Defining Coverage Views to Improve Functional Coverage Analysis

Sigal Asaf
sigalas@il.ibm.com

Eitan Marcus
marcus@il.ibm.com

Avi Ziv
aziv@il.ibm.com

IBM Research Laboratory in Haifa
Haifa, 31905, Israel

ABSTRACT

Coverage analysis is used to monitor the quality of the verification process. Reports provided by coverage tools help users identify areas in the design that have not been adequately tested. Because of their sheer size, the analysis of large coverage models can be an intimidating and time-consuming task. Practically, it can only be done by focusing on specific parts of the model. This paper presents a method for defining views onto the coverage data of cross-product functional coverage models. The proposed method allows users to focus on certain aspects of the coverage data to extract relevant, useful information, thereby improving the quality of the coverage analysis. A number of examples are provided that show how the proposed method improved the verification of actual designs.

Categories and Subject Descriptors: B.6.3 [Logic Design]: Design Aids—Verification

General Terms: Verification, Measurement, Experimentation

Keywords: Functional verification, Coverage analysis

1. INTRODUCTION

Functional verification comprises a large portion of the effort in designing hardware systems [4]. In current industrial practice, most of the verification is done by generating a massive amount of tests using random test generators [1], and checking the results of running these tests on the hardware design. The main technique for checking and showing that the testing has been thorough is called test coverage analysis [7]. Simply stated, the idea is to create, in a systematic fashion, a large and comprehensive list of tasks and check that each task was covered during the testing phase. Coverage analysis helps monitor the quality of testing and directs the test generators to create tests in areas that were not adequately tested.

Cross-product functional coverage [5] is a coverage technique that is rapidly gaining popularity. In cross-product functional coverage, the list of coverage events comprises all possible combinations of values for a given set of attributes. A simple example of cross-product functional coverage checks all the possible combinations of requests and responses sent to a memory subsystem. Cross-product functional coverage can help users specify large portions of their test-plan in a simple and concise way.

Analyzing the coverage data for the memory subsystem example is relatively straightforward, since it consists of a limited number of attributes, each with a limited number of attribute values. The total number of tasks is therefore quite small, and the coverage data can be interpreted simply by looking at the individual coverage of the different combinations.

The situation becomes considerably more complicated however, when dealing with large cross-products. When analyzing the coverage data in these situations, it is neither practical, nor desirable, to look at all the events since many combinations will be illegal or uninteresting to the user. In addition, it is often necessary to abstract the data in order to understand *what* has been covered (or remains not covered), as well as *why* it was covered. A coverage tool should allow users to focus on certain aspects of the coverage data, such as specific attributes or specific attribute values. Moreover, the tool should allow the users to rapidly shift their focus. To do this, the coverage tool must provide a simple way to define views onto the coverage data that allow users to concentrate on the specific aspects of the model they are interested in, while ignoring those areas they are not. Based on the definition of these views, users can generate coverage reports that address their needs.

In this paper, we present a method for defining views onto the coverage data of cross-product functional coverage models. Our definition of views is based on three operations: projection, selection, and grouping. With projection, the coverage data is projected onto a subset of the cross-product attributes. This allows users, for example, to examine the distribution of requests in the memory subsystem model described above, and to determine if all requests have been sufficiently tested. The second operation selects specific events, for example, only events with good responses. Selection allows users to ignore parts of the coverage model that are currently not in the main focus of the verification effort, or to filter out events based on their coverage data. The third operation groups coverage events together. For example, we can group together all I/O requests and look at their combined responses.

This method for defining views onto the coverage data has been implemented, to varying degrees, in a number of coverage tools developed at IBM [5], and plays an integral role in the verification process when using these tools. The combination of this method with more complex analysis techniques, such as hole analysis [6], allows users to efficiently extract information out of large cross-product coverage models.

The rest of the paper is organized as follows. In Section 2, we explain the overall approach of functional coverage. In Section 3, we describe how coverage views are defined, and how such views help in the analysis of coverage data. Section 4 demonstrates how

DAC 2004, June 7–11, 2004, San Diego, California, USA.

