# Lies, Damned Lies, and Coverage

Mark Litterick
Verilab GmbH[1], Germany
Email: mark.litterick@verilab.com

***Abstract*- Functional coverage is a key metric for establishing the overall completeness of a verification process; however, empirical evidence suggests that coverage models are often inaccurate, misleading and incomplete. This paper proposes that such coverage defects are extremely common and since coverage analysis tends to focus on holes, or missing coverage, rather than the accuracy of what is already reported, then this represents a significant risk to the overall quality of the verification process. After a generic introduction to the problem the paper discusses practical examples and proposes pragmatic solutions for minimizing the risk and improving quality. Finally we demonstrate a novel application of the UCIS API to cross-reference different aspects of functional coverage in order to validate correctness of the model under some circumstances.**

## I. INTRODUCTION

Functional coverage is a key metric in establishing the overall completeness of a verification process and is especially relevant for modern constrained-random environments. Functional coverage models are manually developed against verification requirements and implemented using cover points and cover groups within the verification components using the chosen language. During the course of a project the evolving coverage score is then analyzed to determine completeness of the task and any holes in the coverage results are addressed by altering the stimulus.

The problem is that many teams do not rigorously analyze the accuracy of coverage hits in the model as part of the methodology. This paper addresses that false positive problem from a pragmatic view based on observations across many projects, in many companies, using various verification languages. This paper does not provide a silver bullet to solve all these problems; but rather raises awareness with a comprehensive discussion and practical examples, provides pragmatic methodology guidelines and discusses possible tool and methodology enhancements for the future - all of which should add up to improved quality and more honest functional coverage. The concepts presented are language independent, but any terminology used is taken from SystemVerilog and the Universal Verification Methodology (UVM) [1] in order to minimize confusion in the text.

## II. FUNCTIONAL COVERAGE

Functional coverage is essential for constrained random verification (in order to determine what stimulus was actually generated and checked) and beneficial also for directed testing (to validate if the tests achieved their intended goal) [2]. Functional coverage data from multiple runs, with different seeds, and multiple different tests is collated into a single database to give an overall measure of verification status. Although a key metric regarding the scope of stimulus and quality of the testbench, it is important to note that all aspects of functional behavior also need to be checked - something that often gets forgotten in the drive for coverage completion - coverage alone is not enough.

Functional coverage is manually specified to identify what the verification team thinks is important to the success of the project, and should not be confused with low-level code-coverage. Code coverage only identifies which lines of code in the Device-Under-Test (DUT) were activated by the verification environment, but does not identify if these lines of code were exercised correctly or at an appropriate time, and cannot identify missing code or features. Code coverage has the benefit of being automatically measured (although typically manually filtered). It is assumed that both code and functional coverage are used in the reader's environments.

This paper assumes a working knowledge of functional coverage implementation using high-level verification languages and a corresponding methodology such as UVM. The focus of this paper is on determining how accurate, or otherwise, the functional coverage implementation actually is.

[1] www.verilab.com

 

## III. THE TRUTH, THE WHOLE TRUTH, AND NOTHING BUT THE TRUTH...

The goal of the functional coverage model is to provide an accurate, complete and concise measure of what functionality has been stimulated (and checked) in the verification environment.

### *A. A Pack of Lies*

Observations based on empirical evidence from coverage analysis in a large number of different clients, projects, applications and languages suggest that the functional correctness and completeness of functional coverage is typically both incorrect and misleading. For the sake of argument, and in order to stir up interest in the subject matter, we could say that the overall coverage model and specific details of the implementation exhibit all the symptoms of *a pack of lies*.

That is a pretty bold statement, but it is justified by the further observation that coverage results seem to be taken, in general, at face value and the only aspects of functional coverage that raise any interest in the project observers seems to be the gaps or holes - specifically coverage items that are specified but not apparently observed. In other words, the coverage results are presented as fact and therefore any inaccuracies in these metrics may be interpreted as lies.

Indeed the effect is the same in the long run. If a functional coverage model does not stand up to rigorous cross examination at some stage in the project lifetime, you can be sure it will destroy the credibility of the verification environment and harm the reputation of the engineers in question. Neither of these is a good thing.

The types of lies occurring in a functional coverage model that compromise the ability to tell "the truth, the whole truth, and nothing but the truth..." can be categorized as follows:

- deception
- omission
- fabrication

Lies of *deception* are essentially straightforward errors in the implementation of the actual coverage groups - the results do not tell the truth about what was observed in the verification environment. For example, the range of an item might be incorrectly specified so that we never stimulate and observe the appropriate maximum or minimum values that would stress the design. Defects in this category are generally easy to detect and rectify when reviewing (your own, or another person's) code.

Lies of *omission* relate to all of the missing coverage - the coverage results do not tell the whole truth about what we should have considered. For example if a design has any internal storage (pipeline, buffers, FIFOs, etc.) then it is very unlikely that pure transaction-based coverage will be adequate and additional temporal relationships between events should also be considered. Missing coverage is harder to detect and requires a good understanding of the actual verification requirements appropriate to the current abstraction level for the project.

Lies of *fabrication* result in coverage being measured for events, conditions or data that did not occur in reality - the results imply that more interesting scenarios were stimulated than truly occurred. Covering the register content on a write to a register, rather than when the content is used by the design, is an example of fabrication. Lies in this category are probably the hardest to detect and are certainly the most devastating threat to coverage quality; the reviewer really needs to understand both the detailed operation of the verification environment and the application requirements to draw the right conclusions.

