Expression Coverability Analysis: Improving code coverage with model checking

Graeme D. Cunningham, Institute for System Level Integration, Livingston, Scotland
Paul B. Jackson, Institute for System Level Integration and University of Edinburgh, Scotland
Julian A. B. Dines, Motorola, Livingston, Scotland

Abstract

Code coverage analysis provides metrics to quantify the degree of verification completeness. It also allows the designer to identify possible bugs or redundant code thus speeding verification.

Many verification engineers report that the most time consuming area of code coverage analysis is the identification and documentation of intrinsically uncoverable expression cases. With manual inspection of the code being especially time consuming and error prone, automatic methods of identifying uncoverable expression cases are highly desirable.

Our work extends the model-checking-based coverability analysis work at IBM [5] to support analysing the coverability of expression cases.

We present results of applying our implementation on industrial scale designs provided by Motorola. We also analyse how coding style can impact the coverability of expressions and suggest how expression coverability analysis can be applied within the verification flow.

Introduction

Code coverage analysis measures how thoroughly a simulation testbench exercises parts of a design. It provides an indication of the quality of the testbench: if some part of the design is not exercised by a testbench, then perhaps some test case is missing from the test specification which was used to create the testbench. Alternatively, perhaps it is expected that the part is unexercised because the design is more general than strictly needed and some mode is unused. A further possibility is that the part is unexercised because of a bug in the design.

Because code coverage can highlight missing test cases and possible bugs, it is incorporated into many verification methodologies. For example, Motorola has an internal Semiconductor Reuse Standard (SRS) which all designs must conform to and which requires that specific levels of code coverage be attained[9].

There are various kinds of code coverage including statement coverage, path coverage, FSM coverage and expression coverage. Of these, expression coverage is the most fine grain and thorough. We have been particularly interested in expression coverage because of the difficulty in obtaining the level required in Motorola's SRS; specifically that 100% *explained coverage* be attained. Explained coverage means that each uncovered case must be explicitly explained. Many explanations are required because it is relatively common that it is logically impossible to cover certain expression cases. This arises because parts of expressions are frequently not logically independent of each other. Determining whether an uncovered expression case is intrinsically uncoverable, is only uncovered because of a missing test case, or is uncovered because of a bug is a tedious and error prone activity. Projects within Motorola have shown that as many as 20% of expression coverage cases are logically uncoverable and that explaining these cases is the most time consuming aspect of code coverage analysis.

The main contribution of the work described in this paper is to show how a formal verification technique (model checking) can automatically determine whether each expression coverage case for a design is intrinsically uncoverable or not. This speeds the production of explanations of when a case is intrinsically uncoverable. Also, when an expression case is coverable, model checking can suggest a simulation scenario that achieves this case, thus helping the testbench writer find missing test cases. An additional major advantage of our approach is that it can be deployed early in a design cycle before much of a simulation testbench has been produced. This permits early detection of uncoverable cases, early bug identification, and opens up opportunities to assist the testbench writer at all stages of producing a testbench.

Our work builds on work at IBM on coverability analysis that looked at statement coverability (essentially dead-code detection) and checking if variables attain all possible values[5,6]. Our work is distinguished by its engagement with issues raised in expression coverage (such as the frequency of uncoverable cases) that are not as common with these simpler kinds of coverage.

The paper is structured as follows. We first give an introduction to expression coverage, and describe how both intrinsic design features and design bugs lead to 100% coverage being unattainable. Next we highlight the

changes to verification methodology and consequent benefits of using expression coverability analysis. We move on to an exposition of how expression coverability works, present our prototype implementation of it and discuss our experimental results on industry standard designs provided by Motorola.

Background to Expression Coverage

Expression coverage (also called condition, condition-decision or multiple condition coverage [7]) provides coverage statistics for logical expressions. For each expression a set of cases is 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 fully covered when all of the individual expression coverage cases are exercised.

The cases of an expression are usually described by an expression coverage table, where each row of the table specifies values for sub-expressions. For example the Verilog statement

```
assign a = x && y && z;
```

would generate the following expression coverage table

x	y	z
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Expression tables can be generated from expressions found in logical assignment statements, either in continuous assignments or assignments in procedural code. Expressions with selected reduction and arithmetic operators also generate coverage tables, as do the conditional operator and the conditions of `if` statements.