Attribute	Values
Instr	fadd, fadds, fsub, fmul, fdiv, fmadd, fmsub, fres, frsqrte, fabs, fneg, fsel, ...
Result	± 0, $\pm\infty$, $\pm$Norm, $\pm$MinNorm, $\pm$MaxNorm, $\pm$DeNorm,$\pm$MinDeNorm, $\pm$MaxDeNorm, SNaN, QNaN
Round Mode	ToNearest, To0, To$+\infty$, To$-\infty$
Round Occur	true, false

Table 1: Schema attributes

Attributes				Coverage Data		
Instr	**Res**	**RM**	**RO**	**Count**	**First**	**Last**
fadd	+0	To0	false	4	08/04	08/30
fadd	+0	To0	true	0	-	-
fadd	+0	To$+\infty$	false	1	08/04	08/04
fadd	+0	To$+\infty$	true	0	-	-
fadd	+0	To$-\infty$	false	3	07/28	08/30
fadd	+0	To$-\infty$	true	0	-	-

Table 2: Coverage data

coverage views were applied in the verification of an actual storage control element. Finally, Section 5 concludes the paper.

2. FUNCTIONAL COVERAGE

Functional coverage focuses on the functionality of an application and is used to check that all important aspects of the design's functionality have been tested. When using functional coverage, coverage models containing a large set of verification (coverage) tasks are systematically defined by the user. These tasks are used to measure, monitor, and redirect the testing process. Users can perform coverage analysis to detect holes in the coverage space that expose gaps in the testing [6]. New tests can then be created to "plug" these gaps. In addition, coverage progress can be used to determine when coverage has been saturated and when a particular verification strategy can be abandoned. This is an iterative process in which the quality of the testing is continuously measured, and improvements to the testing are suggested, either manually or automatically [3], when the results of the measurement are analyzed.

A cross-product coverage model is based upon a *schema* that is composed of the following components: a semantic description (story) of what needs to be covered, a list of the attributes mentioned in the story, a set of all the possible values for each attribute, and an optional set of partitions for each attribute.

Table 1 shows the attributes and their values for an actual schema used during the verification of a floating-point unit. The model consists of four attributes – Instr, Result, Round Mode, and Round Occur – each with the possible values shown. The semantic description of the schema is: Test that all *instructions* produce all possible *target results* in the various *rounding modes* supported by the processor, both when *rounding* did and did not occur.

Each attribute of a schema can be partitioned into one or more disjoint sets of semantically similar values. This provides a convenient way for users to conceptualize their model and for analysis tools to report on coverage data. For example, we can partition the `Result` attribute according to its sign – positive, negative, or not a number (NaN) – or according to its form – zero, infinity, normalized, etc. Similarly, the `Instr` attribute can be partitioned into arithmetic and non-arithmetic instructions.

A critical step in the coverage process is to define a *coverage model* that contains only the legal events of the schema. This is because not all events in the coverage space of the schema are legal. For example, the results of executing a floating-point absolute value instruction can never be negative. It is imperative that illegal events be disregarded when doing coverage analysis, otherwise the coverage data will be skewed. All subsequent analysis is performed on the coverage model rather than on the schema itself.

The amount of coverage data stored for each event differs among the various coverage tools. For rudimentary coverage and holes analysis, simply recording which events occurred may be sufficient. Knowing the number of times an event occurred, however, is often crucial for proper analysis, since discovering lightly covered events may be as important as discovering events that have not been covered at all. In addition to coverage counts, the time an event was first covered must be recorded if coverage progress is to be computed. Some tools also record the last time an event was measured, which is useful when investigating the effectiveness of a testing strategy. Table 2 shows the coverage data for a few events in the floating-point schema.

3. COVERAGE VIEWS

The size of a schema's cross-product is determined by the number of its attributes and the number of values for each attribute. Consequently, the number of events that make up a schema can be quite large, making it difficult to cover and analyze. Eliminating illegal events helps, but does not necessarily solve the problem. The floating-point coverage model defined in the previous section is relatively simple, yet contains several thousands of legal tasks.

