Supercharge Your Verification Using Rapid Expression Coverage as the Basis of a MC/DC-Compliant Coverage Methodology

Gaurav Kumar Verma
Mentor Graphics Corporation
Fremont, CA (USA)
gaurav-kumar_verma@mentor.com

Doug Warmke
Mentor Graphics Corporation
Fremont, CA (USA)
doug_warmke@mentor.com

***Abstract*— Technology advances allows for the creation of larger and more complex designs. This poses new challenges, including efforts to balance verification completeness with minimization of overall verification effort and cycle time. It is practically impossible to enumerate all of the conditions and states to do an exhaustive test. Therefore, it is imperative to use well defined criteria to measure and check when the verification is sufficiently complete and meets a reasonable quality threshold. Code coverage is a popular measure of design quality. This paper focuses on expression coverage, which is one of the most complex and least understood types of code coverage, and discusses 'Rapid Expression Coverage' (REC), which is a new metric for expression coverage, while comparing it with some popular metrics being used to evaluate expression coverage in the industry today. Even though this paper describes REC in context of code coverage of designs, these same techniques could also be applied to coverage tools for software languages like C, C++, or Java.**

I. INTRODUCTION

Code coverage is a popular measure of design quality and verification completeness. It has low setup cost and analysis is straightforward, which makes it a high ROI component of most modern verification methodologies. Expression coverage is one of the most complex and least understood types of code coverage. The main question verification engineers need to answer regarding expression coverage is:

"There are 2^N possible input vectors for my N-input expression. I saw a subset of this large number during simulation. How well is my expression tested?"

A variety of metrics are available to help answer this question. Metrics distill the data to a meaningful numeric value that can be analyzed and improved. There are brute force metrics like the sum-of-products and truth table analysis. These require a large number of test vectors for the expression to be covered. Then there are smarter metrics like Focused Expression Coverage, which is a form of Modified Condition/Decision Coverage (MC/DC). These metrics allow an expression to be fully covered with just 2xN input vectors. Most expression coverage metrics are based on truth table implementations, and therefore suffer from capacity limitations due to exponential complexity.

REC is based on partitioning an expression into sub-expressions to derive non-masking conditions, while operating at linear complexity and providing MC/DC-compliant results. This allows any arbitrarily large expression to be considered for coverage, while providing simple and easy to understand reports.

The basis of REC is that an expression input must not be masked by the values of other inputs at the time of coverage collection. For example, 'b' must be '1' while measuring the coverage of 'a' in 'a && b'. If 'b' is 0, the result of the expression gets fixed to '0' and the value of 'a' is not of any significance. The '1' state of 'b' is called its non-masking state, and the associated condition is called the non-masking condition.

As REC collects coverage, it ensures that the input being covered has taken both '0' and '1' states while an appropriate non-masking condition is satisfied. An expression is considered covered when all its inputs have been independently covered.

The following sections go into the details of partitioning an expression to derive non-masking conditions, the effect of short-circuiting, considerations for duplicate input terminals, inverting vs. non-inverting input modes, and uni-modal versus bi-modal expression considerations.

The results of performance benchmarks show REC has higher performance with similar level of comprehensiveness in expression coverage as compared to existing implementations. These results are presented towards the end of the paper.

II. EXPRESSION COVERAGE

Expression coverage measures coverage statistics for expressions and conditions. The fundamental output of the expressions considered for expression coverage should be a

single-bit value. Input terminals can be vectors, as long as they are part of a sub-expression that results in a single-bit value. Consider the following expression as an example:

$$((a > b) \&\& (c == 130)) \quad (1)$$

Expression "(1)" may be considered for expression coverage with (a > b) and (c == 130) as its two input terminals.

For each expression, a set of cases are identified, each case specifying how parts of the expression must take on particular values. Expression coverage then considers whether a simulation exercises each case of the expression. An expression is considered fully covered when all of the individual expression coverage cases are exercised.

