parameter real r=1e3 from [0:inf);
16 ⊙ |    analog begin
17 ⊙ |
18 ✓ |      I(in, out) <+ V(in, out) / r;
19 ✓ |      I(out, g) <+ c*ddt(V(out, g));
20 ⊙ |
     |
21 ⊙ | |      if(V(out, g) > 1.0) begin
22 ✓ | |        I(out, g) <+ ((V(out, g)-1.0)
23 ⊙ | |       end
   ✓ |                  ELSE MISSING!
24 ⊙ |
     |
25 ⊙ | |      if(V(out, g) < -1.0) begin
26 × | |        I(out, g) <+ ((V(out, g)+1.0)
27 ⊙ | |       end
   ✓ |                  ELSE MISSING!
28 ⊙ |
29 ⊙ |    end
30 ⊙ |
31 ⊙ | endmodule
```

Fig. 3. Mapping between code, branch, path and state-space coverage. (a) Full discretization of the state space of the implementation. (b) A trajectory through the state space with colorized paths back-annotated from the Verilog-A model ($\zeta_{\text{statement}} = 75\%$, $\zeta_{\text{branch}} = 75\%$, $\zeta_{\text{path}} = 66\%$, $\zeta_{\text{state-space}} = 9.6\%$). (c) Annotated Verilog-A code corresponding to that trajectory. (d) Situation after a full state-space coverage have been executed with back-annoted colorized paths ($\zeta_{\text{statement}} = 100\%$, $\zeta_{\text{branch}} = 100\%$, $\zeta_{\text{path}} = 100\%$, $\zeta_{\text{state-space}} = 100\%$).

space [6] leading to a set of states C covered by this trajectory. The state-space coverage $\zeta_{\text{state-space}}$ can be defined as the ratio

$$\zeta_{\text{state-space}} = \frac{|C|}{|\Sigma_R|} \tag{4}$$

with Σ_R being the reachable set of all states Σ.

C. Mapping Between Different Coverage Metrics

With the analog code coverage metrics $\zeta_{\text{statement}}$, ζ_{branch} and ζ_{path} on the one hand and the state-space coverage $\zeta_{\text{state-space}}$ on the other hand we are now able to investigate the relation between those metrics. Since a behavioral description of an analog device is typically included inside a test bench circuit (voltage source, resistors, etc.) these components influence the simulation results in different ways. Our method shows this relation, guiding the designers to missing pieces of the implementation and test set.

Figure 3 illustrate the proposed method: (a) is the full state space of the simple RC lowpass filter. The stationary DC points are marked blue. (b) a stimulus is simulated by the simulator, resulting in a trajectory through the state space. Each simulation step can be directly matched to a path through the implemented behavioral code (c), showing lines evaluated or not. Additionally we can back-annotate these paths in to (b) and colorize them to see graphically in the state space the location of the reached paths. (c) clearly shows where the not covered branches were: In this case line 26 is not reached. The covered lines are marked with red and green bars in the left column, the branches are annotated around the *if* statements. In order to guide the simulation into a not covered branch one could change the stimulus, the test bench or some parameter. (d) is created by applying a coverage maximization method [6], where each state in the full state space was covered, forming a direct mapping of the state space to the Verilog-A implementation.

This analysis allows a direct comparsion of the described coverage methods:

1) Branch vs. Statement Coverage: At first sight branch and statement coverage have the same significance. But statement coverage is able to measure coverage of visited lines already present in the code only while branch coverage counts branches explicitly missed by the designer. Hence branch coverage is the more significant measure.

2) Branch vs. Path Coverage: It becomes clear that both coverage metrics take non existing branches into account, assuring that the designer don't miss anything important. But as every *if* statement leads to an exponential growth in the overall amount of paths, the computation for non-trivial implementations is not possible. Another strong point against path coverage is the amount of unreachable paths, which occurs due to contrary statements, branches, etc. Even in our small RC example the sum of all paths reduce to three, as both *else* branches express the opposite voltage range. The main problem is that the possible contrary conditions can depend on any intermediate variables and conditions. Hence a symbolic analysis seems to be necessary to detect and proof an unreachable path in general.

