Evaluation of Toggle Coverage for MVL Circuits Specified in the SystemVerilog HDL

Mahsan Amoui, Daniel Große*, Mitchell A. Thornton, Rolf Drechsler*
Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275
**Institute of Computer Science,University of Bremen, 28359 Bremen, Germany*
*{mahsan, mitch}@engr.smu.edu , *{grosse,drechsle}@informatik.uni-bremen.de*

Abstract

Designing modern circuits comprised of millions of gates is a very challenging task. Therefore new directions are investigated for efficient modeling and verification of such systems. Recently, a new language, SystemVerilog, was introduced and became an IEEE standard. SystemVerilog extends the hardware description language Verilog by including higher abstraction levels and integrated verification features. In this paper, we first present the concept of modeling multiple valued logic circuits in SystemVerilog. We demonstrate that this approach allows for efficient simulation of complex multiple valued logic systems. Secondly, we show how SystemVerilog can be used to ensure functional correctness. A generalization of binary toggle coverage for the multiple valued logic domain is presented and evaluated. As a test case, a scalable multiple valued logic arithmetic unit is modeled and experimental results for multiple valued logic toggle coverage are given.

1. Introduction

The rapid development of complex systems requires a reliable modeling and verification approach. Currently, a gap exists in terms of abstraction between the languages used to describe hardware at the Register Transfer Level (RTL) and design languages used to describe the first reference design. Verilog and VHDL are typically used to describe hardware at the RTL and a different language (*C*, *C*++) is commonly employed for the first reference design. This results in a significant amount of effort since the same code has to be written at two or more different levels of abstraction in multiple programming languages. A new language, SystemVerilog [2], aims to bridge this gap by extending Verilog to higher levels of abstraction for architectural and algorithmic design and for advanced verification [4, 6]. There has been a significant past effort in the development of methods for designing Multiple Valued Logic (MVL) circuits. Modeling circuits based on MVL instead of the binary domain can help during verification [10]. Thus, a development environment that supports modeling of MVL circuits is required. A first approach in this direction was introduced in [5] where a package to model ternary circuits in VHDL was provided. In [1] a modeling platform using SystemC was presented. In addition to modeling MVL circuits, their correct functional behavior must also be ensured. Therefore in [1] pure simulation is used but the completeness of the verification process is not discussed.

For simulation-based design validation, an estimate of the functionality that is exercised can be given using coverage metrics [8]. A metric based on the circuit structure and one that is well known for the binary case is toggle coverage [7]. The basic idea is to check whether the value of an internal net toggles between all possible values. For the MVL domain this metric has not been studied previously.

In this paper we present a concept for modeling MVL circuits in SystemVerilog. The concept is demonstrated for a scalable MVL Arithmetic Logic Unit (ALU). To determine the quality of the verification process we introduce a generalization of toggle coverage for the MVL domain. We show that two types of toggle coverage should be distinguished if the underlying logic is MVL. Next, the realization of the method including the instrumentation of the SystemVerilog code for toggle coverage calculation is described and experimental results are given.

The remainder of this paper is structured as follows. In Section 2, we present the basics of MVL, MVL modeling in SystemVerilog, and a scalable MVL ALU as a case study. In Section 3, MVL toggle coverage and the implementation of the toggle calculations in SystemVerilog are shown. In Section 4, we present simulation results in terms of toggle coverage and, finally in Section 5, we conclude the paper.

2. MVL Modeling in SystemVerilog

2.1. Multiple Valued Logic Circuit

MVL is a non-binary logic that involves switching between more than 2 values. We will assume that MVL primitive devices will be limited to 2-input/single-output functions. For example, a logic

function with radix-3 is one that has two inputs that can assume three values (i.e. 0, 1, or 2) and generates one output signal that can have one of these three values. Symbolically, a logic function based upon a single radix-3 device can be graphically depicted as shown in Figure 1. Table 1 shows an example truth table of such a device.