Fully covering every combination of values for sub-expressions of every complex expression in a design usually requires impractically long simulation times. Code coverage tools allow for different modes of scoring expressions that require fewer test vectors but still provide confidence that the design has been thoroughly tested.

Change Scoring

Change scoring checks that each of the sub-expressions of the expression has been both a 1 and a 0 at some time during the simulation. If the expression contains a non-Boolean value it is scored like a scalar with any non-zero value being scored as a 1. This style of scoring reaches full coverage with the least stringent tests. It is also known as Basic Sub-Condition coverage.

Control Scoring

Control scoring only considers only those cases when the change in value of a single sub-expression can cause a change in value of the overall expression. The controlling cases for the logical operators && and || are different and can be seen in the example below.

&&	
0	1
1	0
1	1

||	
0	1
1	0
0	0

Multi-Level Expressions

An expression such as `(a && b && c)` is considered a ***first-level expression*** because it uses only uses one kind of operator. Expressions with more than one kind of operator are known as multi-level expressions. Expression coverage analysis decomposes a multilevel expression into a set of first-level expressions. For example, the expression

```
(a && b) || (d && e)
```

would create three separate expression coverage tables., two at the 2nd level for the sub expressions using the ***&&*** operator and one at the first level.

sub-exp1	
a && b	
0	1
1	0
1	1

sub-exp2	
d && e	
0	1
1	0
1	1

sub-exp1 || sub-exp2	
0	1
1	0
0	0

With complex expressions the precise details of how an expression is decomposed is tool dependent. The expression coverage tables actually generated will depend on the tool being used and options specified by the user, the scoring mode used or the level to which the analysis should be applied, for example. In addition some code coverage tools will examine other coverage metrics such as event coverage as part of expression coverage. This often means that comparing expression coverage analysis results of the same design but using different tools can yield different results.

Control Scoring is often referred to as Modified Condition/Decision Coverage (MC/DC) or as following a Focussed (or Directed) Expression Coverage methodology[4]. A more stringent style of expression scoring exists known as Vector Scoring**.** It is identical to change scoring except that there is individual bit scoring for vectors. A further scoring mode, SOP Scoring, represents the expression as a minimised Sum-of-Products. Control scoring is however the most commonly used of all scoring modes and in all examples we consider a control scoring methodology has been followed.

How Bugs can be identified by Uncoverable Expression Cases

As well a providing a measure of verification completeness, code coverage analysis can also be used in the identification of bugs. Consider the statement

```
assign a = mux[0] & mux [1] & mux [2];
```

If we assume that the sub-expressions mux[0], mux[1] and mux[2] have no dependency on each other, then obviously it is possible to fully cover all expression cases.

Now consider the following situation where the right-hand-side expression has been incorrectly coded.

```
assign a = mux[0] & mux[1] & mux[1];
```

The expression here is not fully coverable, as is shown in the following expression coverage table where the uncoverable cases are scored through.

mux[0]	mux [1]	mux [1] ;
~~1~~	~~1~~	~~0~~
~~1~~	~~0~~	~~1~~
0	1	1
1	1	1

The uncoverable cases draw attention to a possible bug.

Redundant logic doesn't always lead to functionally incorrect logic. For example, we might have the statement

```
assign a =  mux[0] & mux[1]&
            mux[2] & mux[1];
```

Only code coverage analysis (or perhaps a code review) could identify such a situation.

Other ways in which unintended redundancies can be introduced include the use of wrong operators and misplaced parentheses.

How Coding Style can lead to Uncoverable Expression Cases

Although uncoverable expression cases can indicate a bug in the design, they can also be caused by certain coding styles. Consider the following code fragment where we assume that `a`, `b` and `c` are coverable and that no other dependencies exist between them elsewhere in the design.

```
if (a && b && c) . . .
else
if (a && b && !c) . . .
else . . .
```

The expression coverage table for the second condition expression is as follows.

a	b	!c
~~1~~	~~1~~	~~0~~
1	0	1
0	1	1
1	1	1

We can see that the first expression case is uncoverable. Many of the uncoverable expression cases that we found in our investigation were of this form. It is possible to write the previous section of code using nested if statements.

```
if (a)
  if (b)
    if (c) . . .
    else . . .
. . .
```

Writing code in this style creates no uncoverable cases but often does not convey design intent as well as when the first style is used. If one uses the first coding style for n boolean variables and exhaustively enumerates every case, one ends up with $2^n - 1$ `if` condition expressions, and the total fraction of uncoverable cases for these expressions approaches 50%. We are therefore not surprised by reports of designs within Motorola having up to 20% of their expression cases uncoverable.