### *B. Non-Malicious Behavior*

We need to be clear up front that lies in the coverage model are not, in general, a result of malicious behavior. Errors, omissions and fabricated artifacts are not deliberately introduced to dupe other observers or the developers themselves.

Consider this: at any advanced stage in the development of a project it would be a relatively trivial job for a competent verification engineer to manipulate the code-base so that the coverage goes miraculously to 100% - malicious behavior but technically straightforward. We could remove hard-to-reach coverpoints, introduce extra sampling events, manipulate ranges to absorb corner cases, etc. Next, consider the premise that our observations suggest that bins are being ignored because they are missing, ranges are incorrectly scoped, coverage is being recorded at the wrong time or under inappropriate conditions. All of this means that even without malicious manipulation, you could still have the same result - a false (equally miraculous) 100% coverage that keeps everyone happy. Which is worse?

 

## IV. EXAMPLES

This section provides some real-life examples to illustrate the scope of the problem and raise awareness for the reader as to what sort of lies to look for. These examples are taken from different types of verification environments and are language independent. Whether a particular defect is classed as a *little white lie* or a *great big whopper* probably depends on the application, so it is left to the reader to draw that conclusion. This list is by no means comprehensive, but the examples shown are all quite common defects.

### *A. Transaction Coverage*

The field content for transactions and other class objects constitute a major part of the functional coverage model for a verification environment. This is the simplest kind of coverage, but can still have inaccuracies and defects resulting in false reports that mislead the user.

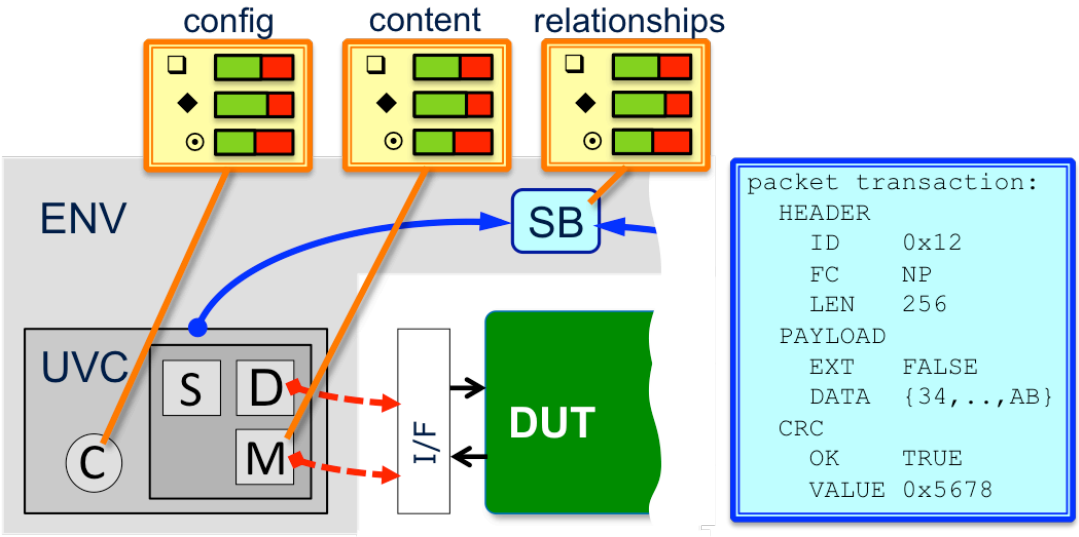


Figure 1.Transaction Coverage

#### *1) Field Ranges*

Correctly specifying ranges for the cover points in transaction fields is really the last line of defense against poorly scoped stimulus. If the bins for key values, such as minimum and maximum, are not separate from other ranges, then we can easily run full regressions with good coverage scores and miss the critical cases. We have observed this in many applications and it can be a killer problem. For example: a graphics application failed to stress the arc drawing algorithm with length or radius parameters of zero, but when the coverage was reviewed and the software team consulted these bins were included explicitly forcing the stimulus to be opened-up resulting in several key bugs being uncovered in the DUT.

#### *2) Conditional Fields*

Another common mistake in transaction coverage is to cover all transaction field values on the same sample event without additional filtering. Many applications have conditional fields that are only valid in specific transaction types (e.g. extension fields, or message parameters) and these should be qualified using conditional constructs in the cover statement (e.g. only count the value if some additional condition is valid) otherwise the coverage results are falsely positive.

#### *3) Configuration Objects*

It is misleading to cover configuration object fields when the configuration is set or changed since the actual value may or may not be used. Instead these field values should only be covered when a dependent operation occurs in the DUT (e.g. a transaction occurs on an interface, or a transformation algorithm is executed). This might require decomposition of the configuration object fields into more than one coverage group to allow for independent sampling events (e.g. separate transmit and receive events).

 

*4) Transaction Relationships*

For many applications comprehensive and pertinent transaction-based coverage is not enough and we also need to consider relationships between transactions both at the low-level interface level and also for high-level system scenarios. For example we need to cover the occurrence of legal packet reordering in PCIe networks, nested frames inside a DigRF stream or transaction ordering following reset or other important events. These coverage points are critical to application success but are often poorly implemented or omitted from the coverage model. This type of coverage is typically implemented in the scoreboards for the environment but is often quite hard to define and even harder to review for accuracy and completeness.