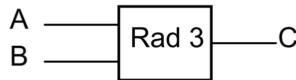

Figure 1. Logic function with radix 3

Table 1. Truthtable of a function with radix 3

	0	**1**	**2**
0	1	0	2
1	0	2	1
2	2	1	0

The example function in Table 1 is commutative, and hence it would not matter if we exchange the 'A' and 'B' inputs. MVL circuits can be modeled as graphs where the edges correspond to nets and the vertices correspond to storage elements or MVL basic gates [3]. All of the basic MVL gates can be used as an operator in higher level descriptions of a circuit. More complex constructs can be defined based on the MVL gates and can be used in the high level description as well.

Given *r* as the radix, the operators (which correspond to basic gates) are defined on a set of $O = \{0,\ldots,r-1\}$. We chose a functional complete set of operators [1], all of them having one output and one or two operands *x* and *y* $(x, y \in O)$ as their inputs. The operators and their functionalities are shown in Table 2.

Table 2. Operators and their functionalities

Operators	Functionality
$MIN(x,y)$	$Minimum\ \{x, y\}$
$MAX(x,y)$	$Maximum\ \{x, y\}$
$INV(x)$	$(r-1)-x$
$EQUAL(X,Y)$	$EQUAL\ \{x, y\}$
$Literal(x,y)$	$Literal_{a,b}(x) = \begin{cases} k-1, if\ a \le x \le b \\ 0, otherwise \end{cases}$

In the binary case MIN, MAX and INV correspond to AND, OR and INV, respectively.

2.2. MVL in SystemVerilog

SystemVerilog is an enhancement to the IEEE 1364 Verilog-2001 standard. These enhancements provide new capabilities for modeling hardware at the RTL and system level, along with a rich set of new features for verification. In this subsection we explain the basics and used constructs of SystemVerilog before we discuss the details of MVL modeling.

2.2.1 Task and functions

Verilog-2001 has static and automatic tasks and functions. Static tasks and functions share the same storage space for all calls to the tasks or function within a module instance. Automatic tasks and function allocate unique, stacked storage for each instance. As an enhancement, SystemVerilog adds the ability to declare automatic variables within static tasks and functions, and static variables within automatic tasks and functions. It adds several new functions to the existing Verilog. *C* data types such as **int**, **typedef**, **struct**, pass-by subroutine arguments, and built-in types such as string, associative arrays, and dynamic arrays can be used to provide better compactness of the code.

SystemVerilog supports classes, the object-oriented mechanisms that provide abstraction, and safe pointer capabilities as well as property and sequence declarations, assertions and coverage statements with action blocks [9].

Also, in SystemVerilog, each formal argument has a data type which can be explicitly declared or can inherit a default type. The task argument default type in SystemVerilog is logic.

SystemVerilog operators are a combination of Verilog and *C* operators, where the type and size of the operands is fixed, and hence the operator is of a fixed type and size. The fixed type and size of operators is preserved in SystemVerilog which allows efficient code generation.

For more details on SystemVerilog see the SystemVerilog website and Accellera website [2, 9]. The C-like functions shown in Fig. 2, are provided to realize the basic MVL operations for the one-digit MVL type as well as the vector MVL type.

```
function int min(int a,int b);
    int res;
    res = ((a < b) ? a:b);
    return res;
endfunction

function int max(int a,int b);
    int res;
    res = ((a > b) ? a:b);
    return res;
endfunction

function int inv(int a);
    int res;
    res = ((r -1)-a);
    return res;
endfunction
```

Figure 2. Functions of arbitrary MVL circuits

These functions can be used by the designer in their particular design to implement the basic operations,

regardless of the radix or the width of the operands. Due to space limitations only the functions for the operators MIN, MAX, and INV are shown. An MVL vector is defined using an integer array that is supported by SystemVerilog. The presented functions can be used for each position in the MVL vector. More complex operations such as addition can be also implemented easily by formulating such operations as SystemVerilog functions using the MVL vector as input. In the next section we demonstrate this modeling approach for a scalable MVL design of an Arithmetic Logic Unit (ALU).

2.2.2 Case Study – MVL ALU

A scalable MVL ALU circuit for different radices is implemented in a SystemVerilog description. The block diagram of the MVL ALU circuit is shown in Fig. 3.