3) Branch- vs. State-Coverage: In this simple RC-example we can reach 100% state-space coverage and 100% branch coverage. However this is not always the case: If we make our example a little bit more difficult by introducing additional parameters or having multiple electrical ports being constraint by the test bench circuitry, we can have 100% state-space coverage for a given parameter set in a given test bench, while the branch coverage is not 100%. This is a clear hint, that our test bench or parameter set in combination with the given input stimuli is not sufficient to test all code of the Verilog-A model. By inspection of the not covered statements/branches we can guess what to change in the test bench. In the next section this is done for a complex EKV transistor model.

III. Experiments

The EKV transistor model [12] is – as many other transistor models – available in a Verilog-A implementation [13] with 730 lines of code. We put these Verilog-A model into a testbench to just characterize the transistor by tuning the V_{GS} and the V_{DS} of the transistor. The drain current I_{DS} is measured and plotted versus the voltages. With this test set, we start calculating coverage metrics by using a DC-Operating point (EKV test set with id 3 in Table I). This leads to a very low state-space coverage and a low branch coverage – here we will concentrate on branch coverage and state-space coverage. Adding some automatically generated paths to increase state-space coverage leads to test set 4, clearly increasing branch coverage. However, all other input stimuli increase the state space coverage further up to 100% as shown with test set 8 but the branch coverage reaches at maximum 67.2% indicating that we have to change the testbench or parameter set. This is done be putting different voltages on the source node in test sets $5-7$. Finally we are able to get 100% branch coverage with the test sets $3-7$.

TABLE I

RESULTS OF THE PRESENTED COVERAGE METRICS. REMARK: RESULTS FOR $\zeta_{\text{STATEMENT}}$, ζ_{BRANCH}, ζ_{PATH} ARE ONLY AVAILABLE WITH INSTRUMENTATION OF THE VERILOG-A IMPLEMENTATION. RUNTIME WITHOUT INSTRUMENTATION IS ONLY LISTED FOR COMPARISON.

DUT	Id	Testbench	Parameter	Stimuli	Runtime [s] without / with instrumentation	$\zeta_{\text{statement}}$	ζ_{branch}	ζ_{path}	$\zeta_{\text{state-space}}$
RC Lowpass	1	standard	default	full stim. [6]	0.007 / 0.008	100%	100%	3/3	100%
OP	2	standard	default	full stim. [6]	0.01 / 0.01	100%	100%	9/9	100%
EKV	3	$(vs = 0)$	*	DC	0.01 / 0.01	65.7%	39.7%	1/91	0.2%
	4	$(vs = 0)$	†	pwl gen.	0.07 / 0.08	85.7%	67.2%	13/91	4.9%
	5	$(vs = vd - 0.1)$	†	pwl man.	0.01 / 0.02	83.4%	60.3%	11/91	8.4%
	6	$(vs = vd)$	⌀	pwl man.	0.01 / 0.01	81.7%	46.5%	7/91	7.2%
	7	$(vs = vd - 1)$	†	pwl gen.	0.05 / 0.05	87.4%	65.5%	15/91	30.1%
	3-7				0.17 / 0.21	100%	100%	47/91	40.7%
	8	same as Id. 3		full stim. [6]	0.80 / 0.95	88.5%	67.2%	45/91	100%

(*) Parameters AD,PD,CJ,AS,JS of transitor model at slightly changed values. (†) Additionally parameter CJSW shifted.
(⌀) Parameters AD,PD,CJ,AS,JS on default values.

The back-annotated path from test set 4 of the EKV model is shown in Fig. 4 left. We can see that a lot of different paths (distinguishable by color code) are visited. However a full inspection, corresponding to test sets $3 - 8$, leads to a complete covering shown in Fig. 4 right.

Finally we applied our method also on a Verilog-A model of an operational amplifier (OP in Table I). As this model does not use many tunable parameters and due to its simplicity we reached with the first try (test set 2) either 100% branch and state-space coverage. A change in the test set by its external connection was not needed.