Several metrics can be used for expression coverage. Some of the popular ones are listed below.

A. SOP Metric

Sum-of-Products (SOP) checks that each set of inputs that satisfies the expression (results in a '1') must be exercised at least once, but not necessarily independently. It does not check the sets of inputs that results in the expression being evaluated to '0'.

B. UDP Metric

The term UDP is borrowed from the Verilog language, which uses the same basic table format to model user-defined primitives. A UDP table describes the full range of behavior for a given expression. If the conditions described by a row are observed during simulation, that row is said to be hit. All rows in the UDP table must be hit for UDP coverage to reach 100%. Row minimization is attempted by use of wildcard matches.

C. MC/DC Metric

Modified Condition/Decision Coverage (MC/DC) is a popular metric for expression coverage [3]. It is also the basis of some of the other metrics, like FEC. The formal definition of MC/DC as defined by DO-178B is: "Every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decisions outcome. A condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible conditions. [1]"

Modified Condition/Decision Coverage (MC/DC), is a code coverage metric often used by the DO-178B safety critical software certification standard, as well as the DO-254 formal airborne electronic hardware certification standard.

D. FEC Metric

Focused Expression Coverage (FEC) is a row based coverage metric which emphasizes the contribution of each expression input to the expression's output value. FEC measures coverage for each input of an expression. If all inputs are fully covered, the expression has reached 100% FEC coverage. In FEC, an input is considered covered only when other inputs are in their quiescent states, i.e. a state that allows the target input to control the output of the expression. Further, the output must be seen in both 0 and 1 state while the target input is controlling it. If these conditions occur, the input is said to be fully covered. The final FEC coverage number is the number of fully covered inputs divided by the total number of inputs. FEC is fully compliant with the MC/DC coverage metric and can safely be used in a strict MC/DC compliance environment.

III. RAPID EXPRESSION COVERAGE

UDP coverage is very strongly based on the truth table. Covering an expression 100% requires hitting each row in the table at least once. In the worst case, when none of the rows can be merged, UDP requires 2^N input vectors to fully cover an expression with N input terminals. FEC reduces verification effort by only requiring 2xN input vectors in the worst case to fully cover the same expression. This helps verification engineers achieve 100% coverage faster, but it doesn't make the tool implementation any easier. FEC is still based on the truth table, which makes it exponential with respect to the number of input terminals in the expression. It is clearly desirable to move towards constant time and linear complexity solutions. So let's think, can we come up with a better metric that uses the power of FEC to cover an expression using only 2xN input vectors, while still being able to provide a linear complexity solution with respect to the number of input terminals? REC [2] is one such metric. Like FEC, REC is also fully compliant with the MC/DC coverage metric and can safely be used in a strict MC/DC compliance environment.

A. Expression Modes

There are two modes in which an input of an expression can be operating. These are called inverting mode and non-inverting mode.

When setting the value of an input to '0' (or '1'), with all other inputs of the expression in their quiescent states, and the expression evaluates to '1' (or '0'), the input is said to be operating in an inverting mode. On the other hand, when setting the value of an input to '1' (or '0'), with all other inputs of the expression in their quiescent states, and the expression evaluates to '1' (or '0'), the input is said to be operating in a non-inverting mode.

Based on the modes of its inputs, an expression can be classified as either uni-modal or bi-modal. An expression whose inputs only ever operate in one mode, either inverting or non-inverting, is called a uni-modal expression. On the other hand, if an expression has at least one input operating in both inverting and non-inverting modes, it is called a bi-modal expression.

The simplest example of a uni-modal expression is a two-input 'AND' gate.

$$a \&\& b \quad (2)$$

TABLE I. TRUTH TABLE OF AND GATE WITH TARGETS

a	b	a && b	Target
0	0	0	--
0	1	0	a_0*
1	0	0	b_0
1	1	1	a_1, b_1

* The target of a_0 indicates that this row delivers FEC testing when a's value is 0.