*5) Error Injection*

Functional coverage related to error injection (i.e. stimulus with deliberate protocol violations, e.g. CRC error) is often poorly represented in the coverage model results. Part of the reason for this might lie in the fact that passive observers like monitors cannot always distinguish between the specific causes of a current transaction when error injection is active (e.g. a coding error could be misinterpreted as a content error). This is one of the few cases where we would recommend supplementing the passive coverage model (if required) with some stimulus coverage from the driver responsible for doing the actual error injection based on sequence item flags passed down from the sequencer.

*6) Irrelevant Data*

At any particular DUT abstraction (e.g. block, sub-system or full-chip) capturing too much irrelevant data in the coverage database is misleading because it looks like lots of interesting stuff is happening and this might not be the case. If this data is not really relevant to the job in hand than this give falsely encouraging results and furthermore can hide the real missing interesting coverage goals - exaggeration, in effect, is also a kind of lie. For example, in a network-on-chip application very many types of packet can be transported by the bus fabric, however if the goal is to validate full functionality of a router subsystem, then only packet information relevant to the router should be covered (e.g. some of the header fields used for routing decisions, flow-class information, overall packet length, etc.), the router does not care about many other aspects of packet content (e.g. payload) and this coverage should be suppressed from the model.

We would recommend against the methodology whereby everything is sampled in the coverage base, to such a degree that the raw coverage is overwhelming, and then only a few aspects are picked-out for annotation to a verification plan for requirements tracking. In practice this approach gives poor results and it is hard to measure the absolute quality - a better approach is to keep the raw coverage lean and pertinent, and then to map the appropriate aspects of the coverage to relevant sections of the verification plan for tracking using forward or backward annotation.

*B. Temporal Coverage*

In this context *temporal coverage* is related to timing of events between different aspects of functional coverage and not just the accumulation of data with in a particular transaction associated with a single protocol interface.

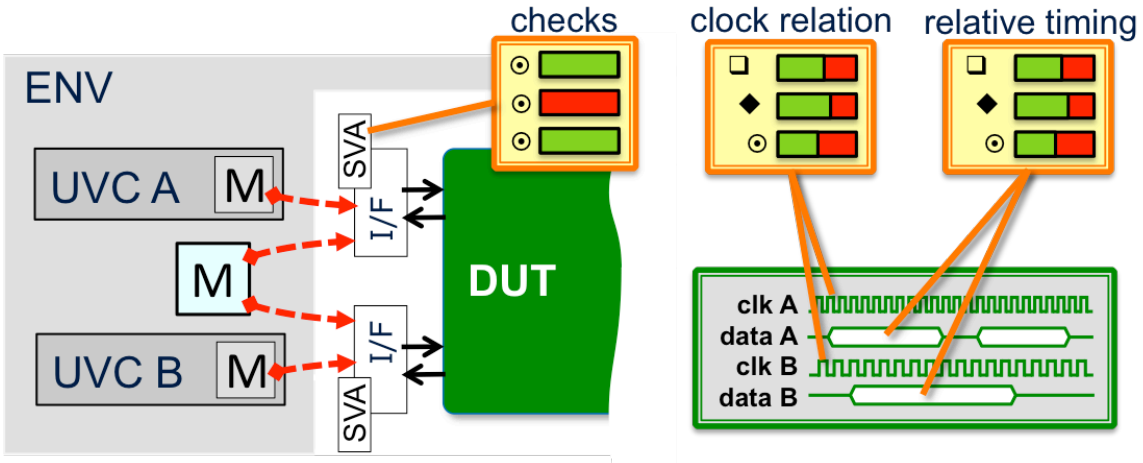


Figure 2. Temporal Coverage

 

*1) Clock Relationships*

Most modern designs have multiple clock domains that require special techniques to verify that the clock domain crossing (CDC) signals are properly synchronized. Even in the presence of fully validated CDC, or for derived synchronous clocks, it is also necessary to verify the operation of the DUT under all relevant clock conditions; but for some reason this coverage is often omitted from the coverage model.

For event-based simulation the absolute clock values are not important (although they might come into play for embedded models of analog blocks) but the clock relationships are. For example we want to measure if the source and target domains have the same, fast-to-slow, or slow-to-fast clock relationships. This can be further broken down with degrees of speed differences, for example asynchronous clocks probably care about less-than, equal-to, and greater-than 2x difference (due to synchronizers), and both asynchronous and derived clock domains care about less-than, equal-to, and greater-than Nx difference where N is the sequential depth of the RTL design (e.g. pipeline depth, routing delay, processing time, etc.). Clock relationships are essentially a configuration setting, and therefore should only be sampled when the actual clocks are used (e.g. when a transaction is sent or received between the clock domains), as opposed to just sampling observed relationships when nothing interesting is happening in the DUT.

*2) Reset Conditions*

Typically we wish to cover reset activation when the DUT is in different states (e.g. processing a transaction, idle, waking-up, going-to-sleep, etc.). It is very common to observe non-zero reset coverage even though only an initial reset has been applied - this is misleading since the DUT was not in any state when the initial reset was applied. It is better to exclude the initial reset event from this coverage and make sure only subsequent reset events are used.

