Enhancing the Control and Efficiency of the Covering Process

Shai Fine **Avi Ziv**
IBM Research Lab in Haifa
Haifa University Campus
Haifa, 31905
Israel
email: {fshai, aziv}@il.ibm.com

Abstract

Coverage Directed Test Generation (CDG) is a technique for providing feedback from the coverage domain back to a generator that produces new stimuli to the tested design. In this paper, we describe two algorithms that act in a CDG framework. The first algorithm controls the coverage events distribution using a "Water-Filling" approach. The second algorithm improves the efficiency of the covering process using clustering techniques.

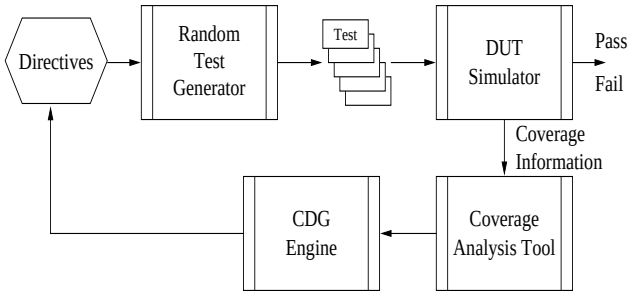

Figure 1. Feedback-based CDG process

Introduction

Functional verification is widely acknowledged as the bottleneck in the hardware design cycle [1]. To date, up to 70% of the design development time and resources are spent on functional verification. The increasing complexity of hardware designs raises the need for the development of new techniques and methodologies that can provide the verification team with the means to achieve its goals quickly and with limited resources.

The current practice for functional verification of complex designs starts with a definition of a test plan, comprised of a large set of events that the verification team would like to observe during the verification process. The test plan is usually implemented using random test generators that produce a large number of test-cases, and coverage tools that detect the occurrence of events in the test plan. Analysis of the coverage reports allows the verification team to modify the directives for the test generators and to better reach areas or specific events in the design that are not covered well [5].

The analysis of coverage reports, and their translation to a set of test generator directives to guide and enhance the implementation of the test plan, result in major manual bottlenecks in the otherwise highly automated verification process. *Coverage directed test generation* (CDG) is a technique to automate the feedback from coverage analysis to test generation. The main goals of CDG are to improve the coverage progress rate, to help reach non-covered events, and to provide many different ways to reach a given coverage event. Achieving these goals should increase the efficiency and quality of the verification process and reduce the time and effort needed to implement a test plan. Figure 1 presents a feedback-based verification process. The CDG engine receives information from the coverage analysis tool, regarding the state and progress of the coverage, and generates directives to the random test generator that are designed to achieve one or many of the CDG goals.

In a previous work [4], we described a new approach for coverage directed test generation. Our approach is based on modeling the relationship between the coverage information and the directives to the test generator using *Bayesian networks* [10]. Simply stated, the CDG process is performed in two main steps. In the first step, a training set is used to learn the parameters of a Bayesian network that models the relationship between the coverage information and the test directives. In the second step, the Bayesian network is used to provide the most probable directives that would lead to a given coverage event (or set of events).

We conducted several experiments (two of which are described in [4]) to test the capabilities of the suggested framework to handle aspects of the CDG problem in various settings. In all the experiments, the Bayesian network based CDG framework was able to reach the main goals of CDG, listed above. A common theme in the experiments described in [4] is that the CDG engine is used to generate directives for specific, single, coverage events. The CDG framework in general, and the Bayesian network approach specifically, are not limited to this mode of operation. Instead, it can be used to generate more generic directives that target a subset of coverage events or a general area in the

coverage space.

In this paper, we explore these capabilities of the CDG framework and describe two algorithms that make use of a CDG engine to control the distribution of the covered events and to improve the efficiency of the covering process. The first algorithm attempts to improve the coverage process efficiency by clustering together related events and generating one set of directives that attempts to cover all events in the same cluster. The clustering is based on the set of directives that was generated by the CDG engine for each event. That is, the algorithm clusters together coverage events whose associated directives are similar. This approach is different from other clustering techniques for coverage analysis (such as those described in [9]), where the clustering is based on similarity in the coverage space.

The second algorithm is designed to reach a desired distribution for the number of times events are covered. For example, to reach uniform coverage, where all events are covered the same number of times. The algorithm uses a "Water-Filling" approach to generate directives that fills the gap between the current distribution and the desired one.

For both algorithms, we provide some experimental results on simple test cases. These results illustrate how the proposed algorithms work and their benefits. Although our experiments were done with the Bayesian network based CDG framework, the algorithms described in this paper are not limited to that engine and can be used with any feedback based CDG engine.