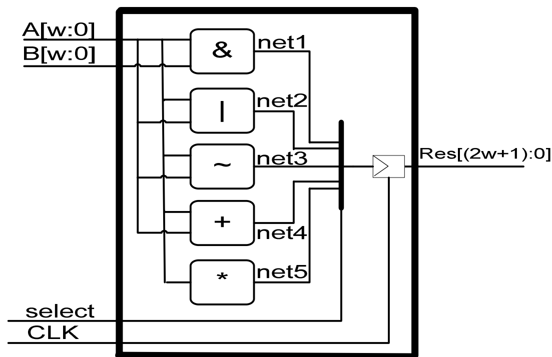

Figure 3. ALU circuit

```
task compute ();
begin
 case (i_sel)
0: //Calculating the MIN of two buses
   begin
    for (i=0; i<width; i++ )
             begin
              alub.ALU_result[i] =
                    min(alub.a[i], alub.b[i]);
             end
   end
1://Calculating the MAX of two buses
   begin
    for (i=0; i<width; i++ )
             begin
      alub.ALU_result[i] =
       max(alub.a[i], alub.b[i]);
             end
   end
  …
end
endtask // compute
```

Figure 4. Implementing the MIN, MAX operation

The ALU has four inputs: clock, two MVL operand inputs A and B, and the control input select. The ALU can perform the INV, MIN, MAX, addition and multiplication operations. The output is stored in the register named Res. The width of the operands A and B is equal and the result bus Res has double width to accommodate the result of a multiply operation. The SystemVerilog implementation of the ALU is scalable since the width and radix are separately defined and can be easily varied.

To demonstrate the efficiency of SystemVerilog code, we show some portions of the code used to implement the ALU circuit in Fig. 4. The task compute is called every rising edge of the clock.

3. Toggle Coverage

In this section, the notion of toggle coverage for binary circuits is briefly reviewed. Next, the generalization of this concept to the MVL domain is presented. Based on this concept, the basic steps for the computation and analysis of MVL toggle coverage in SystemVerilog are discussed.

3.1. Binary Toggle Coverage

Toggle coverage checks whether, during simulation, the value of a wire/register toggles between all possible values. A net is said to be fully covered if both values have been observed on that net. This metric does not indicate that every value of a bit vector was observed. For example, if a bit vector of size 3 is considered and during the simulation the values 000, 110 and 100 (left most bit is MSB) occur, then the coverage of this bit vector is 83.3%. This is computed as follows: bit 3 and bit 2 assumed both values and bit 0 assumed only one value. Hence in total 5 out of 6 values were assumed and the coverage is 5/6 or 83.3%.

In the following section the generalization of toggle coverage to the MVL domain is discussed.

3.2. MVL Toggle Coverage

If the MVL domain is considered, two different types of logic value toggling have to be considered. We refer to these metrics as type I and type II.

3.2.1. Type I Toggling. Type I checks for possible logic values which have been assumed for a given net.

3.2.2. Type II Toggling. Type II is more comprehensive and checks for the transitions made between all possible logic values that can occur.

As an example of type I toggle coverage, consider the case where the radix is 5. Assuming a MVL vector called “A”, we check if each element has taken all the legal values according to the set {0, 1, 2, 3, 4}. For example, if the first element of A, denoted as A[0], has taken the values 0, 1 and 3 during

simulation, then the value of type I toggle coverage for A[0] is 3/5 or 60%. For the complete vector the coverage is determined as an average value of coverage computed over all elements of A. For type II toggle coverage, consider the following example. Again we have an MVL vector called "A" with radix 5. For the first element A[0] of A one would check that there were transitions between all values of the set {0, 1, 2, 3, 4}; i.e. 0→1, 0→2, 0→3, 0→4, 1→0,…, 4→3. In general radix*(radix-1) transitions have to be considered. The transitions which need to be checked for the example are shown in Table 3, where 'T' indicates a transition of interest whereas 'X' indicates a transition that does not give useful information. Hence, if A[0] has made 10 out the 20 possible transitions, we say that the type II toggle coverage is 50% for A[0]. The coverage for the whole vector A is once again computed as an average over the coverage value of each element of A.