*3) Temporal Relationships*

It is not uncommon to see an entire functional coverage model for an environment based exclusively on transaction-based coverage. In any design with more than one interface, or any internal storage stages (e.g. pipelines, buffers, FIFOs, synchronizers, etc.) this is very unlikely to cover all the appropriate verification concerns for such a DUT. We also need to cover important temporal relationships (e.g. relative transaction timing on separate interfaces, transaction timing relative to power-down requests, occurrence of flush requests relative to processing state, etc.). This coverage tends to be quite distributed and will often require dedicated cross-interface monitoring at the environment level to track relative state and timing events for the various interfaces - it is also hard to implement and review, but extremely valuable nonetheless.

*4) Covering Checks*

Functional coverage of all checks operational in the system is a requirement. Some checks, like assertions in the interfaces, are automatically covered as part of the unified coverage reports - but this is typically not enough. For accurate coverage we also need to know under what conditions the checks were successful and this typically requires extra information to be recorded to know we have covered all cases (e.g. an assertion that says “A must cause B or C” is a valid check, but needs two cover points to be defined to know that “A caused B” and “A caused C”).

Transaction content checks in a monitor and relationship checks in a scoreboard also need explicitly crafted coverage. For example if the environment is capable of dropping a packet with a certain class of error, then the scoreboard model must be aware of this to prevent mismatches, but we also need to functionally cover the occurrence of all such valid filtering conditions. For some reason these conditions are often omitted from the coverage model, even though it is critical behavior that we need to know was stimulated and checked. The concept is not so complicated; ask yourself “can you tell, from the coverage results, if every check that was deemed important actually happened, how often, and under what conditions?”

*5) Sub-Transaction Events*

Many protocols require additional coverage for intermediate events while the interface is trying to decode some traffic. Sometimes these important events do not even result in a transaction being published by the monitors (e.g. if two competing masters back-off during arbitration), and therefore need to be recorded by some other means. For example, in the I2C protocol we need to provide coverage, which identifies that, the DUT backed-off during different phases of an arbitration contest (when an external master wins), and we also need to identify if the external master was addressing the DUT at the time.

 

*C. Register Coverage*

Register models typically provide a minefield for the misrepresentation of coverage information related to DUT state and operations. In addition the sheer volume of data associated with the register details can swamp other aspects of coverage if not correctly encapsulated.

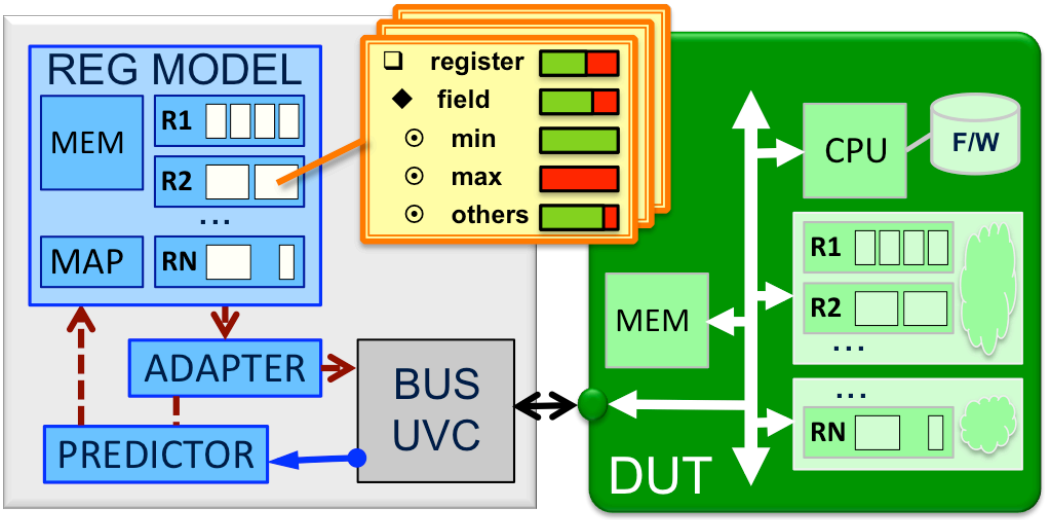


Figure 3. Register Coverage

Here are a few things to look out for when trying to determine register related coverage goals:

- did we use all relevant values and ranges in control and configuration?
- did we read all appropriate status responses from the DUT?
- did we validate all the reset values from the registers?
- did we access all register addresses?
- did we attempt all access types for each register?
- did we prove all appropriate access policies for the register fields?

*1) Register Writes*

It is extremely misleading to cover the field values for control and configuration registers when a write operation is performed. This can lead to very high coverage scores even in tests with little or no additional DUT functionality (for example we can write ten different values to a configuration register but use only one of them). Instead these field values should be covered when they are actually used by the DUT (e.g. to send a transaction, or execute an algorithm). In such cases the coverage implementation can be provided by the register model but triggered by a passive monitor when an appropriate event occurs in the environment.

*2) Register Reads*

Covering field values for status register reads is also misleading, since some of these results come from reset conditions and not DUT operations. It is more appropriate to record field values for a status register read only when the value has changed from its reset value, or from the previously read value. Meta-data can be used to track the previous state of the field value.

*3) Reset Values*

In order to validate that the reset values of the registers in the DUT match those in the model, we really need to cover read accesses to registers when no other operation has been performed on the register after reset. In real life this can be difficult to attain, especially for volatile fields (that are updated by RTL), and this coverage is often omitted from the model. If we add some meta-data to a register field to track the updates from bus operations or RTL (by extending the active-monitoring capability) we can more accurately cover reads of reset status (and separately check the values).

