Pay Me Now or Pay Me Later

Exploring the Implementation and Analysis Cost Trade-Offs of Coverage Model Design

Paul Graykowski
Synopsys
Austin, Texas, United States
paulgray@synopsys.com

Andrew Piziali
Independent Consultant
Parker, Texas, United States
andy@piziali.dv.org

***Abstract* —** *In order to successfully verify a design, the scope and details of the verification problem must be quantified and measured. These are written during verification planning as the feature set of the design. Each feature has associated attributes that may be quantified with selected values and structurally arranged to reflect its nature, thereby defining its associated coverage model. The trade-offs of designing, implementing and analyzing high-fidelity functional coverage models are discussed.*

Keywords — bug, functional coverage, coverage fidelity, coverage model, functional verification

I. Introduction

In order to successfully verify a design, the scope and details of the verification problem must be quantified and measured. These are recorded during verification planning as the feature set of the design. Each feature has associated attributes that may be quantified with selected values and structurally arranged to reflect its nature, thereby defining its associated coverage model. The challenge facing the verification engineer is balancing the conflicting requirements of coverage model fidelity—how closely the model reflects the actual behavioral space of the feature—and model size—the number of points defining the model. Yet, the conflict is counter-intuitive: the greater the model fidelity, the smaller the coverage model and vice-versa! The problem is that a high fidelity model requires more labor to design and implement than a low fidelity model yet requires less back end analysis of the resulting recorded data. On the other hand, a low fidelity model may be designed and implemented with little effort but its resultant large size and questionable approximate regions require much more back-end analysis effort. Hence, we are faced with the choice to pay our labor up-front or pay up later.

In this paper we explore this trade-off using the OpenCores.org WISHBONE DMA controller[1] that compares front-end loading the labor of the coverage aspect of verification with back-end loading. The paper begins in "Related Work" by examining earlier explorations of coverage model design. We then give an overview of the DUV in "The Design Under Verification" and describe the particular features for which models are designed. This is followed by "High-Fidelity Coverage Model Design," the heart of the paper, where the procedure for designing a high-fidelity coverage model that precisely describes feature behavior and its trade-offs is explained. The next section, "Coverage Data Analysis," addresses the analysis of data recorded by the coverage model and the consequences of using low- and high-fidelity models. Finally, our key points are captured in "Summary."

II. Related Work

This work builds upon a long history of coverage measurement and analysis, driven by the need to determine how well the DUV has been exercised by a constrained random stimulus source. In [2] the use of a variety of coverage analysis techniques, ranging from FSM state transition analysis to sequence checking, are explained. [3] introduces the concept of user-defined coverage, where the coverage model is separated from the coverage tool. [4] discusses the costs and benefits of observability-based coverage, where the effects of possible errors may be observed at a DUV output. [5] discusses methods for analyzing coverage holes, those defined regions of a coverage model that have not been observed during verification but are required for verification closure. [6] addresses the complete process of defining coverage models, implementing the models, populating the models by recording coverage data, and analyzing verification progress. Finally, [7] addresses this process within an ESL flow.

III. The Design Under Verification

The DUV we use to illustrate coverage model fidelity is the OpenCores.org WISHBONE DMA controller. This core is able to transfer data between two WISHBONE interfaces while also behaving as a bridge, allowing masters on each interface to directly access storage of slaves on the other interface. Each interface shares a common clock. These are the primary features of the controller:

- Up to 31 DMA channels
- 2, 4 or 8 priority levels
- Linked list descriptors support
- Circular buffer support
- FIFO buffer support
- Hardware handshake support

The core architecture is illustrated in Figure 1 on the following page. The feature we use to illustrate coverage model trade-offs is the DMA channel, in particular the transfer kinds and associated address ranges.