In "Table I" for expression "(2)", a_0 evaluates the expression to '0' and a_1 evaluates it to '1'. Similarly, b_0 evaluates the expression to '0' and b_1 evaluates it to '1'. Therefore expression "(2)" is a uni-modal expression with all its inputs operating in non-inverting mode.

Now consider a two-input XOR gate.

$$a \wedge b \qquad (3)$$

TABLE II. TRUTH TABLE OF XOR GATE WITH TARGETS

a	b	a ^ b	Target
0	0	0	a_0, b_0
0	1	1	a_0, b_1
1	0	1	a_1, b_0
1	1	0	a_1, b_1

In "Table II" for expression "(3)", a_0 evaluates the expression to both '0' and '1'. Similarly a_1, b_0, and b_1 also evaluate the expression to both '0' and '1'. Therefore expression "(3)" is a bi-modal expression, with both its inputs operating in both inverting and non-inverting modes.

B. Theory

Every expression, however complex it may be, can be broken down into N smaller expressions consisting of only one operator each, where N is the number of operators in the expression. We call these expressions 'basic expressions'. Consider the following expression:

$$a \;\&\&\; b \;\&\&\; c \;\&\&\; d \qquad (4)$$

The '&&' operator groups from left-to-right, therefore expression "(4)" can be represented as:

$$a \;\&\&\; (b \;\&\&\; (c \;\&\&\; d)) \qquad (5)$$

Expression "(5)" can be broken down into 3 basic expressions, defined as:

$$a \;\&\&\; EXPR1 \qquad (6)$$

$$b \;\&\&\; EXPR2 \qquad (7)$$

$$c \;\&\&\; EXPR3 \qquad (8)$$

Where EXPR1 is 'b && (c && d)', EXPR2 is 'c && d', and EXPR3 is 'd'. Figure 1 shows the expression tree of "(4)", and highlights its 3 basic expressions.

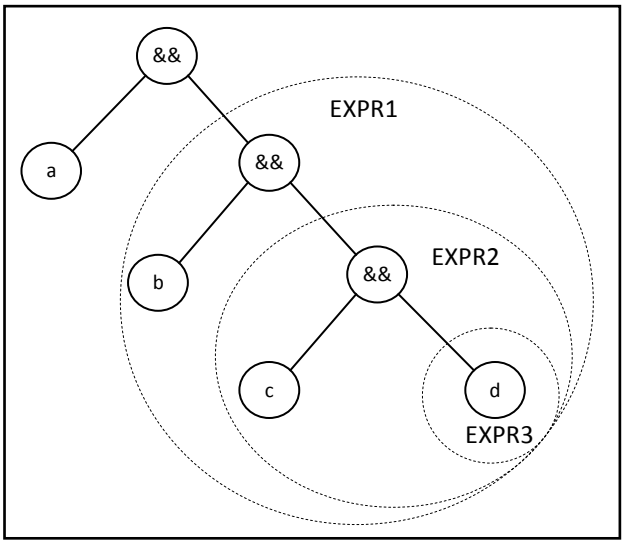

Figure 1. Expression Tree of "(3)" Highlighting Its Basic Expressions

If we work on the truth table of the whole expression, complexity increases exponentially as the expression becomes bigger. Consider the case of a 6 input XOR expression. It has $2^6 = 64$ rows in its truth table. Adding one more input to this expression increases the number of rows to $2^7 = 128$ rows. Adding additional inputs exponentially increases the number of rows, for example a 16 input XOR will consume 65536 rows. Now imagine maintaining a table with all these rows during simulation, and matching all input vectors with rows in this table. This is what exponential complexity metrics like UDP and FEC do.

On the other hand, if we work on basic expressions, the complexity of the solution increases linearly as the expression becomes bigger. A 6-input XOR expression can be broken down into 5 basic XOR expressions. If we add one more input to this expression, the expression can be broken down into 6 basic expressions. If we add one more input, the expression can be broken down into 7 basic expressions.