 

*4) Address Map*

In order to validate the implementation of address decoding and partitioning in the RTL hierarchy, we need to cover accesses to all register addresses in the design. The register content (field values) should be covered separately and backdoor accesses should be excluded from this coverage collection (since we really need to know if we can access the register in the DUT). Ideally the register access would only be covered when the effect is observed (i.e. the write has an observed effect on the DUT or a read value is checked) but typically this is not required if the field content is covered separately.

*5) Access Rights*

The access rights of a register in a particular address map can put additional restrictions on the types of operations allowed on the register (and therefore the enclosed fields). Both legal and illegal access types must be covered for each register in the address map range (e.g. we need to cover attempted writes to read-only registers). This coverage is often omitted and therefore the corresponding access rights and protection could avoid validation in the verification flow.

*6) Access Policies*

Irrespective of the overall access rights in the address map, each register field can have specific access policy applied (e.g. write-only, read-only, read-to-clear, write-one-to-clear, etc.). We need to ensure that all possible accesses have been attempted on these fields and where appropriate take into account the access data values. For example with a read-only register field, we need to cover both read and write attempts to the field, and the write must have a value that does not match the field content. Likewise for a write-one-to-clear field, we need to cover the attempted write of both one and zero.

Additional cross coverage may be required if the access policy is dynamic (e.g. in order to implement a hardware firewall). In which cases we need to cover both legal and illegal attempts, with corresponding data if appropriate, in each of the available modes.

## V. LIE DETECTORS

Unfortunately there are no silver bullets for validating the correctness and completeness of the functional coverage model in the context of the different verification abstraction levels; this represents a significant risk considering the importance of the coverage results in the verification flow. This section provides some pragmatic guidance for improving the methodology including suggestions on how to prepare the coverage model and how go about analyzing coverage accuracy of implemented code. In addition we look at a novel approach that was tried out to automate some aspects of this and propose how this might be extended in the future as tool developers turn their attention to this significant problem.

### *A. Reviews*

*1) Coverage Architecture and Plan Review*

During verification architecture development, prior to the implementation of the coverage code, it is necessary to plan the coverage model with enough detail that a skilled reviewer can identify inaccuracies and omissions. A full discussion on the process and requirements for functional coverage planning is outside the scope of this paper but is discussed in [2] and [3]; the key thing to consider in this context is that the plan needs to be reviewed in detail since this is the best chance of detecting missing or poorly scoped coverage points. For each component of the verification environment we should create a coverage table that details:

- what is covered (items and ranges)
- when the sampling event occurs (temporal event)
- under what conditions the sampling is allowed (logical condition)

All aspects of coverage should be considered including transaction fields, configuration settings, control, status, temporal relationships, checks, etc. The goal here is to be concise and complete, but not get bogged down in code syntax. Prior to implementation interested parties should review the coverage plan in order to detect what is missing, irrelevant or incorrect. The conditional and temporal aspects of this analysis are extremely important - these are often omitted by less experienced teams, resulting in a coverage plan that is very transaction content focused and inadequate for most applications. Leaving the details of the conditional and temporal aspects to the implementation stage is not recommended, since it is much harder to review and more error prone.

 

*2) Coverage Implementation Review*

Once the verification environment is at a mature stage of development it is necessary to perform a review of the functional coverage code. In the context of this paper it is especially important to review the implementation against the planned coverage; looking for inaccuracies, missing items and false positives results from bad sampling, grouping or triggering of the coverage. In particular:

- are all planned items implemented in the actual code?
- are non-implemented items clearly documented in the code-base?
- are the correct ranges used for each coverpoint, with key values correctly isolated?
- are the coverpoints only recorded under appropriate logical conditions?
- is the covergroup sampled at the right time, when the contents are used by the DUT?
- is cross-coverage correctly scoped, relevant and achievable?
- has reused coverage, from other VIP, been correctly tuned for the current application?
- do all checks have corresponding coverage items to identify all pass conditions?

Other aspects of the coverage implementation review include coding style, encapsulation and control. It is important that coverage is encapsulated correctly in isolated classes that allow substitution by factory overloads otherwise users cannot easily replace the coverage when required. Likewise it is important to validate that coverage controls (e.g. *has_cover*) remove the actual coverage from the model rather than just inhibiting the sampling of the *covergroup* (resulting in dead coverage code). It is also worth noting that functional coverage exhibits the somewhat unique characteristic that mistakes are much more likely to go undetected than is the case with stimulus or checks - if you make a mistake in stimulus or checks, then there is a good chance you will kill the regression, if you mess up coverage there is no comeback - hence this implementation review is very important.

This coverage review needs to be performed when the codebase is implemented, but not right at the end of the project since the goal is to identify and repair defects with a goal to improving the quality, not just providing an audit of the status.

*B. Coverage Analysis*

Normally coverage analysis is focused on the holes, or missing coverage scores, reported by the coverage database tools. In the context of this paper, the focus is on analyzing the accuracy of the actual reported coverage hits in order to determine correctness of the model - it is assumed that coverage closure will also be a goal for project completion.