Increasing Coverage Efficiency via Clustering Techniques

A feedback based CDG engine, as shown in Figure 1, can be used to complete the coverage of a given coverage model in the following way: At each iteration the current coverage state is inspected, and then the engine is used to produce the appropriate, possibly multiple, sets of directives, designed to cover the remaining non-covered events. This process continues until the entire coverage model is covered.

A common practice, while using simulation based verification techniques, is to focus attention on the progress of coverage as a function of time or the number of simulation runs. This information is often used to assess the quality of the verification process and estimate the resources needed to complete the coverage of the model [6]. Therefore, "coverage rate" and "coverage completion", i.e., the slope and the length of the progress graph (respectively), are good measures for coverage efficiency.

A CDG engine can be used not only to achieve full coverage, but also to improve the coverage rate and increase the coverage efficiency. One possible approach to increase coverage efficiency, is through careful selection of the order of events to be covered. The possible gain is due to the observation that a simulation run most often covers many other events in addition to the one it was designed to cover. Thus,

```
Let NE be the set of non-covered events
while notempty(NE)
        for each ne ∈ NE
            d_ne = CDGengine(ne)
        end
        C = Cluster({d_ne})
        for each c ∈ C
            d_c = CDGengine(c)
        end
        Run simulations using the set of directives {d_c}
        Update NE
end
```

Figure 2. Coverage via clustering techniques

one may suggest, for example, a covering strategy in which the design of directives to stimulate easier-to-reach events is postponed, hoping that these events will be covered anyhow in earlier stages of the covering process. We would like to note in passing that some CDG engines, such as the BN engine[4], may actually produce a measure which, for example, will predict how likely it is that one event will be covered via directives designed to stimulate another event. Obviously, this type of information is valuable for designing a good covering strategy. However, in this sequel we do not rely on such advanced capabilities of the CDG engine.

Even after exploiting any a-priori knowledge (e.g. ordering the events based on their reachability hardness), and exploiting available semantic (domain) knowledge (e.g. partition the coverage space into sub-regions of events that share some semantic proximity, such as error events), we may achieve better results than the ones obtained by activating the CDG engine to yield directives that stimulate each event at a time. For this we use the fact that the proximity of events is not determined solely by their semantics, but also by the way the simulation environment stimulates them. In the level of abstraction at which the CDG procedure is acting, the later proximity is reflected by the similarity of the designed directives. For clarity of representation, we will limit the discussion to simulation environments in which these directives are sets of numeric values selected from prescribed ranges and reflect assignments to parameters of the test generation (e.g., weights to bias draws of the random test generator, enable/disable flags, etc.). Thus, a metric space is an appropriate choice to embed these directives, measure similarity, and group together events with similar associated directives. Having a partition of events, one may either select out of each group the set of directives that is most similar to the other sets of directives associated with events in the group, or reproduce a new set of directives adequate for the whole group of events (if the CDG engine has this capability). The important point to note is that the grouping is done in a syntactic level (similarity of directives embedded in some metric space), and therefore off-the-shelf clustering techniques can be used to cluster the events and to select representative sets of directives.