Table 3. Transitions for radix 5

	0	1	2	3	4
0	X	T	T	T	T
1	T	X	T	T	T
2	T	T	X	T	T
3	T	T	T	X	T
4	T	T	T	T	X

3.3. MVL Toggle Coverage in SystemVerilog

To demonstrate our implementation of toggle coverage type I using an example, we consider an MVL vector A of width equal to 4 and radix equal to 3. We check if each element has taken all the legal values according to the set {0, 1, 2}. To perform this computation for vector A, we create an array of size width of A * radix. For example the size of the array is 4*3 = 12. Say, we name this array "cov". If the ith element of A takes a value j, $j \in \{0,1,2\}$ then we set a 1 at $(3*i+j)^{th}$ location of cov. Hence, if the vector A takes the following four values successively during simulation: [0, 2, 1, 0], [2, 0, 1, 1], [1, 1, 0, 0], [0, 2, 1, 2], then the toggle coverage of type I for vector A is 11/12, or 91% as shown in Table 4. It should be noted here that while computing the coverage of the result vector the size of the array to be created will be twice (i.e. width of A * radix * 2) since the result vector is twice as big as A.

The SystemVerilog function to calculate toggle coverage of type I is shown in Fig. 6. This general function can be used for a vector of arbitrary width and arbitrary radix. The argument **COV** is a reference since its content is changed inside the function. Basically the access to **COV** is controlled by an adequate offset computation.

Table 4. Transitions for radix 3, toggle cov type I

Loc	0	1	2	0	1	2	0	1	2	0	1	2
A[0]	1	-	-	-	-	1	-	1	-	1	-	-
A[1]	-	-	1	1	-	-	-	1	-	-	1	-
A[2]	-	1	-	-	1	-	1	-	-	1	-	-
A[3]	1	-	-	-	-	1	-	1	-	-	-	1
Cov	1	1	1	1	1	1	1	1	-	1	1	1

```
function coverToggleI( int net[0:(2*WIDTH)-1],
   ref int cov[0:(WIDTH*RADIX*(2))-1] );
     for (int tctr=0; tctr<(2*WIDTH); tctr++)
         begin
           for (int v=0;v<=RADIX;v++)
             begin
               if (net[tctr]==v)
                 begin
                   cov[RADIX*tctr+v]=1;
                 end
               end
           end
endfunction
```

Figure 6. Calculating toggle cov of type I

We now discuss our implementation of toggle coverage type II again with the example of an MVL vector A and radix 3. In this case 3*2=6 transitions have to be considered between all values of the set {0, 1, 2}; i.e. 0→1, 0→2, 1→0, 1→2, 2→0, 2→1, for each element of A. Hence, for the vector A of width 4, 6*4=24 different transitions need to be checked in order to compute the toggle coverage of type II. If the vector A takes the same four values as before: [0, 2, 1, 0], [2, 0, 1, 1], [1, 1, 0, 0], [0, 2, 1, 2] then the according array content of toggle coverage type II for A is illustrated in Table 5.

After each clock cycle we compare the new value of A with the previous value of A. If the value has changed we set a 1 to the appropriate position, and keep the value of the previous cycle as well. This calculation is more exhaustive in the sense that transitions and not just instantaneous values are considered. We observe that the coverage for vector A is 10/24=42%. As expected this value is lower than the toggle coverage value for type I (91%), since type II of coverage is more exhaustive. Fig. 7 shows the SystemVerilog code for calculating toggle coverage type II.

Again the presented formulation of the function is general because the width and radix of an MVL vector for which the coverage is computed can be arbitrary. The estimation of toggle coverage for an MVL circuit is performed by instrumenting the

Table 5. Transitions for radix 3, toggle cov type II

	0→1	0→2	1→0	1→2	2→0	2→1	0→1	0→2	1→0	1→2	2→0	2→1	0→1	0→2	1→0	1→2	2→0	2→1
A[0]	-	**1**	-	-	-	-	-	-	-	-	-	**1**	-	-	**1**	-	-	-
A[1]	-	-	-	-	**1**	-	**1**	-	-	-	-	-	-	-	-	**1**	-	-
A[2]	-	-	-	-	-	-	-	-	**1**	-	-	-	**1**	-	-	-	-	-
A[3]	**1**	-	-	-	-	-	-	-	**1**	-	-	-	-	**1**	-	-	-	-
Cov	**1**	**1**	-	-	**1**	-	**1**	-	**1**	-	-	**1**	**1**	**1**	**1**	**1**	-	-