It is not enough to review the coverage implementation and results in isolation from the actual behavior of the environment. At various stages during the development of the project the coverage should be analyzed against observed traffic for different test scenarios. The reviewer can select a few specific tests - not just the simplest test, but tests that exercise an appropriate spread of features - and validate that:

- all reported coverage is exactly what happened in the test
- no additional coverage is reported for events that did not happen
- all interesting stimulus conditions are appropriately recorded by observed coverage
- all relevant checks have corresponding coverage and no extra or missing checks reported

In other words the analysis is looking for lies of deception, omission and fabrication in the reported results against the actual test scenario. This is of course not that straightforward, and as mentioned earlier typically requires a deep understanding of both the application and testbench environment implementation. For that reason the analysis may well be performed by the technical lead, but it presents a great opportunity for peer programming and mentoring of other team members.

As an example, consider the packet switch shown in Figure 4. For a particular test scenario seed we send 10 different packets into the DUT, one of which has a CRC error injected to invalidate the packet.

 

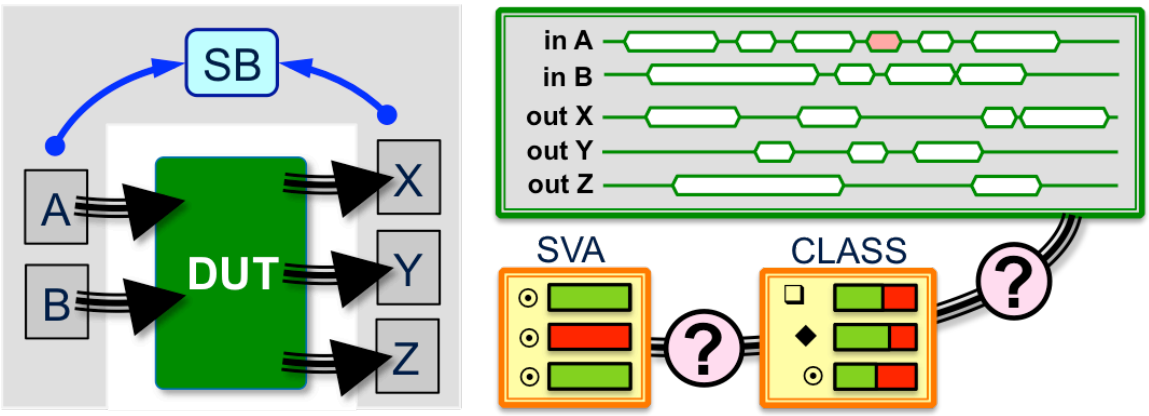


Figure 4. Coverage Analysis and Cross-Referencing

It is important to cross-reference all aspects of verification environment operation in the analysis of the detailed behavior. It is not just the successful hit of particular coverage points that we are concerned with in this analysis, but the absolute score for each and every bin - including the distribution of type and instance-based coverage results. For the current example we should cross-reference coverage with the following:

Log-files tell us what transactions were observed at each interface:

- are all transaction fields correctly reported in coverage?
- what about derived values like packet length and meta-data like delays?
- is there too much data recorded for non-relevant fields (like payload content in this case)?
- is the error correctly reported in transaction coverage?

Waves illustrate the relative timing of the transactions on the various interfaces:

- are the relationships of traffic on the interfaces (e.g. overlaps) recorded in coverage?
- are there any significant transaction ordering events to report (e.g. overtaking)?
- are the delays correctly measured and reported?
- does the coverage reflect how many packets are in flight (inside DUT)?
- does the coverage reflect how many slices are being processed at a particular time?
- are the observed clock relationships correctly reported by coverage?

Checks are performed by assertions (protocol), monitors (content) and scoreboards (relationships):

- which checks were performed and are they all reflected in coverage?
- do the observed assertion scores match the scoreboard and transaction coverage?
- does coverage reflect that the scoreboard model also filtered-out a packet?
- is the effect of error detection included in register or protocol signal coverage?

Such a thorough analysis of detailed coverage scores against observed scenarios tends to quickly find some major defects in the coverage model including omissions - often we can confirm the stimulus and checks are valid, but cannot honestly say the occurrence of the specific conditions is well reflected in the coverage. Successful conclusion of this analysis process also provides improved confidence in the overall verification environment and coverage accuracy in particular.

### *C. Automation*

With current technology there is very little scope for automatically deciding what types of functional coverage are appropriate for most high-level protocol scenarios. Of course, once decided, there is scope for automatically generating the coverage implementation and adapting it to DUT parameters, but that is not the focus for this paper.

A more interesting and relevant question in the context of this paper is “can we automate the validation of functional coverage correctness and completeness in a given coverage model?” Essentially this should be possible, to a degree, since all we need is a rule-based application of the same cross checks that were applied during the manual analysis stage mentioned previously. The author is not currently aware of any commercial tool that does this, but in order to demonstrate a proof of concept we experimented with an ad-hoc approach to validating the coverage using the Unified Coverage Interoperability Standard (UCIS) [4] and PyUCIS [5]. UCIS is an open industry-standard that provides an Application-Programming Interface (API) to enable sharing of coverage data across

 

multiple tools (e.g. simulators, accelerators, formal, custom) and from different vendors. PyUCIS is a SWIG/Python wrapper that enables a simpler user experience than the raw C-code implementation. Other applications of UCIS include on-the-fly stimulus adaption, test grading, and assimilation of coverage reports from multiple sources [6].

*1) Cross-Referencing Checks and Coverage using UCIS*

The basic principle is that coverage results from SystemVerilog Assertions (SVA) in the interfaces and from different class-based functional coverage groups are available in the UCIS coverage database (UCISDB), therefore at least some aspects of coverage cross checking could be achieved using UCIS, as shown in Figure 5.