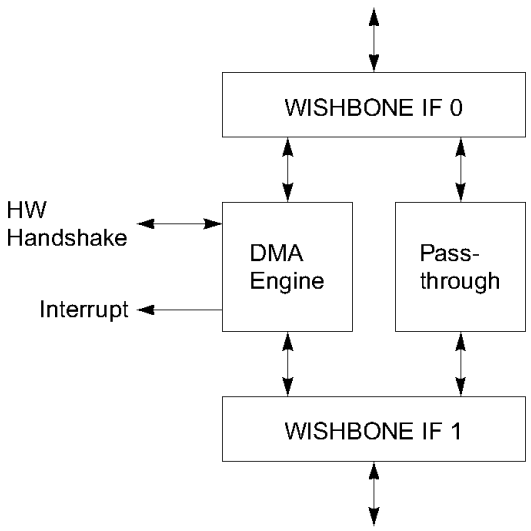

Figure 1. DMA Controller Core Architecture

IV. High-Fidelity Coverage Model Design

The behavior of each feature of the DUV—input, output and I/O behaviors—is specified in a declarative fashion by a coverage model, an abstract representation of device behavior composed of attributes and their relationships.[6] The precision of the model, i.e. how closely it describes a particular device behavior, is referred to as its fidelity. For example, a device behavior may require 15 unique states. If a model reduces these to eight states through abstraction, it would be a lower fidelity model than one that defined all 15 states. We illustrate the design of both a low-fidelity model and a high-fidelity model in the following sections and compare their trade-offs. The use of the models and analysis of data recorded for each model is subsequently addressed.

Coverage model design proceeds through two stages: top level design followed by detailed design. During top level design, we identify the semantics of a model, the attributes of the model, the values of those attributes, and the logical and structural relationships among the attributes. Detailed design is responsible for mapping the model into the verification environment, such as determining what test bench variables or DUV registers will be used as attribute sources.

A. Top Level Design

The first step of top level design is to write the coverage model semantic description. What behavioral space is the model intended to record? For example: "Record all DMA transfer kinds for each address space." This description serves as an abstract functional specification for the coverage model.

The second step is identifying the attributes of the model and their values. Each attribute contributes to or influences the behavior of the modeled feature. For this example the attributes are *transfer kind* and *address*. The attribute values are the valid subset of all possible values specified by the semantic description, in this case the named transfer kinds and address ranges. The partitioning of the 32-bit address range is based upon an expectation that end values (and their neighbors) in a range are boundary conditions more susceptable to implementation errors. This information is captured in a coverage model design table, as illustrated in Figure 2.

Attribute		Transfer Kind	Address
Value		`READ, WRITE, BLK_RD, BLK_WR, RMW`	`0000_0000… FFFF_FFFF`
Sampling Time		**transaction completes**	**transaction completes**
Correlation Time	**transaction completes**	*	`0000_0000`
		*	`0000_0004`
		*	`0000_0008… FFFF_FFF4`
		*	`FFFF_FFF8`
		*	`FFFF_FFFC`

Figure 2. DMA Transfers Low-Fidelity Coverage Model

The left column contains row headings: Attribute, Value, Sampling Time and Correlation Time. Each remaining column is associated with an attribute: Transfer Kind and Address. The legal values for each attribute are specified in the Value row. The time when the value of each attribute is to be recorded is specified in the Sampling Time row: at the end of each driven transaction. The time when the most recently sampled attribute values are to be recorded as a set in a coverage database are specified in the cell immediately to the right of "Correlation Time." In this case, the sampling times and correlation time are the same. Each remaining cell in the correlation time row specifies the particular attribute values to be recorded with the other attribute values in the same row, defining the core of the coverage model. In this coverage model we record all transfer kinds (specified by the wildcard *) for each of five address

Attribute		Transfer Kind	Address
Value		`READ, WRITE, BLK_RD, BLK_WR, RMW`	`0000_0000… FFFF_FFFF`
Sampling Time		**transaction completes**	**transaction completes**
Correlation Time	**transaction completes**	`READ, WRITE`	`0000_0000`
			`0000_0004… FFFF_FFF8`
			`FFFF_FFFC`
		`BLK_RD, BLK_WR`	`0000_0000`
			`0000_0004… 7FFF_FFF8`
			`7FFF_FFFC`
		`RMW`	`8000_0000`
			`8000_0004… FFFF_FFF8`
			`FFFF_FFFC`

Figure 3. DMA Transfers High Fidelity Coverage Model

ranges. This completes the detailed design of a low-fidelity model.