Bugs often occur as a result of several independent factors in the design, conspiring together to cause the problem. Such combinations of factors are difficult for the designer to conceptualize, and the problems they cause difficult to anticipate. As a result, testing for these cases typically involves looking at combinations of events that happen simultaneously. This translates into functional coverage models with many attributes and large cross-products. Making sense of coverage data by looking for patterns of coverage activity, or inactivity, in such models can be an intimidating, often futile, effort. Practically, it can only be done by focusing on different parts of a schema at different times of the verification process. To do proper analysis, it is critical that users be able to control the level of granularity in which they view the data. At times it may be necessary to take a fine, point-like perspective on only selected parts of the coverage data; at other times, it may be necessary to step back and take a coarser, broader view. Thus coverage analysis is a highly flexible, interactive, and dynamic process.

A coverage tool can help users do analysis by providing the following operations on coverage models: *selecting* out a subset of the coverage tasks that satisfy a given criteria based on specific attribute values or value combinations, or on the coverage data itself; *projecting* the coverage model onto a subset of its attributes so that some attributes are essentially ignored; *grouping* together the coverage data of attribute values that belong to an attribute partition.

Applying these operations to a coverage model, either individually or in combination, produces a set of tasks that is in some sense a subset of the model. We call the resulting set of tasks a *coverage view* on the model. It is both smaller and more focused than the original coverage model. The coverage view does not in any way alter the coverage data that has been accumulated for each event in the schema. It simply acts as an overlay or filter that is applied to the coverage data when performing analysis. It should be noted

that other operations, such as the aggregate functions found in relational algebra [2], can be useful, but were deemed too unstructured to be easily incorporated into coverage views.

3.1 Selections

The most straightforward way to reduce the number of tasks that need to be analyzed is to focus on a selected set of tasks, or conversely, to filter out unwanted ones. This is not only a practical approach to take, but a logical one as well, since the set of tasks that are of interest changes according to the verification effort that is being performed.

Selections are expressed as queries over the coverage model. Formally, we write this as $\sigma(pred)$, where `pred` is a logical expression over the attributes of the schema. For example, the selection $\sigma(Instr \in Arith \wedge RoundOccur = true)$ defines a coverage view that contains only arithmetic instructions with inexact results. Coverage reports using this view will display only those tasks that have this combination of attribute values.

The selection noted above filters out events according to some desired attribute values. Equally as important, often more so, are selections based on the coverage data itself. Such selections filter out tasks based upon the number of times tasks have been covered or when they were first or last covered. The query $\sigma(count = 0 \vee last < 08/01)$ defines a coverage view that contains only those tasks that were never covered or were not covered recently. We call selections involving coverage data *dynamic selections*, since the set of tasks they define changes as coverage progresses.

Dynamic selections can be combined with selections on attribute values, so we can look for holes or other such coverage-related constraints, within a restricted set of events. The selection, $\sigma(count < 5 \wedge (Instr \in \{fadd, fsub\} \wedge Result \in Positive)$ for example, finds all lightly covered combinations of `fadd` or `fsub` with positive results.

3.2 Projections

The purpose of projection is to answer such questions as "Have all instructions been covered?" or "What is the coverage data for all combinations of results and rounding modes in the coverage model?" Looking at the coverage data of individual events is far too cumbersome and error-prone to effectively answer such questions. A better approach is to project the coverage schema onto a subset of its attributes. Formally, this is expressed as $\pi(A_1, ..., A_n)$, where A_i is the name of some attribute in the schema. Projections can be applied to the original coverage model, or to some previously defined coverage view obtained either through a selection or by another projection.

In Table 3, we show the partial results of projecting the floating-point coverage model onto the attributes `Instr` and `Result`. Every task in this projected coverage view corresponds to a number of reflected tasks in the original model. For example, the task `<fadd,+0>` corresponds to all tasks whose `Instr` is `fadd` and whose `Result` is `+0`, that is, to tasks `<fadd,+0,To0,false>`, `<fadd,+0,To+∞,false>`, etc. The coverage count of a projected task is equal to the sum of the counts of all its reflected tasks, and the first and last times a projected task was covered is respectively the earliest and latest times of any of these reflected tasks.

The last column in the table shows the *density* of a projected task; it is equal to the ratio of reflected tasks covered to the total number of reflected tasks. Density gives an indication of the distribution of coverage for the projected task. A low density, even for tasks with high coverage counts, means that coverage is not evenly distributed among the reflected tasks.

