Coverage-Oriented Verification of Banias

Alon Gluska
Intel Israel
Science Industries Center
Haifa 31015, ISRAEL
alon.gluska@intel.com

ABSTRACT

The growing complexity of state-of-art microprocessors dictates the use of cost-effective verification methods. Functional coverage was widely applied in the verification of Banias, Intel's new IA-32 microprocessor designed solely for the mobile computing market. In this paper, we describe the practical coverage approach as was carried out in the verification of Banias. According to this Coverage-Oriented verification approach, focus shifts gradually from basic logic cleanup using random testing, where verification follows a predefined test plan, to coverage-driven verification, where verification resources are steered to hit coverage holes. This practical approach enables reaching higher quality for lower effort under a tightened schedule, and provides a clear metric to measure the progress of verification and the quality of the design under test. As the conclusions will show, the retrospective evaluation of this approach shed light on its significant impact beyond original intentions, as well as uncovering several potential areas for refinement that will make this approach even more effective on future projects.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Verification

General Terms

Design, Verification.

Keywords

Logic Design, Logic Verification, Coverage, Functional Coverage.

1. Introduction

Logic verification is a major bottleneck for the completion of large hardware designs, and frequently most of design resources are dedicated to verification. Therefore, a methodological way to design the verification environment, drive its execution and measure its progress has an extremely high impact on the schedule and cost of designs.

Functional coverage, derived from the explicit functional specification of the device, is considered the answer in multiple references. The Coverage-Driven Verification (CDV) approach [9] makes coverage the core engine that drives the whole verification flow. Coverage space is defined up front, and coverage is used to measure the quality of the random testing and steer verification resources towards covering holes until a satisfactory level of coverage is attained. This, in theory, enables reaching high quality verification in a timely manner. However, the use of Coverage-Driven approach is impractical for most designs. Among the reasons are the difficulties in the application of coverage in early stages due to RTL instability, pressure on logic cleanup, not yet acquired knowledge, and the lack of supportive EDA tools.

In this paper, we present our experience with Coverage-Oriented Verification, a practical derivative of the Coverage-Driven approach. In the Coverage-Oriented approach, verification is driven first by the detection of as many RTL bugs as possible using random and direct-random tests that follow a detailed test plan. When this method produces a drop in bug detection, coverage is gradually measured and the results steer the verification toward the completion of the missing events.

Coverage-Oriented verification was applied in the verification of Banias, a microprocessor designed exclusively for the mobile computers. The Banias design introduced a wide variety of complex logic features for which a comprehensive verification approach was required. We applied advanced functional coverage techniques throughout the entire design to ensure exercising all desired corner cases. This effort directly led to RTL bug sightings, although the number of bugs found as a result of the coverage analysis was below our initial expectations. Moreover, almost all these bugs were detected in a single module.

The analysis of our comprehensive coverage effort yielded several very interesting findings. It showed that coverage had a significant impact beyond bug detection. For instance, it enforces the study of delicate mechanisms in order to improve coverage, which led in turn to improvements in testing and a higher confidence level. The analysis also led to a set of conclusions regarding the scope and size of the coverage space, the scheduling of coverage effort along the verification flow, correlation with test plans, and more. We will apply these conclusions in our next projects.

The paper is organized as follows. As a background, we briefly present Banias and its verification strategy. We will also present Coverage-Driven verification, as a basis for the Coverage-Oriented approach we took in Banias verification. We will present the results of our coverage efforts, with some analysis and explanations. Finally, we present our findings and conclusions for more effective use of coverage in verification.

2. Banias and its Verification

2.1 Modular Verification

Banias design targeted highest performance in a given power envelope, different form factors, and longer battery life. The chip holds 80M transistors and 350 functional blocks partitioned into five clusters. Its verification corresponded the design modularity.

We developed a Cluster Test Environment (CTE) for each of the RTL clusters and two Unit-Level Test environments (ULTs) for specific units. The CTEs were built using Verisity's Specman for both specification and test generation. The CTEs were designed carefully to enable verification of most of the RTL functionality. Accurate Behavioral Functional Models (BFMs) of the external world and comprehensive checking mechanisms provided the verification engineers with effective platforms that serve as the core verification platforms throughout the project.