We have seen in the previous section examples of where redundancy within expressions can lead to uncoverable code. Sometimes redundancy is intentional. We have seen several examples where it was caused by dependencies between sub-expressions in different statements. For example:

```
assign hold_addr3 =
  addr_inc & siop_addr[3];
assign addr3_inc3 =
  addr_inc & sp_bwidth & !siop_addr[2];
assign addr3_inc64 =
  addr_inc & sp_bwidth & siop_addr[3];
assign inc_addr =
  hold_addr3 & addr3_inc3 & addr3_inc64;
```

In the fourth assignment, the use of `hold_addr3` is redundant because it is subsumed by `addr3_inc64`. This was not at all obvious because the fourth assignment was separated from the others by several hundred lines of code. This redundancy was caused by a late design decision. It wasn't considered a bug because a further design revision could have easily removed the dependency. The redundancy in the fourth assignment shows up in an uncoverable case of a control-scoring expression coverage table for this assignment.

hold_addr3	& addr3_inc	& addr3_inc64;
1	1	0
1	0	1
~~0~~	~~1~~	~~1~~
1	1	1

Benefits of Expression Coverability Analysis

Expression coverage is an extremely useful coverage metric because of its thoroughness. However it is unrealistic to aim for 100% coverage of all expression cases because some cases can be intrinsically uncoverable. Rather, the goal for the verification engineer is to write test vectors that cover all coverable cases and to explain each case that is intrinsically uncoverable.

To present how coverability analysis changes coverage methodology, we first run through an example expression coverage methodology, highlighting where the problems are.

Example of a Common Expression Coverage Methodology

The left-hand side of Figure 1 shows a simplified methodology.

Firstly at 1 a coverage simulation identifies which expression cases are covered by the current testbench. At 2 a verification engineer then makes a manual inspection of the code and attempts to classify the uncovered expression cases into the categories:

- **coverable by new test cases**
- **uncoverable due to bugs in the code**
- **intrinsically uncoverable**

These manual inspections are both time consuming and prone to human error. During the early stages of verification it is likely that there will be a large number of uncovered cases and the verification engineer will be unable to perform a full inspection of the code. Therefore the coverability of a number of expression cases will be unknown. This is shown in the diagram by the **Coverability Unknown** category.

At 3, any uncoverable cases due to bugs are corrected. Test cases are then created to cover more of the design and further coverage simulation is performed. We repeat further manual inspection, bug fixing and simulation until the coverage cases are either covered or classified as intrinsically uncoverable.

A big issue with this methodology is the difficulty and time consumed in the manual inspection step. In practice this step is delayed until late in the development of the design testbench when most test-cases have been implemented. This minimises the number of uncovered expressions that have to be considered, but means that any bug finding capabilities of coverage analysis are not available earlier in the verification process.