Projections essentially control the granularity of the coverage information presented. Clearly there is some loss of information when we look at projected tasks. The fact that `<fabs,+0>` has been covered three times, tells us nothing about which tasks were actually covered and which were not. But looking at the data at this scale gives a better overall picture of the coverage, and with projection we can zoom in or out to whatever level of detail is needed.

Attributes		Coverage Data			
Instr	**Result**	**Count**	**First**	**Last**	**Density**
fabs	+0	3	08/02	08/11	1/4 (25%)
fabs	$+\infty$	0	-	-	0/4 (0%)
fadd	+0	18	07/14	08/30	3/4 (75%)
fadd	-0	5	08/11	08/11	2/4 (50%)
fadd	$+\infty$	0	-	-	0/8 (0%)

Table 3: Projected coverage data

More expressive coverage views can be built by combining projections with selections. We use selection to focus on particular tasks, and then projection to coalesce their coverage data. For example, the coverage view $\pi(Instr) \circ \sigma(Instr \in Arith \wedge Result = +0)$ projects the model onto the single attribute dimension `Instr`, but only considers tasks with `Arith` instructions and `+0` results when computing the coverage data of each instruction. Similarly, we can combine projections with dynamic selections to look at coverage data in a more compact form. This enables us to define coverage view such as $\sigma(count = 0) \circ \pi(Instr)$, which computes the set of instructions that have not been covered.

3.3 Groupings

In Section 2, we saw a number of examples of partitions of attribute values. We can use these partitions to create coverage views that look collectively at related tasks in the model. This is formally expressed as $\lambda(A_1.P_1, A_2.P_2, ..., A_n.P_n)$ where P_i is a partition on attribute A_i, defined in the coverage schema. Grouping can be combined with projection to focus on specific attributes, or with selection to focus on certain values or coverage data statistics. The coverage view, $\pi(Instr, Result) \circ \lambda(Instr.Type, Result.Sign)$, for example, both groups and projects events according to the `Instr` and `Result` attributes. For the grouping operation, the partitions `Type` and `Sign` are used respectively. The tasks in this coverage view correspond to all combinations of instruction types and result signs, such as `<Arith,Pos>`, `<Arith,Neg>`, etc. The algorithm for computing the coverage data for a grouped event is the same as for projected events. The count, first, and last time of coverage of a task is equal respectively to the sum, earliest and latest values of all of its reflected events, and its density is the ratio of covered to total number of reflected events.

Using partitions, like projections, allows us to analyze clusters of tasks rather than individual ones, thereby raising the level of abstraction at which we look at the data. Another benefit of using partitions is that attribute values that are partitioned together are frequently tested together, so that patterns of coverage activity are likely to be present throughout entire groups of attributes.

4. COVERAGE VIEW EXAMPLE: STORAGE CONTROL ELEMENT

In this section, we demonstrate how coverage views have been used on an actual, large-scale coverage model for the *Storage Control Element* (SCE) of an IBM z-series system. The simulation environment for the SCE is shown in Figure 1. It consists of four nodes connected in a ring. Each node is comprised of eight CPUs

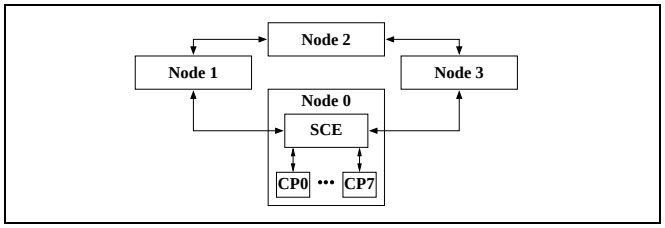

Figure 1: SCE simulation environment

(CP0 – CP7), and an SCE that handles commands from the different CPUs. Each CPU contains two cores that generate commands to the SCE independently. Incoming commands are handled by an SCE using two internal pipelines. When an SCE finishes handling a command, it sends a response to the commanding CPU.