The majority of Banias verification was carried out at the cluster level where 62% of the RTL bugs were detected. Tests developed in the cluster-level were mostly random and directed-random, with the heavy use of checkers written in Specman's *e* and C. CTEs proved to be very convenient for test development and debug. They enabled parallel progress in all clusters and a subsequent pull-in of the verification effort. However, they did not serve as the sole verification platforms because some complex interfaces could not be modeled, and checkers proved sometimes missing, incomplete or inaccurate.

Verification at the Full-Chip (FC) level consisted of the simulation of IA-32 legacy test suites, with additional ~15% of new tests developed for new features. Legacy tests were mostly directed, while the new tests were direct-random generated using an enhanced version of a proprietary IA-32 random test generator. FC verification was used primarily for features that were not modeled in the CTEs as well as a safety net for the whole design. This was very successful in hitting 37% of the bugs, several of which that were in areas missed by the cluster teams. Although the FC simulation environment was robust, it was relatively quite slow with significantly lower debug capabilities than the compact CTEs.

An additional small team used Formal Verification techniques to prove properties in few selected areas. They detected several dozens of RTL bugs, 10 of which are considered 'high quality' and had a high probability of escaping to silicon.

3. Use of Functional Coverage in Design Verification

3.1 Functional Coverage

In current industrial practice, verification consists of the generation and simulation of massive amounts of random tests. Advanced random generators can improve the quality of generated tests, but cannot detect areas in the design that are not tested while others are tested repeatedly. The main technique for checking and showing that the testing has been thorough is called coverage analysis [2], [12]). The idea is to create a comprehensive list of tasks and check that each task was covered during verification. Verification resources can be steered toward areas of low coverage, making verification efforts more effective.

In general, coverage is divided into two distinct types: *program-based* and *functional*.

In program-based, coverage tasks are automatically derived from the HDL or RTL. For example, this method will check that each statement in the HDL was executed or each transition in the state-machine was made. Program-based methods are easy to define and measure. Once the coverage metric is defined, coverage tasks are automatically derived. Many commercial program-based coverage tools are available in the market.

Functional coverage, as the name implies, focuses on the functionality of the design and is to prove that all functions undergo simulation. Therefore, functional coverage is implementation specific and coverage spaces need to be defined manually. This makes the coverage subjective and difficult to measure. In addition, very few commercial coverage tools are available. IBM presented in [1] a user-defined coverage tool that employed a relational database for coverage definition and collection. Verisity recently introduced improved coverage definition and measurement capabilities within their Specman simulation platform. Intel has its own coverage tool that is natively linked to its proprietary design environment [4].

Being tightly coupled with the quality of the RTL, functional coverage is referred to as essential in panels and methodology papers (e.g., in [1], [3], [4], [7] and [9]). However, being difficult to specify and measure, it is not yet considered as an inseparable part of the verification flow. When used, coverage is usually carried out in late stages of the design and focuses on limited parts of the RTL as a means to find escapees from the core verification effort. Functional coverage is rarely mentioned in most test cases published in recent years.

We can mention two exceptions: Functional coverage was intensively used for the verification of Pentium 4 in order to direct future testing towards the uncovered areas (see [4]). In [5], multiple coverage techniques were used at late stages of the design as a metric for the completeness of verification.

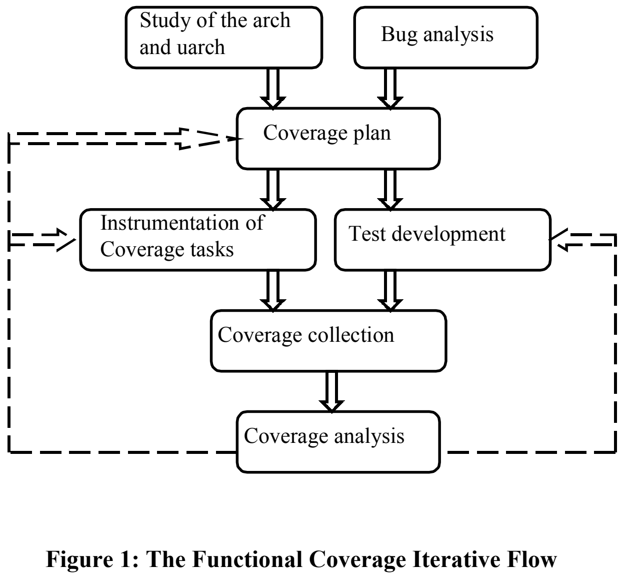

Figure 1: The Functional Coverage Iterative Flow

3.2 The Coverage-Driven Verification Approach

The Coverage-Driven Verification approach incorporates functional coverage as the core engine that drives the verification flow. It is presented schematically in Figure 1. Verification starts with a coverage plan derived from a study of the RTL functionality. The simulation environment, test generator, and coverage tools are designed accordingly to facilitate the implementation of the coverage plan. Random tests are generated and simulated. Coverage is then used to steer the verification resources toward the coverage holes. Note that feedback from coverage analysis is used for enhancements and bug fixes to the simulation environment and/or the test generator, as well as directing updates to the coverage plan. Except for specific cases, they should drive the improvement of the random testing capabilities rather than the development of specific tests.