Closer inspection of the DUV specification reveals restrictions on the address ranges available to each of the transfer kinds. Although reads and writes between the two WISHBONE interfaces may use any modulo-4 address, block reads and writes—transfers to and from a single interface—are only performed in the lower half of the address space. A DMA transfer is ignored if specified in the upper half of the address range. Furthermore, atomic read-modify-writes are only defined for the upper half of the address space. Again, a read-modify-write transfer is otherwise ignored. Hence, we revise the semantic description for the new model to: "Record each DMA transfer kind with its corresponding address space." This leads to a higher fidelity version of the coverage model, captured in Figure 3 above.

B. Detailed Design

Having completed the top level design, detailed design must map the top-level design of the coverage model into the verification environment. Three questions must be answered when mapping the model:

- *What* must be sampled for the attribute values?
- *Where* in the verification environment should we sample?
- *When* should the data be sampled and correlated?

We first consider the detailed design of the low-fidelity model of Figure 2. With regard to the *what* question, we are designing a matrix coverage model[6] that captures the different WISHBONE transfer kinds and the address ranges exercised.

To answer *where* and *when* to sample, a more detailed explanation of the verification environment is necessary. The environment for the WISHBONE DMA is implemented in SystemVerilog using the VMM architecture.[8] Referencing Figure 4, within the environment is a scenario generator that generates WISHBONE transactions of type `wb_cycle`. Each transaction—a *transfer* in the parlance of the WISHBONE specification—defines the parameters used by the driver to execute a bus transaction. The `wb_cycle` is injected into a channel (like a FIFO using inter-process communication) that is routed to a `vmm_xactor` extension implementing the WISHBONE master driver. Once the master detects a new transaction in the channel, it creates a copy and drives it as a bus transaction. As soon as the transaction completes, the master subjects the transaction to a `post_cycle` callback. This callback is a hook for user code—in our case coverage recording code—to execute using the most recently transmitted transaction. The callback receives a copy of the `wb_cycle` that is used to record this coverage data. After the callback completes, it is finally removed from the channel using a destructive read and the master repeats the cycle.

We have available the transaction itself, `wb_cycle`, that contains the transfer `kind` and the address of the transaction, `addr`. A copy of this transaction is received as soon as it finishes being driven on the bus. Hence, we can define a class, `wb_master_cb_cov`, that implements the callback extension to record the coverage data.

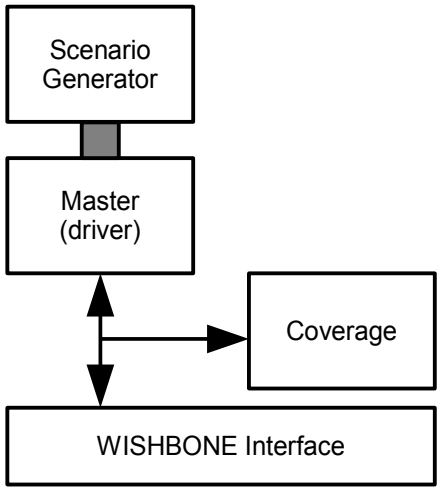

Figure 4. Coverage Model Detailed Design Architecture

As noted earlier, for every transaction that is generated and driven on the bus we need to know the kind of transfer—READ, WRITE, BLK_RD, BLK_WR or RMW—and its associated address range—0000_0000, 0000_0004, 0000_0008:FFFF_FFF4, FFFF_FFF8 or FFFF_FFFC. This is the implementation of the low-fidelity coverage model:

```
class wb_master_cb_cov extends wb_master_
callbacks;
   wb_cycle cycle;
   covergroup master_cov;
      option.per_instance = 1;
      c_kind: coverpoint cycle.kind {
         bins s_READ    = {wb_cycle::READ};
         bins s_WRITE   = {wb_cycle::WRITE};
         bins s_BLK_RD  = {wb_cycle::BLK_RD};
         bins s_BLK_WR  = {wb_cycle::BLK_WR};
         bins s_BLK_RMW = {wb_cycle::RMW};
      }
      addr: coverpoint cycle.addr[31:0] {
         bins s_LO_00   = {[32'h0000_0000]};
         bins s_LO_04   = {[32'h0000_0004]};
         bins s_MID     = {[32'h0000_0008:
                            32'hFFFF_FFF4]};
         bins s_HI_F8   = {[32'hFFFF_FFF8]};
         bins s_HI_FC   = {[32'hFFFF_FFFC]};
      }
      rwXaddr: cross c_kind, addr;
   endgroup
   extern function new();
   extern virtual task post_cycle(wb_master
      xactor, wb_cycle cycle); ...
endclass:wb_master_cb_cov
```