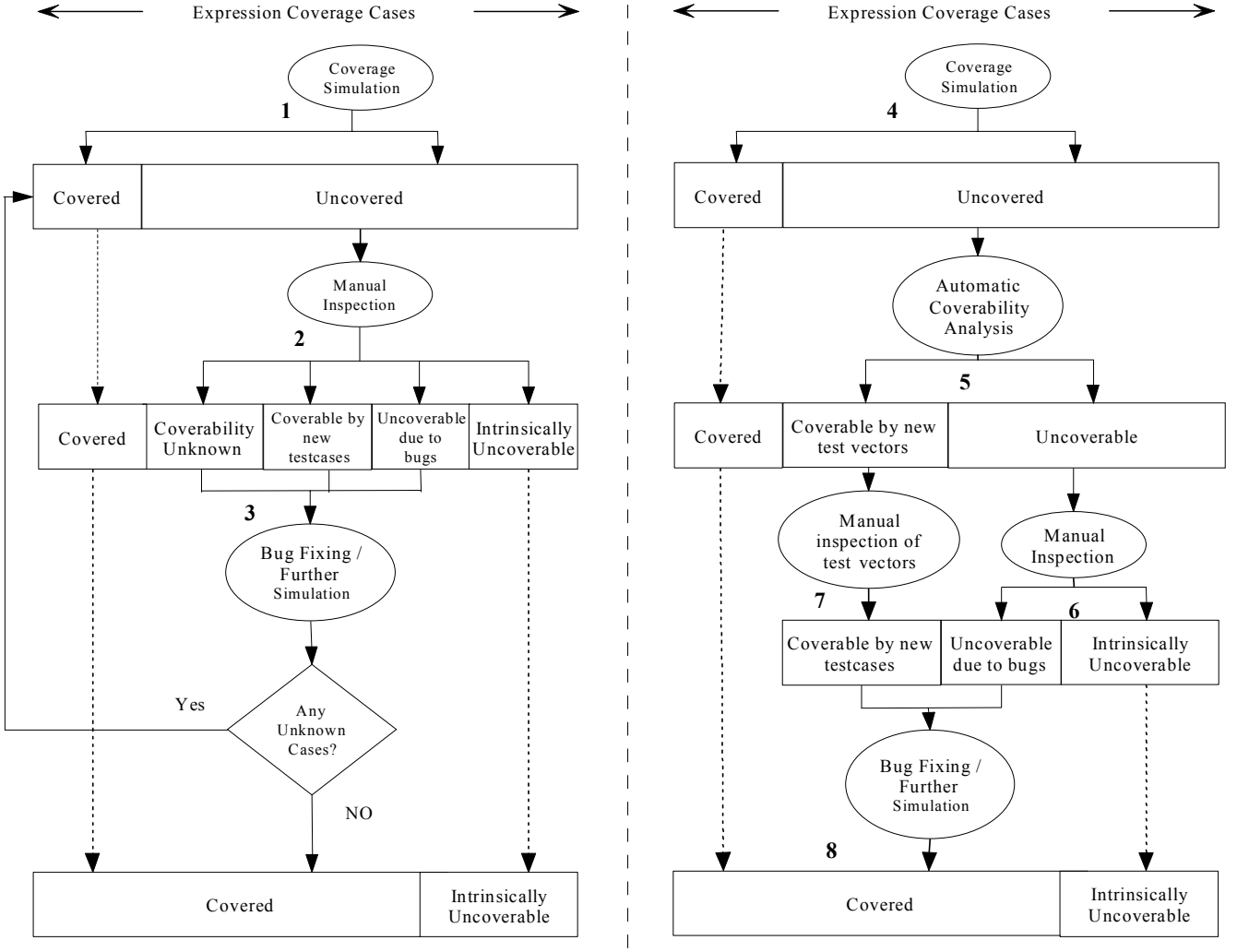

Figure 1. Coverability Analysis Methodology

Adding in Expression Coverability Analysis

On the right hand side of Figure 1 we show how coverability analysis might be added into a coverage methodology. Again, at 4, a coverage simulation produces a list of covered and uncovered expression cases. Now, at 5, expression coverability analysis automatically determines whether each uncovered expression case is coverable or not. For the uncoverable expression cases, manual inspection at 6 now only needs to determine if the uncoverable case is due a bug or is intrinsically uncoverable. For each coverable expression case, coverability analysis can suggest example test vectors to exercise that case. Generally, it would be unwise to simply augment the testbench with these test-vectors. Doing so does not address the observability of the case, nor does it highlight incompletenesses in the test plan or specification. Rather, using the test-vectors as a guide, one should go back to the design specification and the test cases derived from it and improve the test cases. This is shown in the diagram by step 7. This is an important methodological aspect of code coverage analysis in general.

We expect the time needed for manual inspection to be significantly reduced, and consequently there to be much less need to iterate coverage simulation and inspection. We show at step 8 in the diagram the ideal case of no iteration being needed.

A key feature of expression coverability analysis is that is does not depend on any test vectors or testbench. This means it could be used at any stage in the verification flow, and therefore we can exploit bug finding capabilities much earlier than with traditional expression coverage. Indeed it would be possible for a designer to use expression coverability analysis even before a design is complete.

Key Benefits

We summarise here the key benefits of expression coverability analysis.