Verification of the SCE environment involves testing all the possible command/response transactions between the various CPUs and their SCE. The schema contains eight attributes corresponding to the actual command (01, 0E, 1D, ...), the node (0-3), CPU (0-7), and CPU core (0,1) that initiated the command, the SCE pipeline (0,1) that handled the command, and three attributes that relate to the response, Response (MGW01, IGW09, DGW11, ...), Error (true, false, dontcare), and DataSource (0-3). The semantic description of the schema is: Test that all *CPU cores* have sent all possible *commands* to each of their *SCE pipelines*, and have received all possible *responses*. In addition, a number of partitions were defined, in particular, the `Response` attribute is grouped according to whether it is a normal completion (NR), an invalid address (IVA), or unrecoverable error (UE). The coverage space contains over a million tasks, but because each command has only a limited number of responses, only 18,816 of them are legal. The (partial) coverage results shown in this section were produced after several thousands of test-cases were measured, covering more than 90% of the model. Thus, they reflect a fairly mature point in the verification process.

The first item we chose to look at is the distribution of coverage of the various commands in the system. This is accomplished by projecting the coverage model onto the `Command` attribute. From Table 4, a number of revealing facts about the command coverage are immediately apparent. First, there is a great difference in the number of times each command was tested. Some commands, like `01`, were covered several million times, while others, like `1D`, occurred much less frequently. In addition, some commands essentially achieved 100 percentage coverage of their reflected events, while for others, particularly the `2E` command, the coverage density is much lower. Finally, note that there is no correlation between the number of times a command occurred and its coverage density. In general, having projected tasks with high coverage counts and low density is an indication that testing concentrated on some areas at the expense of other areas.

To further investigate the coverage of some of the commands, we looked at command response combinations of various commands. The report in Table 5 uses projection onto the `Command` and `Response` attributes, selection of three commands (`1D, 2D, 2E`), and grouping the responses according to the NR, IVA, or UE values of their partition. Note that command `2D` has only normal responses. These results show that normal completion responses for the three commands are well covered, but invalid address and unrecoverable errors less so. Moreover, for command `2E`, these two types of responses were never been hit. Further investigation revealed that under the environment setting used, these responses cannot occur for this command. This led to a change in the definition of the model. In addition, the low coverage count and low density for unrecoverable error for the `1D` command indicated that additional testing was needed in that area.

Attributes	Coverage Data	
Cmd	Count	Density
01	5161467	1664/1664 (100%)
0E	4772092	1152/1408 (82%)
1D	4244	439/512 (86%)
20	751665	256/256 (100%)
22	22270	558/640 (87%)
2D	205172	252/256 (98%)
2E	1976865	256/512 (50%)

Table 4: Command coverage report

Attributes		Coverage Data	
Cmd	Resp	Count	Density
1D	NR	2822	256/256 (100%)
1D	IVA	1344	128/128 (100%)
1D	UE	78	55/128 (43%)
2D	NR	205172	252/256 (98%)
2E	NR	1976865	256/256 (100%)
2E	IVA	0	0/128 (0%)
2E	UE	0	0/128 (0%)

Table 5: Command/Response coverage report

5. CONCLUSION

In this paper, we described various techniques for analyzing large cross-product coverage models. The basic approach is to define coverage views based on selection, projection, and partition operations. We provided two examples from actual designs, and showed how these operations, when used singularly or in combination, can improve coverage analysis.

We are currently working on methods for improving the visualization of coverage analysis results. We are also investigating various algorithms for automatic coverage analysis, such as hole and quasi-hole discovery. We believe that the power and flexibility offered by coverage views, as described in this paper, combined with improved graphical visualization and hole analysis, will greatly enhance the quality of coverage analysis of large coverage models.

6. REFERENCES

[1] A. Ahi, G. Burroughs, A. Gore, S. LaMar, C. Linand, and A. Wieman. Design verification of the HP9000 series 700 PA-RISC workstations. *Hewlett-Packard Journal*, 14(8), August 1992.

[2] E. Codd. A relational model for large shared data banks. *CACM*, 13(6), June 1970.

[3] S. Fine and A. Ziv. Coverage directed test generation for functional verification using Bayesian networks. In *Proceedings of the 40th Design Automation Conference*, pages 286–291, June 2003.

[4] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. In *Proceedings of the 1999 Design, Automation and Test in Europe Conference*, pages 434–441, March 1999.

[5] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[6] O. Lachish, E. Marcus, S. Ur, and A. Ziv. Hole analysis for functional coverage data. In *Proceedings of the 39th Design Automation Conference*, pages 807–812, June 2002.

[7] B. Marick. *The Craft of Software Testing, Subsystem testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1985.