Since this is a callback class, the method `post_cycle()` is called at the appropriate time. We implement this method, manually updating the coverage data, providing a transaction kind and address sample for every new transaction driven on the wishbone interface:

```
task post_cycle(wb_master cycle, wb_cycle
cycle);
   // Downcast from vmm_data to wb_cycle
   $cast(this.cycle, cycle);
   // Manually trigger a coverage update with
   // the current wb_cycle
```

```
   master_cov.sample();
endtask:post_cycle
```

Implementing the correlation time for this model is straightforward because the transaction is sampled at the time the transmission completes, with both the transaction kind and address available.

C. Coverage Model Design Trade-Offs

In this section we compare the trade-offs of designing, implementing, using and analyzing a high-fidelity coverage model with the same for a lower fidelity model. The design of a high-fidelity model generally requires more effort than that of a lower fidelity model because more information is required. Moreover, precisely describing device behavior for a feature is more difficult. Consider the initial low-fidelity model of Figure 2. The five transfer kinds supported by the DMA controller were identified, along with their 32-bit address. Without any further analysis, we could quickly partition the address space into five ranges using intuition and experience, specifying a full permutation of the resultant values. This model structure is known as a matrix model[6] and is implemented using cross coverage. The coverage space size is 25 points (5 transfer kinds x 5 address ranges).

Contrast this with the high-fidelity model of Figure 3. Again, we identified the five transfer kinds but more time spent analyzing the specification revealed that some addresses are ignored for some transfer kinds. Isolating the boundary addresses in each range from the rest of the address space yields a coverage space of only 15 points ({READ, WRITE} x (3 address ranges) + {BLK_RD, BLK_WR} x (3 address ranges) + {RMW} x (3 address ranges)), structured as a hierarchical model.[6] In general, the design of a high-fidelity model requires three to four times as much time as a low-fidelity model.

As previously illustrated, implementing the coverage model for the low-fidelity model is straightforward because a common address range is considered valid for all transfer kinds, yielding a matrix model easily implemented using SystemVerilog cross coverage. However, there are trade-offs with using such a simplified model. First, we are recording coverage for many address ranges that are ignored for their transfer kinds. This unnecessarily inflates the denominator of the coverage ratio $\frac{\text{points observed}}{\text{points required}}$, making it more costly to achieve full coverage. Our decision to model the full address range for each transfer kind resulted in 25 coverage points (Figure 2). Eliminating the ignored address ranges leaves only 15 remaining points. Simulating with 100 initial seeds, the maximum coverage achieved for the low-fidelity model was 60% $(\frac{15}{25})$. This is the maximum possible because the generation constraints prevented generating ignored transactions. A careful comparison of generation constraints to the input coverage model design would have revealed this inconsistency. Moreover, the opportunity certainly exists for the verification environment tool chain to detect such inconsistencies.