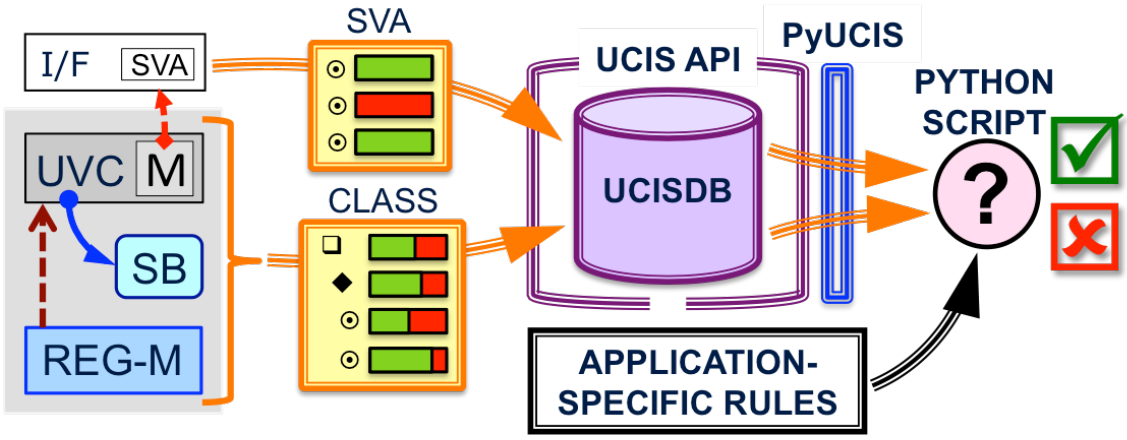


Figure 5. Validating Coverage Using UCIS

In order to experiment with this novel approach, we chose to look at an OCP verification environment that has configuration-aware assertions (enabled, or not, based on profile configuration) and class-based transaction coverage. Using the UCIS we can access and compare:

- assertion and class-based coverage scores
- scores for different assertions in an interface
- different aspects of class-based coverage

In theory there is a strong relationship between the assertion results and the observed functional coverage for the transactions (e.g. if we get N valid protocol assertions passing, then we should not have a score of more than N for any aspect of transaction coverage). Likewise different assertion scores can be compared (e.g. assertions for different phases of the protocol - request, data-handshake, response - should have appropriate coverage scores based on observed full transaction types). Transaction based coverage can also be compared between components and layers in the environment (e.g. the number of protocol errors reported in the transaction coverage can be compared with the checker coverage from the scoreboard).

The UCISDB stores hierarchical information (*scope*) and coverage counts (*coveritem*). In order to access the required bin count information we need to open the UCISDB, and search through the scopes iteratively until we match the required *coveritem* name, then we can extract the coverage count. Note, in the following code snippets:

```
from pyucis import *
import sys
import re

#  ucis_* methods are the actual UCIS API methods wrapped with SWIG into Python code.
#  pyucis_scope_itr is a Pythonic iterator build from the UCIS API methods:
#    - ucis_ScopeIterate, ucis_ScopeScan, ucis_FreeIterator
#  pyucis_cover_itr is a Pythonic iterator build from the UCIS API methods:
#    - ucis_CoverIterate, ucis_CoverScan, ucis_FreeIterator
```

We implemented additional python helper methods to find the UCIS *scope* from specified path description, provided as a string or compiled regular expression (*pyucis_find_scope*), and extract the coverage count from UCISDB for the corresponding *coveritem* (*pyucis_get_count, pyucis_get_cov_count*) as shown in the following code. These helper methods are also available via PyUCIS [5].

 

```
# Returns the first found PyUCISScope object which matches <path>
# <path> can be a string or a compiled regular expression object (re.compile("regex"))
def pyucis_find_scope(db, path, scope=None):
  result = None
  # If of encounter a coverpoint, we need to iterate over bins, else scopes
  if ((scope != None) and (ucis_GetScopeType(db, scope) == UCIS_COVERPOINT)):
    sh = PyUCISScope(db, scope)
    scopepath = sh.GetStringProperty(property=UCIS_STR_SCOPE_HIER_NAME)
    for ch in pyucis_cover_itr(db, scope):
      (status, name, data, source) = ucis_GetCoverData(db, scope, ch.coverindex)
      try:
        if (path.search(scopepath+"/"+name)):
          return ch
      except AttributeError:
        if ((scopepath+"/"+name) == path):
          return ch
  else:
    for sh in pyucis_scope_itr(db, scope):
      scopepath = sh.GetStringProperty(property=UCIS_STR_SCOPE_HIER_NAME)
      try:
        if (path.search(scopepath)):
          return sh
      except AttributeError:
        if (scopepath == path):
          return sh
      scopetype = ucis_GetScopeType(db, sh.handle)
      if (scopetype in [UCIS_INSTANCE, UCIS_PROGRAM, UCIS_PACKAGE, UCIS_INTERFACE,
                        UCIS_DU_MODULE, UCIS_DU_ARCH, UCIS_DU_PACKAGE,
                        UCIS_DU_PROGRAM, UCIS_DU_INTERFACE, UCIS_COVERGROUP,
                        UCIS_COVERINSTANCE, UCIS_COVERPOINT, UCIS_CLASS]):
        result = pyucis_find_scope(db, path, sh.handle)
        if (result != None):
          return result
  return result

# Returns the accumulated hit count
def pyucis_get_cov_count(db, pyucis_scope):
  total_count = 0
  scopetype   = ucis_GetScopeType(db, pyucis_scope.handle)
  if (scopetype in [UCIS_COVERPOINT, UCIS_CROSS, UCIS_COVER, UCIS_ASSERT]):
    if (pyucis_scope.coverindex != -1):
      (status, name, data, source) =
        ucis_GetCoverData(db, pyucis_scope.handle, pyucis_scope.coverindex)
      total_count += data.data.int64
    else:
      for ch in pyucis_cover_itr(db, pyucis_scope.handle):
        (status, name, data, source) =
          ucis_GetCoverData(db, pyucis_scope.handle, ch.coverindex)
        if (data.type in [UCIS_COVERBIN, UCIS_PASSBIN, UCIS_CVGBIN, UCIS_DEFAULTBIN]):
          total_count += data.data.int64
  elif (scopetype == UCIS_COVERGROUP):
    for sh in pyucis_scope_itr(db, pyucis_scope.handle):
      total_count += pyucis_get_cov_count(db, sh)
  else:
    raise PyUCISError("..." ...)
  return total_count

# Returns the accumulated hit count for items under <path>.
# <path> has to point to a group, item, cross, transition, assertion, cover or bin
def pyucis_get_count(db, path):
  pyucis_scope = pyucis_find_scope(db, path)
  if (pyucis_scope == None):
    print("WARNING: ...")
    return -1
  else:
    return pyucis_get_cov_count(db, pyucis_scope)
```