The main idea behind the Coverage-Driven approach is that many or most of the interesting corner cases can be easily hit by random or direct-random testing. However, as long as we do not know for sure, we tend to craft tests manually and use a huge amount of random tests and simulation cycles that frequently contribute very little to the overall testing space [3]. Focusing on coverage holes and hard-to-reach cases reduces the total effort invested in verification in both headcount and computing.

Once the desired coverage is achieved, we are ready to tape-out the product. The same coverage monitors and random test templates can be subsequently reused to quickly verify modifications to the RTL and in future proliferations.

3.3 Coverage-Oriented Verification

In many cases, the pure Coverage-Driven verification is simply impractical. Design instability with the focus on bug detection, together with not yet acquired detailed knowledge, render the development of coverage monitors inapplicable at the beginning of the verification flow.

In Banias, we used a different approach that we called Coverage-Oriented. Our verification started by developing random tests according to the test plan specifications, in order to detect as many bugs as quickly as possible. Considering our future coverage efforts, we used random and directed-random testing, without embellishing corner cases or careful crafting delicate test cases. This delivered a major clean up of the RTL for a reasonable effort, while enabling the progress of other activities that are dependent on the health of the RTL such as circuit design, timing and power analysis. As was also evident, the direct-random testing that employed abstractions and testing knowledge within testbenches, enabled hitting the majority of conditions defined in our test plans. Functional coverage started only after the RTL became stable, with a drop in the bug rate, and when verification engineers acquired a detailed knowledge of the RTL. Coverage became the steering vehicle for the completion of the verification process, giving the verification engineers and the project management a quality-related picture of the convergence of verification. By that, the Coverage-Oriented approach is a practical alternative to Coverage-Driven.

4. Experience and Results

4.1 Coverage in Banias

Most of the logic verification of Banias was carried out at the cluster level. Consequently, we also performed most of the coverage tracking and analysis at this level, with complementary effort at the FC level, where coverage resulted from IA-32 legacy and new tests. Some coverage tasks could be hit only in the FC environment, either because they were difficult to model in the CTEs or because they spanned over the cluster boundaries. For others, we wanted to ensure a certain level of coverage for further confidence beyond that which was already achieved at the cluster level. This was due to the inherent uncertainty in the accuracy of CTE behavioral stubs or the completeness of checking.

We assigned resources to the instrumentation of coverage monitors when the number of RTL bugs started dropping. Schedule-wise, this occurred around the middle of the verification period, after most of the random templates had been implemented according to the detailed coverage plans.

Our primary coverage tool was Proto [11], which is used widely in Intel for coverage instrumentation and collection. Proto enables the definition of complex temporal matrices of vectors. We used Proto to build coverage monitors and measure results for a total number of 1.3 million micro architectural conditions. Out of the total number of new direct-random tests, approximately 15% were developed as a result of coverage analysis. The sum effort for coverage definition, instrumentation, collection and analysis is estimated as 12% of the total staffing resources invested in the verification of Banias.

4.2 Coverage Results

We counted 19 RTL and numerous simulation environment bugs as a direct outcome of analysis of coverage holes. This number corresponds to 5.2% of the number of RTL bugs filed in the six months preceding tape-out. The raw number was below our original expectations, and furthermore, the RTL bugs were not distributed uniformly among the design clusters. However, our study suggests that these numbers are reasonable considering the manner in which we applied coverage. Since these bugs were detected in late stages of the design and in areas of high importance, their actual severity is significantly higher than almost all other bugs revealed at that period. Moreover, at least 10 of these bugs would probably have escaped without the coverage feedback mechanism.

To better understand the impact of our coverage analysis, we classified all 360 RTL bugs that were revealed during the last six months before tape-out. The results are presented in Table 1 and show that the weight of the coverage-related bugs for functions that underwent basic cleanup and were ready to coverage was high.