In a nutshell, cluster analysis is the process of grouping

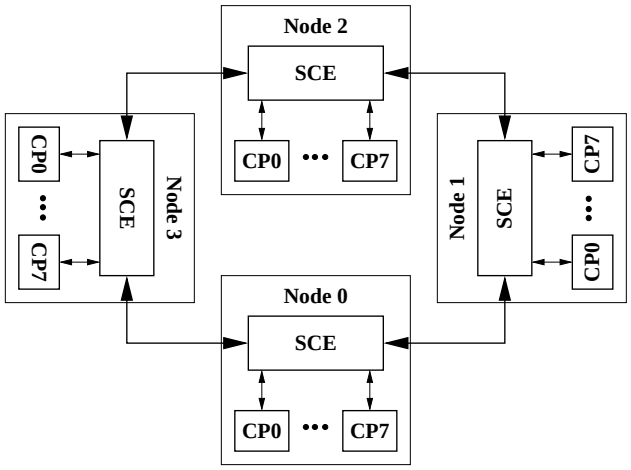

Figure 3. Structure of the SCE simulation environment of an IBM z-series system

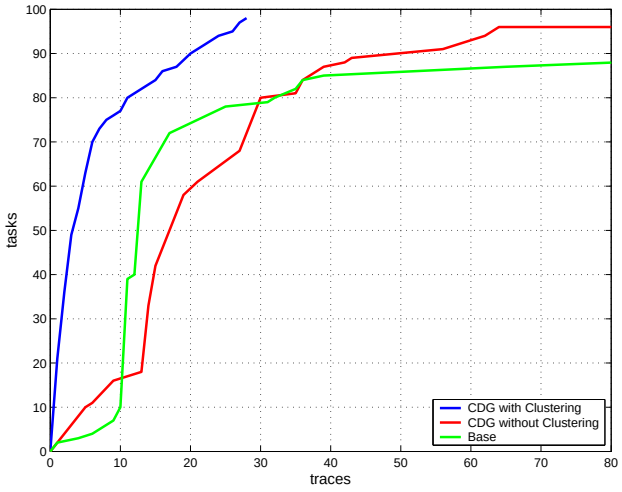

Figure 4. Coverage progress of the unrecoverable error (UE) subset of the SCE

or segmenting a collection of objects into a set of meaningful subclasses, called *clusters*, based on their similarity, so that the degree of association is strong between members of the same cluster and weak between members of different clusters. Each cluster thus describes the class to which its members belong. Clustering is essentially an unsupervised learning technique, i.e., the data elements in the training set are not pre-classified. This means that there is no reference, textbook solution, to use for comparison. Thus, cluster analysis is often used as a tool of discovery. It may reveal associations and structure in data which, though not previously evident, nevertheless are sensible and useful once found. The results of cluster analysis may contribute to the definition of a formal classification scheme; suggest statistical models with which to describe populations; indicate rules for assigning new cases to classes for identification and diagnostic purposes; provide measures of definition, size and change in what previously were only broad concepts; or find exemplars to represent classes.

The variety of techniques for representing data and measuring the proximity between data elements, the various forms of grouping data elements (hard/fuzzy partitioning), and the techniques to control and measure cluster validity (assessing how good the cluster configuration is), have produced rich literature and an overwhelming variety of clustering methods. The diligent reader is advised to consult [8] and references therein.

The approach we took in this sequel deviates from classical clustering techniques in that the grouping of elements in one space is determined by the proximity of related elements in a different space; coverage tasks are grouped together if their associated stimuli are similar. Our approach is reminiscent of the work of Pereira, Tishby and Lee [11], which suggested clustering words based on associated context distributions. Thus, the resulting clusters group together words that share a typical context, which in turn can be interpreted as their latent cause. Our intuition seems to agree with this notion of proximity that yields a target oriented clustering, since in the course of the CDG process, the grouping of coverage events is useful if we can trigger them all within a single simulation run.

In a deeper sense, an approach that suggests a shift from targeting specific to groups of events, actually considers trading accuracy (of the designed directives to stimulate the desired events) and efficiency (in terms of the number of simulations); this notion is central to the clustering approach and its related model selection techniques (cf. [3]). In the context of coverage, this notion implies that as the covering procedure progresses, the remaining non-covered events will become less similar to one another. A reasonable clustering technique is expected not to group them together. Therefore, they will be stimulated using their specifically designed directives. In other words, a smooth transition back to the high-accuracy end occurs spontaneously.

Experimental Results

We tested the suggested method on a subset of a coverage model used in the verification of the *Storage Control Element* (SCE) of an IBM z-series system, as shown in Figure 3. The environment and coverage model used in the experiment are similar to those used in [4]. The environment contains four nodes that are connected in a ring. Each node is comprised of a local store, eight CPUs (CP0 – CP7), and an SCE that handles commands from the CPUs. Each CPU consists of two cores that generate commands to the SCE independently. Each SCE handles incoming commands using two internal pipelines. When the SCE finishes handling a command, it sends a response to the commanding CPU.

The coverage model consists of all the possible transactions between the CPUs and the SCE. It contains six at-

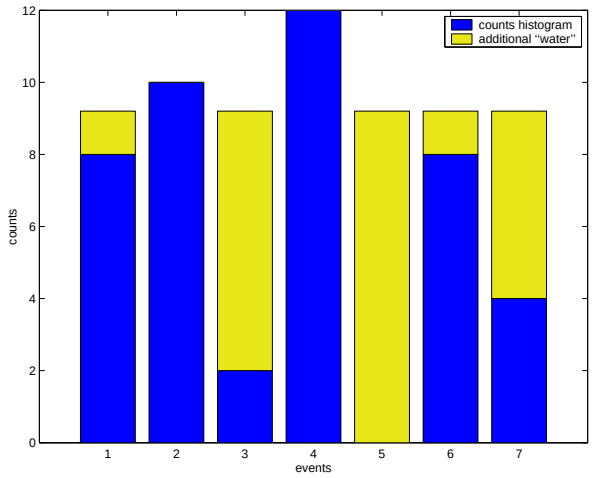

Figure 5. The *Water-Filling* approach

tributes: The core that initiated the command, the pipeline in the SCE that handled it, the command itself, and three attributes that relate to the response.