- Automatically determines the coverability of a expression coverage case
- Automatically produces test vectors that can cover uncovered code
- Reduces required amount of manual code inspection and thus reduces verification time
- Can be applied at any stage in the design flow. Before a test plan, testbench or test vectors have been written.
- Immediately draws designer / verification engineer attention to possible bugs

Introduction to Model Checking

Model checking considers synchronous designs as Mealy state machines, where current state and input value information is used to calculate current output values and the state on the next active clock edge [3]. It provides automatic techniques for checking if user-supplied properties hold for such designs. Properties are commonly specified in temporal logics such as Linear Temporal Logic (LTL), Computational Tree Logic (CTL) or the Accellera Property Specification Language (PSL) standard [8]. For our purposes it is sufficient if one can check simple invariant properties (properties true at all states of all execution paths of the design) and such properties are expressible in any of the previously mentioned logics. Unlike simulation, model checking is exhaustive; it considers all sequences of input values. More generally, when not all input value sequences are valid, model checking can be constrained to consider just those valid sequences.

Work in Coverability Analysis by IBM

Our work extends work at IBM Haifa labs on using model checking for automatic coverability analysis [5]. They focused on checking statement coverability and determining whether variables attain all possible values.

For checking coverability of a given statement they used the following procedure.

1. Add a new auxiliary variable `V` to the design and have it initialized to 0.
2. Replace the statement with the assignment statement
   ```
   V = 1;
   ```
3. Use a model checker to check whether the modified design has the CTL property `EF(V == 1)`.

This CTL property is true just when there exists some sequence of input values to the design that drives it to a state in which V has value 1. The change in functionality here because of the code replacement is not important, because the model checking completes as soon as the replaced statement is visited for the first time. The method used to determine if a variable can attain all possible values is even simpler: no instrumentation is necessary, and one checks the property EF(V == k) to establish whether variable V attains the value k.

IBM created a tool that automates the design instrumentation and makes use of their RuleBase model checker. In further work, they explored several enhancements to make the model checking more efficient [6].

Calculating Expression Coverability

Expression coverability analysis extends the statement coverability and variable attainability work discussed previously. We first introduce the new instrumentation and the process required for performing coverability analysis. Then we discuss our implementation.

As with conventional expression coverage analysis, complex expressions are broken down into sub-expressions and analysed separately. We discussed earlier that for expression coverage analysis, an expression generates an expression coverage table with each line of the table representing a different expression coverage case. For expression coverability analysis we use model checking to determine whether each expression coverage case can be covered.

The following process is used to determine the coverability of expression cases generated by procedural code. For each case ***i*** of a coverage table ***j*** an auxiliary variable ***Vj.i*** initialised to 0 is added to the design. Immediately before the execution of the expression being considered, a statement is inserted that assigns the auxiliary variable to 1 when the sub-expression values specified by the expression case are achieved. We then call a model checker to check whether the modified design has the CTL property `EF(Vj.i==1).` As with statement coverability, this property will be true just when the corresponding expression coverage case is coverable. For example consider the procedural statement

```
z = (x || y);
```

Assuming we use control style scoring, we have the following corresponding expression coverage table

x	||	y
1		0
0		1
0		0

To perform expression coverability analysis the following instrumentation would be added just before the above statement:

```
e1_1 =  x && ~y;
e1_2 = ~x &&  y;
e1_3 = ~x && ~y;
```

The following properties would be generated and presented to the model checker.

```
EF(e1_1 == 1)
EF(e1_2 == 1)
EF(e1_3 == 1)
```

If we assume that line 2 of the coverage table is uncoverable then the results returned from model checking might look at follows:

```
# MC: formula passed   --- EF(e1_1==1)
# MC: formula failed   --- EF(e1_2==1)
# MC: formula passed   --- EF(e1_3==1)
```

For continuous assignment statements a slightly different method is used. The instrumentation generated is added in the form of a monitor created from a Verilog *always* block. Consider the following assignment statement

```
assign z = (x || y);
```

The expression coverage table and the properties checked are identical to those in the previous example, but the instrumentation generated would be as follows.

```
always @(x or y)
begin
  e1_1 =  x && ~y;
  e1_2 = ~x &&  y;
  e1_3 = ~x && ~y;
end
```

Implementation Details

Traditional code coverage analysis follows three key stages; Instrumentation, Simulation and Reporting. During the instrumentation stage modifications are made to the code to allow the different coverage metrics to be measured. The changes to the code should not affect the functionality of the code. Coverage is recorded in the form of coverage files that are then used to generate reports. The generated reports can give summary results about the design or detail exactly which statements, expressions etc. are uncovered.