Each of the *Ci*-s in the table corresponds to bugs detected in a specific cluster. SD stands for the Steer-Decode unit that is part of the C5 cluster.

Out of the 360 RTL bugs, 44% were detected in functions that were not included in the coverage space. Most of these were features, such as Power saving, Testability and Performance Monitors where verification was pushed back to later stages of the design. 15% of the bugs were detected before the corresponding function was cleaned enough to enable coverage. 10% of the bugs were in functions that went through coverage, but the specific cases were overlooked. Another 10% were detected in the cluster level following enhancements to CTEs or to checkers that improved their testing and checking capabilities. The final 8% were the result of late changes and bug fixes

The interesting and disconcerting fact is that 17 of the 19 RTL bugs derived directly from the analysis of coverage holes were found in a single unit. In all the other clusters, coverage yielded a single bug at most. A careful study of these bugs suggests that at least 10 RTL bugs in the Steer-Decode unit would have been very hard to detect without the feedback from coverage. Complexity made the controllability over the numerous internal delicate cases very difficult. It was coverage that reflected properties that have never been verified.

	C1	C2	C3	C4	C5*	SD	Total
Coverage-related	0	0	0	1	1	17	*19*
Out of coverage space	47	17	26	32	18	20	*160*
Before basic cleanup	2	4	5	0	19	26	*56*
Holes in the coverage plans	5	5	11	1	10	5	*37*
Enhanced CTE/checker feature	13	8	1	0	10	1	*36*
Inserted bug	23	1	3	0	4	3	*32*
Coverage hole not analyzed	0	0	0	0	3	4	*7*
Other	3	1	1		8		*13*
Total	*93*	*36*	*47*	*34*	*74*	*76*	*360*

Table 1: Classification of bugs detected in the six months before tape-out

4.3 Coverage Tracking in Banias

When applied over all major design functions, coverage has the potential to serve as an important indicator for the convergence of the verification process. We found the standard density indicator extracted from dividing the number of events hit by the total number of events to be insufficient. Given that a coverage space is derived from combinations of temporal vectors, hitting 100% is not the necessary target in most cases. A superior approach is to define coverage spaces as combinations of events, and then require a certain subset of events to be hit. As a result, a partial density such as 80% is difficult to interpret, whereas its distribution can deem it either satisfactory or unacceptable.

We therefore defined a metric that takes into account the density, distribution and relative importance of each coverage monitor. The following formula was used for grading the coverage of a given set of coverage monitors (e.g., all monitors of a single functional unit):

$$G_m = \frac{\sum_{i=0}^{N} W_i \frac{e_i}{E_i} \frac{\min(p_i, P_i)}{P_i}}{\sum_{i=0}^{N} W_i} * 100$$

Where:

- N is the number of monitors belong to the set
- W_i is the weight of the specific monitor
- E_i is the target number of coverage elements in the desired level of distribution
- e_i is the actual number of covered elements
- P_i is the target percentage of covered events
- p_i is the actual percentage of covered elements

The coverage grades for any level of hierarchy up to the whole design were calculated according to:

$$G_u = \frac{\sum_{i=0}^{N} W_i G_i}{\sum_{i=0}^{N} W_i}$$

Where N is the number of sets in the lower level, and W_i and G_i are respectively the weight and grade per set.

The metric we defined provided a hierarchical quality-related indicator for the design health. Coverage was tracked on a weekly basis and was used in addition to traditional indicators such as the number and severity of RTL bugs detected.

5. Conclusions

After tape out, we conducted a detailed study of our coverage experience. The objectives of the study were to evaluate the impact of the coverage effort, to understand the reasons for the small number and non-uniform distribution of RTL bugs, and, consequently, to identify the necessary steps required to make coverage more effective. We studied the execution of coverage and its results along the verification flow, and dispatched a detailed questionnaire answered by all involved engineers.

We identified the following reasons for the relatively low number of RTL bugs found by the coverage effort:

- As also reflected in Table 1, coverage was applied in almost all clusters to areas that were thoroughly verified using random and direct-random tests. No coverage was carried out for features of relatively lower criticality, such as performance monitoring and DFT where verification was executed during the later stages of the project.
- Our simulation environments were very effective in bug detection. In spite of being new and incomplete, we detected 62% of the total RTL bugs in the cluster level. Excluding the testing space that was not supported by the CTEs, the percentage is significantly higher.
- In many cases, our coverage was too detailed. This hampered our focus on the riskier features. In addition, we generated

coverage spaces that were too large to be adequately covered.