When working on any input in an expression, it is important that the input must not be masked by the other input because of its value. For example, if we want to measure coverage of 'a' in 'a && b', it is important that the value of 'b' is '1'. If 'b' is 0, the result of the expression gets fixed to '0' and the value of 'a' is no longer of any significance. Also, since the expression is evaluated left-to-right in the presence of short-circuiting, it is correct to only look at the right side of the concerned input. The effect of short-circuiting and non-short-circuit operators is discussed in more detail later in the paper. The assumption is that if we're evaluating the concerned input, it means that its left side is already in a non-masking state. For example, if we're evaluating 'b' in 'a && b && c', it means that 'a' was already evaluated to '1'.

Note that the terminology, 'left side of the input', and 'right side of the input', is being used to make the theory easier to understand. The same theory can be extended to an expression tree as well.

"Table III" lists operators with non-masking states of inputs. Note that the coverage of 'A' is being collected in the expression in the table.

TABLE III. OPERATORS WITH THEIR NON-MASKING STATES

Operator	Expression	Non Masking State
NOT	NOT A	N/A
OR	A OR B	B = 0
	B OR A	B = 0
NOR	A NOR B	B = 0
	B NOR A	B = 0
AND	A AND B	B = 1
	B AND A	B = 1
NAND	A NAND B	B = 1
	B NAND A	B = 1
XOR	A XOR B	B = 0, B = 1
	B XOR A	B = 0, B = 1
XNOR	A XNOR B	B = 0, B = 1
	B XNOR A	B = 0, B = 1
TERNARY	(COND)? A:B	COND = 1, B is non-masking
	(COND)? B:A	COND = 0, B is non-masking

C. Algorithm

Start by breaking the expression into basic expressions. Once a basic expression has been identified, only work on the basic expression, and don't care about the actual complex expression (divide-and-conquer).

At the time of coverage collection, make sure the input being covered has taken both '0' and '1' states when the other input of the expression is in a non-masking state. Note that we're only looking at the value of the other input in the basic expression, and not inputs of the expression. A truth table lookup is not required. In fact, a truth table itself is not required to be created or stored at any time.

An input will be fully REC covered if it has taken both 0 and 1 values during simulation, in a state where the other input in its basic expression has a non-masking value. In case of XOR, the value of the other input of the expression must be the same in both collections. This ensures that both collections are made when the input is active in the same mode (inverting or non-inverting). An expression is considered fully REC covered when all the inputs of the expression have been independently covered.

An interesting implementation aspect of the REC algorithm is that it can be woven directly into the machine code generated by the compiler. Since there are no alternative evaluation mechanisms such as truth tables, fidelity with the primary evaluation engine is a non-issue. Furthermore, performance is optimal when only light additions of machine code are required to observe and collect coverage.

D. Non-Short-Circuit Operators

Since expressions are always evaluated left-to-right, short-circuiting enables us to assume that if an input is being evaluated during expression evaluation, the LHS of the input is already in a non-masking state. It is the responsibility of the implementation to ensure that LHS of the input is in a non-masking state when an operator doesn't short-circuit.

REC can be made to work together with non-short-circuiting operators using the following algorithm:

- Initialize the LHS_MASKED flag to 0.
- Don't change this flag while evaluating a short-circuit operator.
- In case of a non-short-circuiting operator, set this flag to 1 while evaluating the right operand (expression tree of right operand) if the value of the left operand masks the value of the right operand. i.e. set the flag to 1 while evaluating 'b' if 'a' evaluates to '0' in 'a & b'.
- While collecting REC for an input make sure that in addition to the masking expression being in the non-masking state, LHS_MASKED flag is also 0.

The effect of masking due to non-short-circuiting will be propagated to all inputs in the sub-expression being masked from left-to-right.