To address this deficiency, we derived the higher-fidelity model of Figure 3 and implemented it as follows:

```
addr: coverpoint cycle.addr[32:0] {
   bins s_rw_lo = {0} iff
               (cycle.kind==wb_cycle::READ ||
                cycle.kind==wb_cycle::WRITE);
   bins s_rw_mid = {[32'h0000_0004:
                     32'hFFFF_FFF8]} iff
               (cycle.kind==wb_cycle::READ ||
                cycle.kind==wb_cycle::WRITE);
   bins s_rw_hi = {32'hFFFF_FFFC} iff
               (cycle.kind==wb_cycle::READ ||
                cycle.kind==wb_cycle::WRITE);
   bins s_blk_lo = {32'h0000_0000} iff
               (cycle.kind==wb_cycle::BLK_RD ||
                cycle.kind==wb_cycle::BLK_WR);
   bins s_blk_mid = {[32'h0000_0004:
                      32'h7FFF_FFF8]} iff
               (cycle.kind==wb_cycle::BLK_RD ||
                cycle.kind==wb_cycle::BLK_WR);
   bins s_blk_hi = {32'h7FFF_FFFC} iff
               (cycle.kind==wb_cycle::BLK_RD ||
                cycle.kind==wb_cycle::BLK_WR);
   bins s_rmw_lo = {32'h8000_0000} iff
               (cycle.kind==wb_cycle::RMW);
   bins s_rmw_mid = {[32'h8000_0004:
                     32'hFFFF_FFF8]} iff
               (cycle.kind==wb_cycle::RMW);
   bins s_rmw_hi = {32'hFFFF_FFFC} iff
               (cycle.kind==wb_cycle::RMW);
}

rwXaddr: cross c_kind, addr {
   // VCS option that disables auto-binning of
   // cross bins
   option.cross_auto_bin_max = 0;
   bins s_cross_rlo = binsof(c_kind)
      intersect {wb_cycle::READ} &&
      binsof(addr.s_rw_lo);
   bins s_cross_rmd = binsof(c_kind)
      intersect {wb_cycle::READ} &&
      binsof(addr.s_rw_mid);
   bins s_cross_rhi = binsof(c_kind)
      intersect {wb_cycle::READ} &&
      binsof(addr.s_rw_hi);
   bins s_cross_wlo = binsof(c_kind)
      intersect {wb_cycle::WRITE} &&
      binsof(addr.s_rw_lo);
   bins s_cross_wmd = binsof(c_kind)
      intersect {wb_cycle::WRITE} &&
      binsof(addr.s_rw_mid);
   bins s_cross_whi = binsof(c_kind)
      intersect {wb_cycle::WRITE} &&
      binsof(addr.s_rw_hi);
   bins s_cross_bkrlo = binsof(c_kind)
      intersect {wb_cycle::BLK_RD} &&
      binsof(addr.s_blk_lo);
   bins s_cross_bkrmd = binsof(c_kind)
      intersect {wb_cycle::BLK_RD} &&
      binsof(addr.s_blk_mid);
   bins s_cross_bkrhi = binsof(c_kind)
      intersect {wb_cycle::BLK_RD} &&
      binsof(addr.s_blk_hi);
   bins s_cross_bkwlo = binsof(c_kind)
      intersect {wb_cycle::BLK_WR} &&
      binsof(addr.s_blk_lo);
   bins s_cross_bkwmd = binsof(c_kind)
      intersect {wb_cycle::BLK_WR} &&
      binsof(addr.s_blk_mid);
```

```
   bins s_cross_bkwhi = binsof(c_kind)
      intersect {wb_cycle::BLK_WR} &&
      binsof(addr.s_blk_hi);
   bins s_cross_rmwlo = binsof(c_kind)
      intersect {wb_cycle::RMW} &&
      binsof(addr.s_rmw_lo);
   bins s_cross_rmwmd = binsof(c_kind)
      intersect {wb_cycle::RMW} &&
      binsof(addr.s_rmw_mid);
   bins s_cross_rmwhi = binsof(c_kind)
      intersect {wb_cycle::RMW} &&
      binsof(addr.s_rmw_hi);
}
```

While much more thought and care went into designing and implementing this model, it is much more accurate, describing more precisely the valid coverage points. The high-fidelity model has only 15 coverage points. While all 15 points are reachable (i.e. generated by our generation constraints), we found in using a regression similar to the low-fidelity model (100 seeds) that we could not reach our goal of 100%. Analysis of the results showed that the first `s_cross_*lo` and last `s_cross*hi` bins for each of the transaction types were not observed. This isn't surprising since we are generating a random 32-bit value. The likelihood of generating a specific value is one in four billion (2^{32})! In this situation, the generation constraints for the end values should be modified to increase the probability of generating these corner cases.

The labor required to code and debug the high-fidelity model was about two and a half times that of the low-fidelity model. Although the time spent implementing the model was minimal, the culprit was debugging. Since these are declarative statements, debugging coverage groups presents some challenges. For example, you cannot step through a coverage group like procedural code. Fortunately, this model is quite regular. Once a working template was implemented for one combination of address ranges and transaction kind it was easy to replicate for the other transaction kinds. This further strengthens the argument that it takes more up-front effort to design a high-fidelity model but that this time—and more—is regained once regressions begin.