Systemverilog design with function calls for the nets of interest. Finally, after simulation and analysis of the coverage results is performed.

```
function coverToggleII(int net[0:(2*WIDTH)-1],
  ref int prev_net[0:(2*WIDTH)-1],
  ref int cov2[0:(2*(WIDTH* RADIX*(RADIX-1)))   1]);
  int off;
   for (int tctr=0; tctr<(2*WIDTH); tctr++)
      begin
        off=0;
         for (int x=0;x<RADIX;x++)
              begin
               for (int y=0;y<RADIX;y++)
                  begin
                    if ( x != y )
                       begin
                       if (prev_net[tctr]==x & net[tctr]==y)
                        begin
                          cov2[(RADIX*(RADIX-1))*tctr+off]=1;
                        end
                         off++;
                    end
              …
  prev_net = net;
endfunction
```

Figure 7. Calculating total average of toggle II cov

4. Simulation Result

In this section MVL coverage is computed for different instances of the MVL ALU.

4.1. Tool Environment

All simulations were run on a Linux machine with a Pentium processor at 2.8 GHz and 512 MB of RAM using Synopsys tools.

4.2. Coverage Estimation

In the following experiments the radix of the ALU is 3 and the width of the input vectors A and B is 4. During the simulation, random values for all inputs of the ALU are generated using the SystemVerilog constrained based randomization technique. Thus, the value of each input is chosen with a uniform distribution out of the legal value set. The results of the operations MIN, MAX, INV, ADD, and MULT are referring to the ALU circuit in Fig. 3 as nets 1-5, respectively. The width of all these nets is 8 (width*2) since the largest result is the product value of the multiplication unit that determines the size of
the other inputs for the multiplexer. In the following we analyze the coverage for these nets. All coverage numbers have been obtained by averaging the results over 100 runs. In Fig. 8, the results for toggle coverage type I are shown. As expected, the coverage for the vectors for operations MIN, MAX and INV did not exceed 66%, since only the 4 lower digits of the result vector are affected and the remaining digits do not toggle. In case of the ADD operation 71% were achieved since 5 lower digits of the result vector toggle. However, in case of the MULT operation the coverage is 100% since all digits of the vector toggle. Similarly, in Fig. 9 we show the toggle coverage of type II.

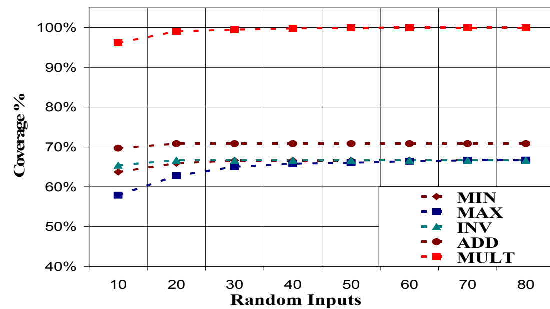

Figure 8. Toggle coverage type I vs. number of inputs, vector width = 4.

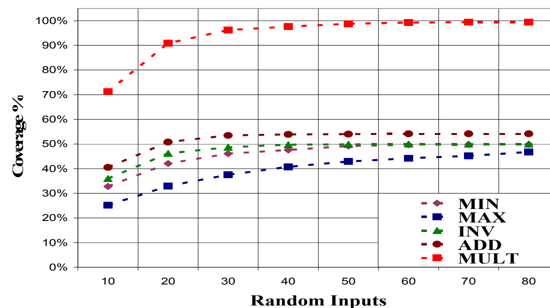

Figure 9. Toggle coverage type II vs. number of inputs, vector width = 4.

Both graphs show that the coverage converges very fast. As expected for the toggle coverage type II more random inputs are necessary to achieve convergence.