E. Duplicate Inputs

Special handling is required when an expression contains one or more inputs that have more than one occurrence in the expression. There are four ways in which REC handles duplicate inputs.

1) Relaxed

When there are duplicate inputs in the expression, the input will be considered covered if any of its occurrences is covered. While considering an input 100% REC covered, we'll not enforce that both _1 and _0 hits occur for the same occurrence of the input. Mixing hits from different occurrences will be allowed. I.e. even if _0 is hit for the first occurrence of an input in the expression and _1 is hit for the second occurrence of that input in the expression, the input will be considered covered.

For example, consider the following expression:

$$(a \,\&\, b) \mid (c \,\&\, a) \qquad (9)$$

There are two occurrences of 'a' in the above expression. Let's call them a{1} and a{2}. Even if a_0 is hit for a{1}, and a_1 is hit for a{2}, as shown in Figure 2, we will consider input 'a' REC covered.

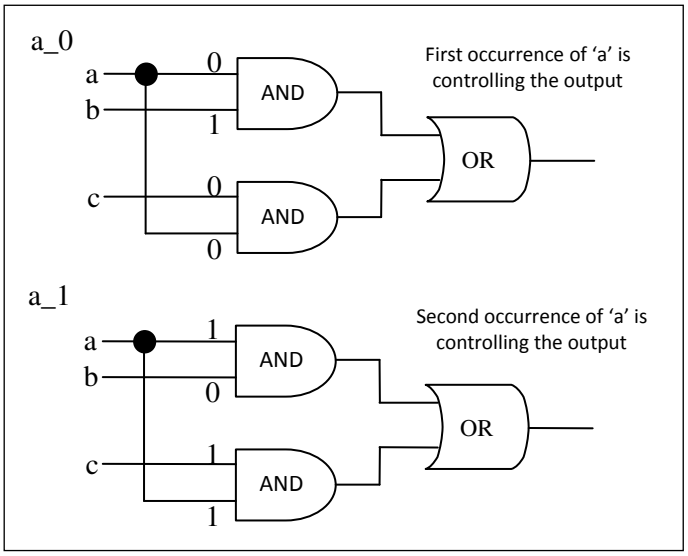

Figure 2. An Example of Relaxed Collection for expression "(9)"

We collect REC for a duplicate input in the same way as any other input. The masking expression needs to be in the non-masking state, and the LHS_MASKED flag needs to be taken into consideration for each input individually to make sure it is not masked due to non-short-circuit operators in the expression. There will be cases where the masking expression of an occurrence of an input terminal contains another occurrence of the same input terminal. There will only be one set of counters corresponding to the input terminal. All occurrences of the input will increment the same set of counters.

This mode of collection can help a verification engineer achieve 100% coverage faster, but it will not be as comprehensive as the other approaches listed below.

2) Strict

In the strict approach, we ensure that when REC is collected for an input, all its occurrences in the expression are simultaneously controlling the output of the expression. This is a stricter criterion than relaxed, and may not be suitable for all expressions. There may be cases when all occurrences of an input can never control the output of the expression simultaneously. It also makes coverage of the input more difficult as it reduces the set of input vectors that can cover it.

In Strict mode, REC for a duplicate input should be collected only when all the following conditions are met:

- None of the duplicate inputs is masked in the expression.
- The LHS_MASKED flag is taken into consideration for each input individually to make sure it is not masked due to non-short-circuit operators in the expression.
- The masking expression of the last occurrence of the input in the expression is in a non-masking state.

For example, consider the following expression:

a && (b && (c || (d && (a || (e && f))))) (10)

There are two occurrences of 'a' in the above expression. Let's call them a{1} and a{2}. There are no non-short-circuit operators, so LHS_MASKED flag will always be 0. We should collect coverage for 'a' if both a{1} and a{2} are reached during expression evaluation, and the masking expression for a{2} i.e. (e && f) is in its non-masking state of 0. This will happen when the value of 'a' is '1', 'b' is '1', 'c' is '0', 'd' is '1', 'e' is '1', and 'f' is '1'.