In our environment we execute a Python script as a post-processing step after single simulations or complete regressions. In order to access the required bin count information we need to open the UCISDB, for example:

 

```
db = ucis_Open("example.ucdb");
```

The application-specific rules associating the *coverpoints* are hard-coded into the user script. For example, in OCP we can check that the class based command coverage matches the number of passes for the command hold assertion (*MCmd* must remain stable for the duration of the request phase):

```
if (pyucis_get_count(db, "/tb_top/ocp_if/checker/assert_request_hold_MCmd")
 != pyucis_get_count(db, "/vlab_ocp_pkg/vlab_ocp_monitor/cg_req/cp_cmd"))
   print("ERROR: ...” )
```

For OCP many of the coverage checks are more complicated due to the profile configuration and also some assertions fire more than once per phase. For example the *burstlength* configuration field identifies whether *MBurstLength* is present; if it is, then the assertion checks it's value on every clock during the request phase. We can (only) check that the class-based coverage never reports a higher score for burst length values than the total assertion score if *burstlength* was ever active (*cp_burstlength* bin "1" was hit) in the profile configuration (otherwise the *cp_burst_length* coverage is omitted from the database):

```
if (pyucis_get_count(db, "/vlab_ocp_pkg/vlab_ocp_monitor/cg_cfg/cp_burstlength/1") > 0)
  if (pyucis_get_count(db, "/tb_top/ocp_if/checker/assert_request_value_MBurstLength_0")
    < pyucis_get_count(db, "/vlab_ocp_pkg/vlab_ocp_monitor/cg_req/cp_burst_length"))
    print("ERROR: ...”)
```

In this proof of concept stage, we identified some 40 rules (not all of which were implemented) for this OCP environment with 60 assertions and 5 *covergroups* containing 30 *coverpoints*. In the process of coding and validating the rules for associating the different coverage scores, and trying it out on several regression runs with different profile (configuration) settings, we were able to detect:

- incorrect assertion coding resulting in higher than expected scores
- misleading functional coverage for some aspects of transaction content

It is necessary to bear in mind that association of the assertions with the coverage classes used a manual rule-based approach, and also that this level of cross checking is only one aspect of coverage validation for items that are actually implemented. In effect, validating coverage using the UCIS only provides us with a sanity check for the available coverage, nevertheless this technique could also be extended to provide some level of unit testing for the coverage and assertion aspects of reusable verification components. In addition the technique demonstrates potential for automation of the coverage validation in future environments if it were extended to make use of generic formal algorithms.

## VI. RESULTS

All of the examples illustrated in the paper were from real projects. The methodology and discussion alone should raise awareness and improve the quality and accuracy of the reader's coverage models. Results from the automatic crosscheck approach using the UCIS are presented in the context of one project, but the concept itself is open to further development by the verification community and tool providers.


## ACKNOWLEDGMENT

The author would like to thank André Winkelmann (Verilab) for implementing the UCIS crosschecks and the additional PyUCIS helper methods, and Gordon McGregor (Nitero) for the original implementation of PyUCIS.


## REFERENCES


[1] UVM (*Universal Verification Methodology*), Accellera, www.accellera.org
[2] G. Allan et al, *Coverage Cookbook*, Mentor Graphics, https://verificationacademy.com
[3] J. Sprott, P. Marriott and M. Graham, *Navigating The Functional Coverage Black Hole,* DVCon US 2015
[4] UCIS (*Unified Coverage Interoperability Standard*), Accellera, www.accellera.org
[5] PyUCIS (*SWIG/Python bindings for the UCIS API*), Verilab, https://bitbucket.org/verilab/pyucis
[6] A. Yehia, *UCIS Applications: Improving Verification Productivity, Simulation Throughput, and Coverage Closure Process*, DVCon 2013


 