In the second series of experiments we introduced an error to the ADD and MULT operation by inserting a design error that is manifested as performing fewer iterations than the number required in the main computation loops. The results are shown in Fig. 10 and Fig. 11. We observe that the coverage drops. Toggle coverage of type I for the MULT operation drops from 100% to 83.33%. Also, in case of the ADD operation, the coverage dropped from 71% to 62.5%. The same can be observed in Fig. 11 for toggle coverage of type II but with an even more noticeable reduction.

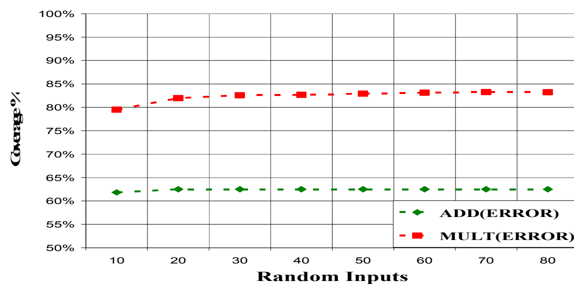

Figure 10. Toggle coverage type 1 with errors in mult & add functions vs. number of inputs, vector width = 4.

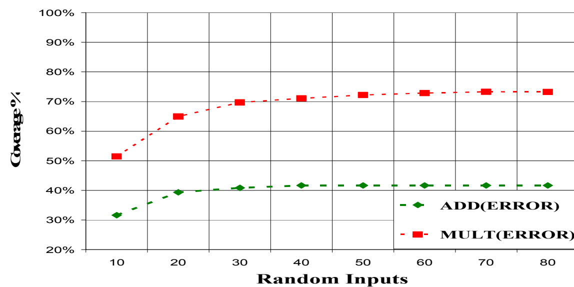

Figure 11. Toggle coverage type II with errors in mult & add functions vs. number of inputs, vector width = 4.

5. Conclusion and Future Work

In this paper we first presented a concept for modeling MVL circuits using SystemVerilog that allows for easy scaling of operand size (in terms of digits) and radix value. A scalable MVL ALU has been implemented and simulated efficiently. To determine the quality of the verification process we generalized the concept of toggle coverage for MVL. It was shown that two types of toggle coverage must be distinguished for the MVL domain. In the experimental results, the convergence of both coverage types for the MVL ALU is empirically shown.

Future directions in this research include developing a general flow to automate the code instrumentation of an MVL SystemVerilog design.

6. Acknowledgment

The authors are grateful for the support provided by the Synopsys Corporation for their generous donation of SystemVerilog language and design tools as well as interactions with Synopsys employees.

7. References

[1] Daniel Große, Görschwin Fey, Rolf Drechsler. *Modeling Multi-Valued Circuits in SystemC.* In International Symposium on Multi Valued Logic, pages 281 – 286, 2003.

[2] The SystemVerilog Homepage. http://www.systemverilog.org

[3] J. Muzio and T. Wesselkamper. *Multiple-Valued Switching Theory*. Adam Hilger, Bristol and Boston, 1986.

[4] Accellera, "SystemVerilog 3.1 Accellera's extensions to Verilog". Npa, CA, 2003, pp.1-2.

[5] C. Rozon, *On the Use of VHDL as a Multi-Valued Logic Simulator.* Royal Military College, Kingston, Ontario, Canada. 1996, IEEE.

[6] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari. *SystemVerilog Assertions Handbook,* VhdlCohen Publishing. Los Angeles, California, 2005.

[7] Dean Drako, Paul Cohen. *HDL Verification Coverage.* EETimes, Global news for creators of technology, 1998.

[8] H. Chokler, O. Kupferman, M. Y. Vardi, *Coverage Metrics for Formal Verification*, Correct Hardware Design and Verification Methods*,* pages 111 – 125, 2003, Springer Berlin/Heidelberg publisher.

[9] Accelera SystemVerilog 3.1a reference manual, Extension to Verilog. Available at: http://www.accellera.org/home

[10] Rolf Drechsler, *Using World-Level Information in Formal Hardware Verification*, Automation and Remote Control, Volume 65, Issue 6, pp. 963-977, June 2004.