3) Balanced

When there are duplicate inputs in the expression, an input is considered covered if all its occurrences have been independently covered. While considering an occurrence of an input REC covered, the implementation enforces that both _1 and _0 hits occur for the same occurrence. Mixing hits from different occurrences is not allowed.

Each occurrence of an input in the expression should have its own set of counters, and coverage should be collected for each occurrence of an input independently, in the same way as any other input. The masking expression needs to be in the non-masking state, and the LHS_MASKED flag needs to be taken into consideration for each input individually to make sure it is not masked due to non-short-circuit operators in the expression. An occurrence of an input should increment only the counters associated with it. Each occurrence of an input is reported independently in the REC report, as if it was a distinct input.

In expression "(10)", a{1} will have '_0' and '_1' counters for it, and a{2} will have its own set of '_0' and '_1' counters. Input 'a' will be considered covered when both a{1} and a{2} have been independently covered. This will happen when all four counters have non-zero counts.

4) Relaxed Balanced

This approach combines the balanced and relaxed approaches to create a 'relaxed balanced' approach. Each occurrence of an input has its own set of counters. They are collected and maintained in the same way as balanced approach described above. However, an input will be considered REC covered when any one of its occurrences is covered. Unlike the balanced approach, it is not required to cover every occurrence of the input. On the other hand, unlike the relaxed approach, both the _0 and _1 hits need to come from the same occurrence of the input.

In expression "(10)", a{1} will have '_0' and '_1' counters for it, and a{2} will have its own set of '_0' and '_1' counters. Input 'a' will be considered covered when either a{1}, or a{2}, or both have been independently covered.

Figure 3 shows how the four methods compare with each other with respect to comprehensiveness of coverage and the ease of achieving 100% coverage. Method strict provides very comprehensive coverage, but requires a very precise set of input vectors, which makes it difficult to achieve 100% coverage. On the other hand, method relaxed may not provide as comprehensive coverage, but it makes it very easy to achieve 100% coverage. Method balanced also provides comprehensive coverage, but requires more effort to 100% cover the expression because each duplicate input must be covered independently.

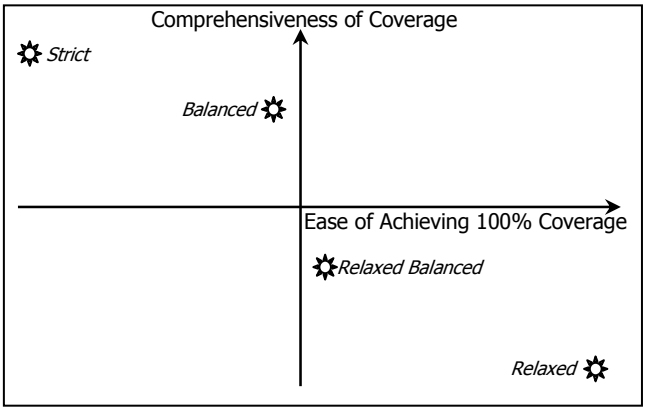

Figure 3. Comparison of Four Ways Of Handling Duplicate Inputs

F. Comprehensive Bi-modal Analysis

A bi-modal expression is considered covered if the target input controls the output to different values while remaining in the same bi-modal state (i.e. either inverting mode or non-inverting mode).