This brings us to comparing the use of a low-fidelity model to a high-fidelity model in simulation. Looking at the simulation results reveals the values of Table 1. One hundred tests were executed using both the low- and high-fidelity models. The aggregate coverage reached a maximum of 65.7% after 31 tests on the low-fidelity model. As discovered through more detailed specification analysis, the model was over-simplified and had to be modified to remove unnecssary values.

Model	**Maximum Coverage**	**Number of Tests**
low-fidelity	65.7%	31
low-fidelity (redundancy removed)	90.9%	17
high-fidelity	65.5%	56

Table 1. Low- and High-Fidelity Coverage Results

This redundancy analysis reduced the size of the coverage space, enabling us to reach 90.9% coverage with only 17 runs. However, the high-fidelity model introduces a more precise model, eliminating the need for redundancy analysis. Simulation achieved a maximum coverage of 65.5% after running 56 tests. What does this imply in our simulation environment?

Disk storage and simulation time differ for each project. However, certain trends are common. A high-fidelity model records the minimum number of coverage points required to satisfy the specification. In a project with a large attribute matrix, implemented with cross coverage, expect a small increase in simulation time due to the requirement of additional guards (conditional exclusions used to implement a hierarchical coverage model using cross coverage). Guards require more conditional checks within the model but this reduces the actual number of bins and crosses stored. While more simulation time may be required, less memory is typically used. Based on prior experience, substituting a high-fidelity model for a low-fidelity model may increase the time spent recording coverage (not overall simulation time) 10-20% but the amount of memory used by coverage data often falls as much as 25-35%.

Analysis is where the high-fidelity model offers the biggest gain. Although a high-fidelity model front-end loads the development process, the back-end is the big winner. Data is recorded and may be immediately presented without further analysis or massaging. This gain is further realized as multiple regressions are run over the course of the project. The more frequent the regressions, the more time we gain. For the WISHBONE DMA controller, analysis time was reduced 90% on the first run and essentially eliminated on subsequent runs. We had only moderate confidence in the correctness of the model during the first run but this turned to full confidence in subsequent runs. We still needed to review results for each run but the time spent on this incrementally fell.

With a low-fidelity model we can converge more quickly to 100% observed coverage. However, we introduce certain inaccuracies that require a substantial amount of analysis time —i.e. coverage data must be reviewed and justified before being presented to the project team. In practice, analyzing the results is time consuming and requires deep design knowledge. Repeatedly performing this analysis for frequent regressions is not advisable because of the potential for user errors and delay in obtaining results.

The high-fidelity model, on the other hand, requires more time up-front to architect and verify. However, this doesn't have to be repeated with each regression run, making it less error prone and providing a faster turnaround. As for impact on simulation time, the low-fidelity model provided "better" results in fewer simulation runs but that time was easily lost with repeated unreachability analysis. Disk space consumption and runtime per test between the two models was not significantly different (the high-fidelity model is slightly larger). It is possible this difference would be magnified with a larger high-fidelity model but the savings in back-end analysis easily justifies this cost.

V. Coverage Data Analysis

Having compared the trade-offs of using low- and high-fidelity coverage models, in this section we address the use of three techniques for analyzing coverage results—in particular coverage holes—with an aim toward identifying the reasons for missing coverage and how to fill it. These techniques are hole aggregation, partitioning and projection.[5]

A. Hole Aggregation

In order to quickly determine why particular coverage points have not been observed after extensive simulation, identifying commonalities among the holes is required. Since our low- and high-fidelity coverage models measure input coverage—a record of stimulus applied to the DUV—a hole may reveal a defect in either the generation or coverage aspects of the verification environment. In Figure 5 we illustrate the high-fidelity model as a matrix with excluded regions. The black regions are observed coverage points; the white regions are coverage holes; and the gray regions are outside the defined