We constructed a prototype implementation to analyse the effectiveness of expression coverability analysis. It would have been possible to create a parser that could analyse the code and generate the coverability instrumentation. However, to minimise our work, we generated our coverability instrumentation by keying off the source level coverage instrumentation generated by a commercial code coverage tool, specifically Summit Design's HDLScore[10]. Experience has shown that different code coverage tools identify expression in slightly ways often making comparisons impossible. For this reason we feel it is vital that coverability analysis should be linked to the code coverage tool of choice. Linking has obvious other benefits. For example, intrinsically uncoverable cases identified by coverability analysis can be removed from explicit further consideration by the coverage tool.

We show on the left-hand side of Figure 2 a tool flow for doing code coverage analysis using HDLScore and on the right-hand side how we fitted expression coverability analysis into this flow.

Our coverability instrumenter is a series of Perl scripts which take the normally instrumented HDL code from HDLScore, add the coverability instrumentation, and generate the corresponding properties for coverability checking.

Initially we model checked using VIS, an academic model checker developed jointly by University of California at Berkeley and the University of Colorado[1]. This was simple to interface with, but was unable to handle larger designs and also did not support non-blocking assignments. Later implementations used RuleBase, a commercial model checker from IBM[2]. RuleBase

performs a number of reduction methods that allow larger designs to be verified in a shorter time.

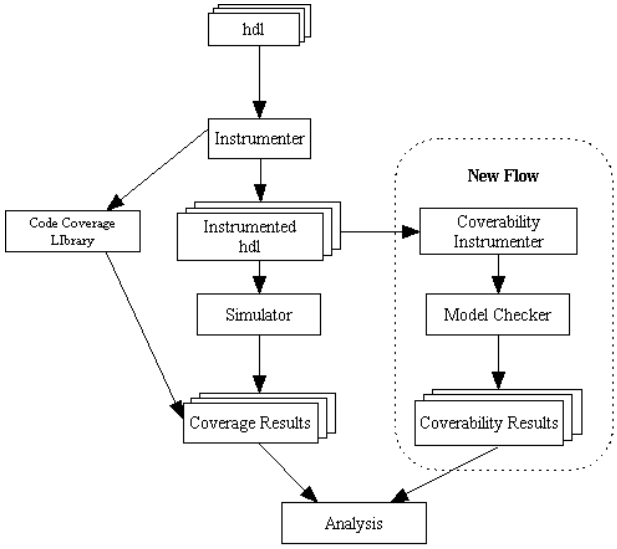

Figure 2. Our Coverability Tool Flow

Results and Discussion

We evaluated our expression coverability analysis on two commercial designs provided by Motorola.

	Design 1	Design 2
Lines of Code	63436	20932
Total Expression Cases	13702	3615
Intrinsically Uncoverable Cases	105	62

Design 1 was a 3M gate RapidIO™ network switch. We had access to this design at a relatively late stage of its verification when a large amount of manual inspection had already been performed. Our automatic expression coverability analysis identified 45 intrinsically uncoverable expression coverage cases that had been previously found by manual inspection of the code. In addition a further 60 intrinsically uncoverable cases were identified. Our analysis also identified an expression coverage case that had been wrongly classified by manual inspection as uncoverable.

Design 2 was a decoder module of a reconfigurable DSP for use in 3G base-stations. The verification of this design had been completed but we did not have access to any verification information. As well as identifying a number of complicated intrinsically uncoverable cases, our analysis also uncovered a bug where a signal had been incorrectly included into an expression..

The uncoverable expression cases identified in designs 1 and 2 represent 0.7% and 1.7% respectively of the total number of expression cases. These values are obviously design specific and are also related to the coding style employed, as discussed previously. Because of coding style, some designs at Motorola have as much as 20% of their expression cases uncoverable.

Practical experience showed that manually identifying dependencies between the sub-expressions in related `if-then-else` statements was relatively simple due to their close locality. Identifying the source of uncoverable expression cases due to dependencies between assignments, particularly in continuous code, was much more difficult. These can often take upwards of five minutes to identify. During our analysis we found one uncoverable expression case that required the analysis of over 20 related expressions through 4 levels of hierarchy.

Although our implementation only identified one bug in the two designs, it should be considered that the analysis was performed when the designs were at the integration stage. Bugs of this sort are common at the module level at the early stages of design.