- And finally, in many cases our test plans were not coverage friendly. Test conditions were described in general terms only, with inconsistent references to specific signals or time windows This allowed subjective interpretation of test plan content and erroneous implementation of coverage monitors.

The Steer-Decode unit was an exception. This unit was most complex and introduced major new features in Banias in order to improve throughput and timing. As a result, the verification of the unit significantly lagged behind for the duration of the project. The time pressure dictated an initial cleanup using random and directed-random only, along with trimming the coverage space to consist of the risky features. This enabled coverage to reveal cases that would hardly be hit in any alternate method we applied.

5.1 Conclusion #1: Impact of coverage is beyond the number of bugs

The ultimate goal of verification is to reveal all functional bugs. Accordingly, the impact of activities is evaluated according to the number of RTL bugs it exposed. We believe that this should not be the only parameter for functional coverage. Coverage analysis provides the feedback for the accuracy and effectiveness of the random testing. It also serves as a quality-related indicator for the convergence of verification and thus as an important criterion for tape-out.

In Banias, about half of the coverage-related bugs were considered very hard to find by other means. The related function was complex, with many low-controllability parameters. The constraint-solver embedded in the random test generator enabled hitting a significant portion of the coverage space, but only coverage revealed combinations that had been never generated.

In addition, almost all engineers claim that coverage instrumentation, and even more coverage analysis, enforced the study of micro-architecture and RTL details, and consequently led to the development of more creative tests. Analysis of coverage holes revealed cases that were actually impossible, and shed light on factors that were originally neglected. This served towards the modification of the coverage plan.

A further outcome of the coverage effort is raising the confidence level of the verification engineers. As long as verification consists on random tests, it heavily relies on the accuracy and completeness of the testbenches and the quality of test generation. Uncertainty tends to lead to the generation of massive number of tests. Feedback from coverage enables trimming expensive simulation cycles without impacting the quality targets.

5.2 Conclusion #2: Focus and prioritize coverage

Similar to other techniques, coverage is not uniformly effective for the varied types of validation tasks. Clearly, the more complex and potentially buggy a feature is, the more attention it deserves. However, additional considerations should be taken into account when identifying features for which coverage yields a smaller ROI. Among them:

- Controllability. Some functions can be directly hit from the boundaries of the simulation environment. For example, the decoding and execution of instructions in all architectural modes can be verified by generating appropriate ASM tests. In such cases, it may be more cost-effective to hit the desired scenarios by exhaustively driving all the appropriate sets of inputs, rather than coding them in coverage monitors.
- Extensiveness of random testing. Frequently, analysis of tests and bugs lead to a strong indication, and consequently to a high confidence level, that random testing exercise the desired features very well. Coverage measurement of such areas is likely to reflect the quality of testing, but will has no added value to the quality of the design.

5.3 Conclusion #3: Don't specify what you cannot cover

An easy way to produce coverage spaces is by the enumeration of values for multiple vectors. In Banias, for example, we required the coverage of back-to-back bus transactions that were associated with numerous of fields (e.g., request), each of multiple possible values (e.g. code-read, data-read, write, etc.). The outcome space was huge and included several million entries that were very difficult to hit. Analysis of coverage holes did not point to any clear issue; it was rather the large space that made it difficult to increase the drive coverage numbers up. We eventually dropped that space and replaced it by multiple less aggressive ones. Each of these small spaces was directed to a specific small subset of the original huge domain. Therefore, analysis of holes became a manageable and effective task.

5.4 Conclusion #4: Start coverage just before failure rate drops

In Coverage-Oriented verification, focus shifts gradually to coverage. So, when is the right time to start coverage?

There is no simple answer. In general, coverage should start when bug rate drops., and we near the point where the potential of the random testing is exploited to its fullest. We can continue with the development of tests directed toward more and more specific corners of the design. Such tests require an increasing effort to develop and verify their correctness. In addition, such tests can hardly be generalized to consist of random selections that make their scope wider. A generally better approach is to develop coverage monitors instead. If our testbench is of reasonable quality, we may find that a significant portion of those corner cases has been already hit.

In addition, at this point in the project, verification engineers have already acquired sufficient understanding of the design to more accurately implement coverage monitors and analyze their results.