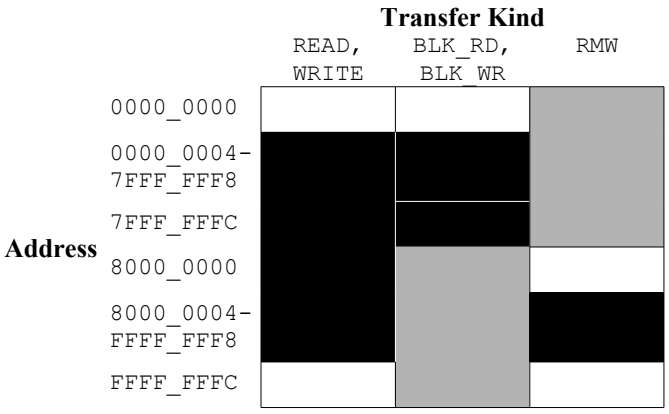

Figure 5. Hole Aggregation

space of the model (a consequence of illustrating a hierarchical model as a matrix). For example, if we represent a coverage hole as a tuple of attribute values, <attr A, attr B>, two coverage holes in this figure—<RMW, 8000_0000> and <RMW, FFFF_FFFC> may be coalesced into the single hole <RMW, {8000_0000, FFFF_FFFC}>, revealing the common attribute value of transfer kind, RMW, for the two addresses.

B. Partitioning

The second means of identifying the commonalities of coverage holes is to find out how they are conceptually or semantically similar. Lachish et al refer to this as *partitioning*. For example, regarding the three holes <{BLK_RD, BLK_WR}, 0000_0000> and <RMW, FFFF_FFFC>, we might discover, while talking with the DUV architect, that block transfers and atomic accesses to end regions of the full address space are prohibited because of timing constraints. We could group these points together into a single region for analysis.

C. Projection

A coverage hole having one or more attributes never observed may be rendered and subsequently analyzed as a projection. Projection collapses one or more dimensions of a coverage model in order to simplify it and expose common hole values. For example, if we project the transfer kind attribute, considering its five values identical, the coverage hole described earlier in Section IV.C. may be represented as <*, {0000_0000, FFFF_FFFC}>, where * represents all five transfer kinds. When presented graphically as in Figure 5, it becomes apparent we have not generated the address space end points for any transfer kinds.

VI. Summary

In this paper we stated a counter-intuitive postulate: There is an inverse relationship between coverage model fidelity and size. The primary trade-off between using low- and high-fidelity models is that a low-fidelity model postpones most labor to the end while a high-fidelity model demands it up-front. After reviewing previous work and introducing the design-under-verification, a WISHBONE DMA controller, we described the procedure for designing a high-fidelity coverage model. Top-level design requires a semantic description, followed by attributes, their values and their relationships, all captured in a coverage model design table. The refinement of an initial low-fidelity model to a high-fidelity model was illustrated. Detailed design requires answering the three Ws: What? Where? and When? Once detailed design was complete, the implementation of the low- and high-fidelity models using SystemVerilog was illustrated. We addressed the trade-offs in designing, implementing, using and analyzing low- and high-fidelity models, reporting results for each. If resources are available early in the project to design high-fidelity models, do so! Otherwise, use low-fidelity models at the outset, substituting high-fidelity models later on. Finally, three useful techniques for analyzing coverage holes—aggregation, partitioning and projection—were explained and demonstrated.

References

[1] http://opencores.org/project/wb_dma, November 2010

[2] M. Kantrowitz, L. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor," Design Automation Conference, 1996

[3] R. Grinwald, E. Harel, M. Orgad, S. Ur, A. Ziv, "User Defined Coverage — A Tool Supported Methodology for Design Verification," Design Automation Conference, 1998

[4] F. Fallah, S. Devadas, K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," Design Automation Conference, 1998

[5] O. Lachish, E. Marcus, S. Ur and A. Ziv, "Hole Analysis for Functional Coverage Data," Design Automation Conference, 2002

[6] A. Piziali, Functional Verification Coverage Measurement and Analysis, Springer, 2004

[7] B. Bailey, G. Martin, A. Piziali, ESL Design and Verification, 2007, reference chapter ten, "Post-Partitioning Verification"

[8] http://www.vmmcentral.org/, November 2010.