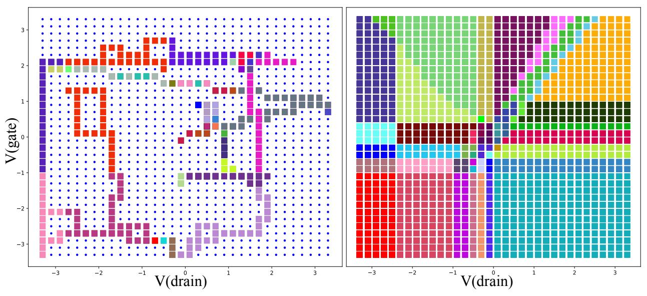


Fig. 4. Back-annotated path for the EKV model. Left: Path from test set 7. Right: All paths from test sets 3-8 showing very nicely different regions of the branches in the state space.

The overhead in terms of runtime for coverage calculations during simulation can be extracted from column 6 and is in all cases numerically below 20% recording all 3 code coverage metrics at one single run.

IV. CONCLUSION

This paper presents code coverage metrics for behavioral models of analog circuits and a full automatic technique for computing these metrics. Additionally we compare the introduced metrics (line, branch, path coverage) with each other and with state-space coverage. The theory and evaluation on the examples show that the branch coverage and the state-space coverage are the important measures which can help and guide the designer to a well tested behavioral model and bug free behavioral code. The path coverage has no real use, as the number of possible paths could not be calculated for realistic examples leading to a not determined coverage goal.

The branch coverage has it's strength in judging if all code branches and lines are visited and lead to a set of test benches and parameter shifts which visit at least once each branch. A flaw could be nonlinear functions like already mentioned in [7]. For this we use the state-space coverage to analyse their excitation in different regions which can be also done automatically and reduce the effort for stimuli creation drastically. If both coverages reach 100% the possibility that some bug is still unexcited in the code is very low. Bigger examples can be processed with code coverage results as well as the runtime analysis suggests. As much more sophisticated circuits are composed of smaller subcircuit components very often, a detailed investigation of these single components using the proposed state space coverage method should be sufficient and will be discussed in future work.

Additionally, we want to extend the methodology for nonlinear functions in the behavioral code by treating them in a similar way as branches. Additionally it seems to be useful to develop some automatic methods to generate stimuli and test benches/parameter sets in order to maximize the branch coverage.

REFERENCES

[1] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[2] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. Springer Publishing Company, Incorporated, 1st ed., 2007.

[3] J.-Y. Jou and C. Liu, "Coverage analysis techniques for hdl design validation," *Proc. Asia Pacific CHip Design Languages*, 1999.

[4] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A Practical Tutorial on Modified Condition/Decision Coverage," tech. rep., 2001.

[5] R. Ho and M. Horowitz, "Validation coverage analysis for complex digital designs," in *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers*, Nov 1996.

[6] A. Fürtig, S. Steinhorst, and L. Hedrich, "Feature based State Space Coverage of Analog Circuits," in *Proceedings of the Forum on Specification and Design Languages (FDL 2016)*, 2016.

[7] Y.-B. Sha, M.-S. Lee, and C.-N. Liu, "On code coverage measurement for Verilog-A," in *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pp. 115–120, Nov 2004.

[8] A. T. Davis, "An overview of algorithms in Gnucap," in *University/Government/Industry Microelectronics Symp.*, pp. 360–361, 2003.

[9] ADMS Automatic Device Model Synthesizer, "https://sourceforge.net/projects/mot-adms/."

[10] M. Zaki, Y. Mokhtari, and S. Tahar, "A path dependency graph for Verilog program analysis," in *Proceedings of the IEEE Northeast Workshop on Circuits and Systems (NEWCAS'03)*, 2003.

[11] S. Steinhorst and L. Hedrich, "Trajectory-directed discrete state space modeling for formal verification of nonlinear analog circuits," in *Proceedings of the ICCAD'12*, ACM, 2012.

[12] C. C. Enz, F. Krummenacher, and E. A. Vittoz, "An Analytical MOS Transistor Model Valid in all Regions of Operation and Dedicated to Low-Voltage and Low-Current Applications," *Journal of Analog Integrated Circuit and Signal Processing*, no. 8, pp. 83–114, 1995.

[13] Nielsen,I. R. and Warning, D., "Epfl-ekv Version 2.6: Verilog-A Implementation," *http://legwww.epfl.ch/ekv*, 2006.