1 Introduction

We present a methodology that uses a formal property checker to analyse coverage holes left by module-level simulation in order to achieve early coverage closure. Coverage [3] is used to assess completeness of simulation-based verification. Coverage metrics are either based on the code being verified, ie code coverage, or on a more abstract view of the functionality of the design, ie functional coverage. The most common code coverage metrics are statement and branch coverage, and 100% are normally required during verification.

Coverage targets need to be carefully defined and justified since it is rarely possible to exercise all statements or take all branches. This is for several reasons, such as code structure or configurability of a design. Thus, for example, it may only be possible to take 95% of the branches in the code for the configuration being verified, perhaps leaving several hundred branches uncovered. These several hundred branches have to be analysed to decide whether they can in fact be covered, so then how, and if not, justification given as to why not, potentially identifying a coding error.

There are two major challenges in practice. Firstly, analysis of the missing coverage can consume a large amount of valuable engineering time. Secondly, recording of code that cannot be covered is typically based on line number or pattern recognition, both of which are susceptible to changes to the code; indeed changes to the code may mean that code that previously could not be covered can now be covered. To avoid this, coverage closure is typically left to near the end of the project, after sufficient code stability is achieved. This makes schedules more difficult to predict and means that bugs in the design and test bench may only be found late, both increasing the likelihood of schedule slips.

Our methodology uses a formal property checker to overcome these two challenges by automating the analysis of coverage holes. To illustrate the fundamental idea, let us consider an uncovered branch. We observe that, if the condition is not satisfied then the branch will not be entered during simulation. Now, if it is possible to formally prove that this condition is never satisfied, then clearly this branch is not coverable and can be discounted wrt branch coverage. If, however, the proof fails, then a counter example is produced by the property checker. This counter example provides an indication (rather than proof) of whether the branch really can be covered and if so how to do it.

Some formal tools have dead code checks built in, eg [1, 2], and we compare our methodology with such built-in checks indicating why our method is both more efficient and effective. Finally we report the results of using this methodology on the development of the Infineon TriCore 1.6 microcontroller.

Although the methodology includes formal verification, the development of the properties does not require formal verification expertise beyond what is described in Section 2, The Methodology. In addition, interaction with the formal tool is encapsulated within the process thus automating away any direct interaction with this tool.

2 The Methodology

The core principle of this methodology is based on the notion of temporal induction. A behaviour of a design will vary with time as the state of the design changes. Temporal induction states that to show B is true for any state of the design it is sufficient to prove the following two properties:

1. B holds at reset and
2. B ⇒ next(B) holds

Intuitively temporal induction works as follows. The first property shows B holds at reset, then the second property shows that B holds at cycle reset+1, another application of the second property shows that B holds at cycle reset+2, and so on.

In particular, if there is a branch in the code such as

if X then ...

it can be shown that the branch is never entered by substituting B above with not(X), ie showing that

1. not(X) at reset and
2. not(X) ⇒ next(not(X))

both hold. These properties are simple in that they can be easily handled by any property checker in seconds and they can be derived from a coverage report via scripting.

The properties are conservative in that if they pass then the branch (and any statements within the branch) definitely cannot be covered, but if they fail it may or may not be possible to cover the branch.

The properties can be strengthened while still remaining simple by analysing code structure. Thus for priority coding of branches eg an else if,

if X then A
else if Y ...
it can be proven that the else if branch is never entered by substituting B above with not(X) and not(Y).

Similarly for nested branches:

```plaintext
if X then
  if Y then ...
```

it can be proven that the nested branch is never entered by substituting B above with X and not(Y).

With understanding of the HDL used, these rules can be easily generalised to branches with any level of priority coding or nesting to give the highest possible chance of proving a branch (and its corresponding statements) cannot be covered.

The methodology, applied to close branch coverage, starts from analysing coverage reports from simulation and then generates a set of properties for each uncovered branch. If these properties pass then the branch can be filtered from the coverage report. By using a naming convention, e.g. one that contains the line number of the code, this filtering can easily be scripted. Thus the full process of reading the coverage report, generating and proving the properties, and filtering code that cannot be covered becomes push button.

In practice, some code will not be covered because the test bench is not intended to cover it. This information can be encapsulated in the properties in the form of extra assumptions. For example, if an input is intentionally never driven high by the test bench, or if inputs never violate a bus protocol, then this can easily be added to the properties as extra assumptions. Assumptions can also specify values of internal signals, e.g. configuration registers. Of course, all of these extra assumptions should be checked as assertions during simulation to ensure that they correctly reflect test bench behaviour.

Going one step further, for someone with formal expertise, unreachable state information about the design can be proved and added as an assumption to all of these properties. In this way, it is possible to formally prove that all uncovered code cannot be covered, although the effort for this may be considered too great (unless required for e.g. safety accreditation).

3 Contributions

Firstly, because the entire process, once scripted, is fully automatic, the methodology saves considerable engineering effort. In addition, the scripts are re-usable between projects.

Secondly, this methodology can be applied early in the project since it does not rely on code stability. Code changes are automatically reflected in the generated properties. Engineering effort can be invested into adding assumptions to the properties as described above in order to increase the number of holding properties. Hence, code that is not covered but can be covered can be identified much earlier, allowing tests or constraints to be written, regressions improved and bugs found earlier avoiding significant late code changes.

A third advantage is that it has been formally shown that any code excluded from coverage in this manner cannot be covered. This contrasts with the standard approach, often based on informal arguments. In the context of safety-critical applications, or when re-verifying legacy code with new coverage metrics, this is particularly significant.