The size of designs that can have coverability analysis applied is obviously limited by the size of designs that can be handled by the target model checker. To simplify and shorten the time taken to perform our analysis we applied the analysis on a module by module basis. This is sound methodologically. Expression cases found uncoverable at the module level are always going to be uncoverable at the chip level. There is however the chance that module level analysis will suggest how to cover an as yet uncovered case that actually is uncoverable at the chip level because of poorer controllability.

The majority of the modules could be analysed without any intervention from the user, although some of the larger modules required changing options that controlled the operation of the model checker. The time taken to determine the coverability of an expression table line varied from module to module but was generally in the region of 5 seconds per expression case.

The original IBM work on coverability analysis involved a separate run of their RuleBase model checker for each statement checked. This model checker like many others employs cone-of-influence reduction which on each run cuts down the design to just the parts required to check the property of interest. This dramatically improves the model checker performance.

For expression coverability analysis we initially had one model checking run per expression case. It's clear that the cone-of-influence reductions induced by each case of a given expression should be similar. We were interested in whether we would get better performance if we checked all cases of an expression together. We confirmed through experiment that the reductions are indeed similar and that

performance is significantly better when cases are grouped together. The table below illustrates this for two different modules.

Module 1	**Module 1**	**Module 2**
Flip Flops	55	246
Gates	4108	1512
Expressions/Exp.Cases	18/ 92	36/188
Cases Run Individually		
Total Run Time	3120s	3840s
Average time/Exp. Case	33.9s	20s
Cases grouped by Exp.		
Total Run Time	491s	847s
Average time/Exp. Case	5.3s	4.5s
Speed Increase	535%	353%

Suggestions for Future Work

Although our implementation has been successfully used in the analysis of complex industry standard designs, further work would obviously be required before an efficient tool could be created. For example, we never explored the processing of coverability witnesses provided by model checking into full input vector sets, as has been done in [4].

It is now becoming increasing common to integrate third party IP into designs. Confidence as to the quality of the design may be under question and it may be necessary to verify the IP to some extent. This would very likely involve code coverage analysis. There may however be a situation where a complex IP block had been integrated even though much of its functionality will not be used. We believe that an extension of coverability analysis could be developed where code that is specific to certain mode of operation could automatically be excluded from code coverage analysis.

Conclusion

Our work has shown that, even with a simple implementation, expression coverability analysis can be performed on industrial-scale designs. It has the potential to significantly reduce the time needed to achieve recommended levels of expression coverage.

References

1. VIS Group (1996), "VIS: A system for Verification and Synthesis", in proceedings of *8th International Conference on Computer Aided Verification (CAV)*, pp428-432, Vol. 1102 Lecture Notes in Computer Science, Springer. July.
2. Ben-David, S., C. Eisner, D. Geist and Y. Wolfsthal (2003), "Model Checking at IBM", *Formal Methods in System Design* 22(2), pp 101-108 . Kluwer.
3. Clarke, E, O. Grumberg and D. Peled (1999), "Model Checking", MIT Press.
4. Hayhurst, K. J., D. S. Veerhusen, J. J. Chilenski and L. K. Rierson (2001), "A Practical Tutorial on Modified Condition/Decision Coverage", Technical Memorandum TM-2001-210876, NASA.
5. Ratsaby, G., B. Sterin and Y. Wolfsthal (2001), "Coverability Analysis Using Symbolic Model Checking", in proceedings of *11th IFIP WG 10.5 Advanced Research Working Conference on Correct hardware design and verification methods (CHARME)*, Vol. 2144 Lecture Notes in Computer Science, pp155-160. Springer.
6. Ratsaby, G., B. Sterin and S. Ur (2002), "Improvements in Coverability Analysis", in proceedings of *11th International Symposium of Formal Methods Europe (FME)*, Vol. 2391 Lecture Notes in Computer Science, pp41-56. Springer.
7. Dempster, D and M. Stuart (2002), "Verification Methodology Manual", 3rd Edition, Teamwork International.
8. Accellera (2003), "Property Specification Language Reference Manual", Version 1.01, April. http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf
9. Motorola (2003), "Semiconductor Reuse Standards". http://e-www.motorola.com/webapp/sps/site/prod_summary.jsp?code=SRSSTANDARDS
10. Summit Design (2003), "HDL Score Product Overview". http://www.summit-design.com/HDLScore.htm