An expression can be bi-modal without the presence the XOR operator. To do a comprehensive analysis of such expressions, the output value of the expression should be taken into consideration while collecting REC. An input should be considered REC covered only when it has been observed in both '0' and '1' states during simulation with both these states resulting in different output values of the expression. This condition on the output should be in addition to all other conditions required for REC collection.

```
# ----------------Rapid Expression View----------------
# Line       23 Item    1 assign DO = (!IT1 && IT2) || (IT1 && IT4);
# Expression totals: 1 of 3 input terms covered = 33.33%
#
#   Input Term    Covered     Hint
#   ----------    ----------  ------------------------------
#            IT1           Y
#            IT2           N    Hit '_0' and '_1' for different outputs
#            IT4           N    Hit '_0' and '_1' for different outputs
#
#    Rows:  Hits(0)  Hits(1)  Target    Masking Expression
# ----------  -------  --------- ---------  ---------------------------------------------
#  Row   1:       1         0   IT1_0    {(IT2=1) && (IT1 && IT4)=0} OR (IT4=1)
#  Row   2:       0         1   IT1_1    {(IT2=1) && (IT1 && IT4)=0} OR (IT4=1)
#  Row   3:       0         0   IT2_0    (IT1 && IT4)=0
#  Row   4:       0         0   IT2_1    (IT1 && IT4)=0
#  Row   5:       0         0   IT4_0    IT1=1
#  Row   6:       0         0   IT4_1    IT1=1
```

Figure 4. Sample Coverage Report with non-XOR Bi-Modal Expression

Either a deeper analysis of the expression can be done to determine the modes of all inputs, or the mode can be automatically switched to bi-modal REC after identifying certain characteristics of the bi-modal expression, like the presence of XOR, or duplicate terminals working in different modes, etc. If a uni-modal expression is falsely classified as bi-modal, even though the coverage percentages will remain unchanged, bi-modal reports are more difficult to understand than uni-modal reports, and need more processing.

IV. PERFORMANCE BENCHMARKS

We created our own small tests to do some benchmarking, and found that as expected after converting an exponential algorithm to a linear algorithm, the performance gain increases significantly as the size of the expression increases.

In real world tests, an interesting effect tends to occur when the implementation shifts to a non-truth-table-based algorithm like REC. As a result of REC's ability to handle any expression, substantially more expressions are coverable than with other approaches. In addition, the newly coverable expressions tend to be the largest and most complex in the design. As a result, REC's performance gains tend to be offset by the fact that it covers more expressions.

Here are some results of our benchmarks:

TABLE IV. RESULTS OF BENCHMARK ON SMALL TESTS

Expression	FEC Sim Time	REC Sim Time	Ratio
3-input AND	25.1536	21.2093	1.18x Faster
6-input XOR	41.5346	33.9541	1.22x Faster
7-input XOR	63.1399	29.4898	2.14x Faster
8-input XOR	113.843	27.1377	4.19x Faster
9-input XOR	243.015	32.458	7.49x Faster
11-input XOR	1215.68	37.1395	32.8x Faster

V. CONCLUSION

Existing expression coverage metrics are based on truth table, and this makes them exponential in nature. This is the main reason why they suffer from performance and capacity limitations. But metrics like MC/DC and FEC are a boon for verification engineers who can use them to cover their complex expressions using only 2xN input vectors. REC brings in the core principals of MC/DC that give it this advantage, and works on basic expressions rather than the truth table, to give it the power of FEC while being linear in complexity with respect to the number of inputs in the expression. This helps it achieve tremendous gains in performance and capacity, as we've seen in our benchmarks. We saw that there is a more than 3X slowdown due to FEC when handling an 11 input XOR expression in a testcase. This is because FEC generated a 2048 row truth table for that XOR. A truth table also puts a restriction on the size of the expression because it blows up quickly for large expressions. REC is linear in complexity, and doesn't require a truth table. This helps it handle any arbitrarily large expression for expression coverage.

REFERENCES

[1] DO-178B, "Software Considerations in Airborne Systems and Equipment Certification", RCTA, December 1992, pp.31, 74.

[2] Gaurav-Kumar Verma and Doug Warmke,2013, "Rapid Expression Coverage", United States Patent Application

[3] Kelly J. Hayhurst, Dan S. Veerhusen, et. al., 2001, "A Practical Tutorial on Modified Condition/Decision Coverage", NASA/TM-2001-210876