Formal tools with built-in dead code analysis provide the above advantages to some extent. However, there are some major benefits to using this methodology in terms of performance and effectiveness. The main reason that performance will be significantly better is that properties are only generated for uncovered code. This means that, even very early in a project, more than 90% of the code will not be considered which makes the difference between an overnight run and a run taking several weeks. A built-in solution will only be able to take advantage of the coverage information if the formal tool is integrated with the coverage tool and specifically designed to do this.

The fact that the user has control over the properties greatly improves the effectiveness of the proposed methodology compared to built-in tools. Thus, for example, including priority coding or nesting of branches, is only possible because the user is writing the scripts that extract the properties. This alone, in our experience, already identified a greater amount of unreachable code than a built-in solution. For the TriCore 1.6 microcontroller, this methodology found 93% of uncovered statements were indeed not coverable, while a built-in tool only found 55%. Also, the addition of extra assumptions to reflect test bench scope and input behaviour may or may not be available when using a built-in solution.

We have found that built-in solutions target a limited number of coverage metrics, often only statement and branch coverage. Our scripted solution can be tailored for any metric, including focused-expression coverage and even functional coverage, provided that the functional coverage is specified in a language understood by formal tools such as SVA or PSL.

4 Results

This methodology was applied at Infineon Technologies during the TriCore 1.6 microcontroller verification. Statement, branch and focused-expression coverage (FEC) metrics were considered, all in the context of branch prioritisation and nesting. For statement coverage, 331 of 41074 statements were not covered during simulation and of these, 309 were proved not coverable. For branch coverage, 353 of 12341 branches were not covered during simulation and of these, 334 were proved not coverable. For FEC, 1581 of 27230 FEC points were not covered during simulation and of these, 1080 were proved not coverable. Using this methodology meant that statement and branch coverage were achieved well before RTL tape-out, and the number of FEC points left for consideration could be prioritised. This directly led to the discovery of several bugs in the RTL, test bench and random constraints. These bugs may not have been found otherwise.

References


### Appendix

```plaintext
343 if ( ready && sel_mast )
344 begin
345 if ( burst <= 3 )
346 begin
347 case(Mast_num)
348 1003 Mast’h1 : master = 1;
349 0 Mast’h2 : master = 3;
350 5000 default : master = 0;
351 endcase
end
353 else if ( burst > 4 )
354 begin
355 case(Mast_num)
356 507 Mast’h0 : master = 2;
357 432 Mast’h3 : master = 4;
358 872 default : master = 0;
359 endcase
end
end
```

Fig. 1: Small fragment of a coverage report consisting of three columns. The first column refers to the line number in the RTL file. The second column indicates the number of times the statement has been executed during simulation. The third column refers to the actual RTL code. Note that for line number 349 there are zero hits, *i.e.* there is a coverage hole at this line and coverage closure will focus on this line.

<table>
<thead>
<tr>
<th>Label</th>
<th>Control Flow Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_path_343</td>
<td>ready &amp;&amp; sel_mast</td>
</tr>
<tr>
<td>branch_path_348</td>
<td>ready &amp;&amp; sel_mast &amp;&amp; burst &lt;= 3 &amp;&amp; Mast_num == 1</td>
</tr>
<tr>
<td>branch_path_349</td>
<td>ready &amp;&amp; sel_mast &amp;&amp; burst &lt;= 3 &amp;&amp; Mast_num == 2</td>
</tr>
<tr>
<td>branch_path_350</td>
<td>ready &amp;&amp; sel_mast &amp;&amp; burst &lt;= 3 &amp;&amp; Mast_num != 1 &amp;&amp; Mast_num != 2</td>
</tr>
<tr>
<td>branch_path_356</td>
<td>ready &amp;&amp; sel_mast &amp;&amp; burst &gt; 4 &amp;&amp; Mast_num == 0</td>
</tr>
<tr>
<td>branch_path_357</td>
<td>ready &amp;&amp; sel_mast &amp;&amp; burst &gt; 4 &amp;&amp; Mast_num == 3</td>
</tr>
<tr>
<td>branch_path_358</td>
<td>ready &amp;&amp; sel_mast &amp;&amp; burst &gt; 4 &amp;&amp; Mast_num != 0 &amp;&amp; Mast_num != 3</td>
</tr>
<tr>
<td>branch_path_n343</td>
<td>!( ready &amp;&amp; sel_mast )</td>
</tr>
</tbody>
</table>

Fig. 2: Control flow conditions extracted from the code fragment depicted in Figure 1. Note that the expression on the last line, the one labeled `branch_path_n343`, is generated from the implicit `else` branch that complements line 343.
macro branch_path_349
    ( ready && sel_mast && burst <=3 && Mast_num == 2 )
endmacro;

property branch_not_covered_349 =
    always ( !(branch_path_349) ) @ (posedge clk);
endproperty;
assert branch_not_covered_349;

Fig. 3: Translation of the control flow expression for line 349 in Figure 2 to the formal property. The property is defined in three steps, namely macro generation, claiming of property and asserting the property. The macro generation encapsulates the control flow condition in formal syntax and assigns a name to the macro. The property definition then refers to this macro and claims that the specified condition is not reachable, hence the negation. The final step is asserting the property so that it is executed by the formal property checker.

Fig. 4: Waveform of a counter example provided by the formal property checker in case the property fails. Observe that the signal Mast_num has assumed a value of 1 at time ‘t’. The formal tool inductively tries to prove that it holds the same value in the next time interval ‘t+1’. However, in this case the property fails and the property checker provides a counter example in form of a waveform. This waveform shows that the signal Mast_value can take a value of 2, which indicates that the code is reachable.