5.5 Conclusion #5: Test plans should be Coverage-Aware

In Banias, we crafted detailed test plans that specified the scenarios to be generated and reviewed them thoroughly. We then used these test plans to drive the development of tests and coverage monitors, as well as test generation and checking capabilities. In many cases, test plans that served very well for the development of high-quality tests were vague and incomplete for coverage.

In order to be Coverage-Aware, test plans should:

- Formally specify the coverage space
- Refer to well-defined events including the specific RTL signals
- Define the expected coverage target as defined by density and distribution
- Define the relative importance level of each of the coverage monitors

The formal definition is necessary to allow a smooth interpretation of a test plan entry into a coverage monitor. This need is magnified further considering that engineers other than those who wrote the test plans may do this at a later stage. For coverage purposes, test plans need to be reviewed thoroughly as soon as coverage monitors are developed. When possible, a additional preliminary review should take place as soon as RTL becomes available.

5.6 Conclusion #6: Use coverage to improve test generation

Random testing should be not only legal, but also 'interesting'. This is achieved by embedding testing knowledge within the simulation environment to increase the chance of hitting interesting, frequently rare, cases. Coverage analysis can identify inaccuracies in implementation. In particular, these can be the inaccurate specification or selection of corner cases in large spaces, or the improper distribution of fields within their possible value spaces. This latter type occurs frequently when fields are interdependent and are selected using a constraint solver.

Coverage should be used to ensure that the test generation is tuned according to specification. This requires measuring the distribution of basic events and accuracy of implementation. In Banias, analysis of coverage holes revealed environment bugs in almost all CTEs.

6. Summary

We applied the Coverage-Oriented Verification approach for the verification of Banias. In the Coverage-Oriented approach, focus at the beginning is given to catching the easy-to-find bug detection using random testing. When bug detection becomes harder and number of bugs drops, functional coverage gradually becomes the main driving strategy behind the detection of the hard-to-find bugs.

In the paper, we described the Coverage-Oriented approach*,* and our experience in Banias. Coverage enforced the study of low-level design details and the development of creative tests. Analysis of coverage holes led to the detection of 19 RTL bugs, most of them in a single unit. And finally, using a coverage-based metric, we produced a quality-related indicator for the convergence of verification. As with most first time efforts, we have identified several areas for improvement that can make coverage more effective. Among them are a better definition of coverage schedule, more coverage-friendly test plans, manageable scope of coverage spaces, etc.

In spite of the advanced, aggressive and complex features and the relatively small verification team, Banias logic was exceptionally clean with only very few logic bugs detected in silicon. This suggests that Banias Coverage-Oriented verification approach and execution was very successful. We therefore recommend the Coverage-Oriented Verification approach as a practical alternative for Coverage-Driven.

7. References

[1] R. Grinwald, E. Harel, M. Orgad, S. Ur, A. Ziv "User Defined Coverage – A Tool Supported Methodology for Design Verification". DAC 98, 158-163.

[2] B. Marick "The craft of Software Testing, Subsystem Testing Including Object-Based and Object-Oriented Testing". Prentice Hall. 1985.

[3] Paul Gingras, "Panel: Functional Verification – Real Users, Real Problems, Real Opportunities", DAC 1999, 260-261.

[4] Bob Bentley. Validating the Intel Pentium 4 Microprocessor. DAC 2001.

[5] S. Taylor, M. Quinn, D. Brown, N. Dohn, S. Hildenbrandt, J. Huggins, C. Ramey "Functional Verification of a Multiple-issue, Out-of-order, Superscalar Alpha Processor". DAC 98, 638-643.

[6] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal "Coverage Directed Generation Using Symbolic Techniques", FMCAD Confenernce November 96.

[7] S. Ur, Y. Yadin "Micro-Architecture Coverage Directed Generation of Test Programs", DAC 99, 175-180.

[8] L. Fournier, A. Koyfman, M. Levinger "Developing an Architecture validation Suite". DAC 99, 189-193.

[9] "Coverage-Driven Functional Verification", white paper by Verisity. http://www.verisity.com/html/coverage_driven.html.

[10] A. Vincentelli, P. McGeer, A. Saldanha, "Verification of Electronic Systems", Tutorial in DAC 96, 106-111

[11] B.E. Nelson, R.B. Jones, and D.A. Kirkpatrick, "Simulation Event Pattern Checking with PROTO", Proc. of the International Conference on Simulation and Hardware Description Languages, 1994

[12] Tasiran, S ; Keutzer, K, "Coverage Metrics for Functional Validation of Hardware Designs"