In the experiment, we concentrated on a subset of the coverage model that deals with unrecoverable errors (UE).[1] The size of the subspace is 98 events. We tried to cover the subspace using three methods. First, in the base case, we used two sets of directives, provided by the verification team of the SCE, that attempt to create unrecoverable errors. The second method generated directives using a CDG engine, for each non-covered event in the UE space, and run simulation until all the events were covered. In the third method, we applied the algorithm of Figure 2, while using the *K-medoid* clustering method[2]. Figure 4 presents the coverage progress for all three methods. The figure shows that with the clustering method we were able to cover the entire UE subspace with just 29 traces. On the other hand, the CDG using the single-event method covered 96 events after 64 traces, while the base was able to cover only 89 events.

Biasing Coverage Distribution using a "Water-Filling" Approach

A coverage event is comprised of many different behaviors that lead to it. Therefore, in many cases it is desirable not only to cover all the events in the model, but also to control how many times each event is covered, by biasing the distribution of their counts towards a desired distribution. Obviously, we cannot reduce the existing number of counts, so we need to manipulate future simulations such that the additional counts they produce will bias the total (accumulated) counts towards the desired distribution. Because our control over the simulation environment is limited, it is reasonable to expect a convergence to the desired distribution via multiple simulations. Moreover, the desired distribution may not be reachable. Thus, we need to come up with a coverage strategy that will not only converge as fast as possible, but also produce reasonable intermediate results (distributions).

The strategy that we adopt is the "Water-Filling" approach, a well known technique in the information theory realm[2]. In a nutshell, we assume to have given a (limited) amount of "water"[3] that will be used to bias the distribution, and we pour it over the current histogram of counts such that events with a lower number of counts will be "filled" first (cf. Figure 5). Note that the amount of water to be used is insufficient to achieve the desired distribution. Let m_i denote the number of times (count) event i has been covered, and $M = \sum_i m_i$ denote the total number of counts so far. Let Z denote the estimated total number of counts that will be produced in the next simulation[4] (i.e. the amount of "water" that is allowed to be used), and let n be the cardinality of the coverage space. The "Water-Filling" strategy translates to designing a probability distribution $\tilde{\mathbf{p}} = \{\tilde{p}_1, \ldots, \tilde{p}_n\}$ over the coverage space such that[5]

$$\begin{aligned} \forall i \qquad & m_i + \tilde{p}_i \cdot Z = \frac{Z+M}{n} \\ \forall i \qquad & \tilde{p}_i \geq 0, \quad \sum \tilde{p}_i = 1 \end{aligned} \tag{1}$$

The designed distribution $\tilde{\mathbf{p}}$ is fed to a CDG engine, which is capable of producing the appropriate stimuli to the simulation environment to meet the requested distribution of counts. Our limited control over the simulation environment, will force us, in practice, to run multiple simulations, each time with a newly designed $\tilde{\mathbf{p}}$, and converge towards the desired uniform distribution. Moreover, as this iterative procedure proceeds, the limited amount of "water", Z, will become the main bottleneck for attaining the desired distribution, The difference $\delta_i = \frac{Z+M}{n} - m_i - \tilde{p}_i \cdot Z$ represents the "missing" amount of water, which needs to be added to p_i in order to get a uniform distribution. Negative δ_i reflects the fact that we would like to "reduce" the number of counts of event i in order to achieve uniform distribution. Obviously, this is not possible and the best that we can do (subject to independence assumptions) is to set $\tilde{p}_i = 0$ and distribute the amount of "water", Z, among the other events respectfully. Mathematically speaking, the design of an appropriate "Water-Filling" $\tilde{\mathbf{p}}$ translates to a linear programming problem, which can be easily solved by the procedure specified in Figure 6.

[1] This part of the model was not part of the experiment in [4].

[2] A clone of the well known *K-means* alg., cf. [7] and references therein.

[3] This is the estimated total number of counts that will be produced in the next simulation.

[4] Since we use a random simulation environment this number is unknown and can be affected by various factors, some of which are out of our control.

[5] In this sequel, we limit the discussion to address only a desired *uniform* distribution. This is done for clarity of presentation, and the extension to handle the general case is trivial.

Let $\mathbf{m} = \{m_1, \ldots, m_n\}$ be the event counts sorted in ascending order
For $i = 1 \ldots n,\ z_i = 0$
while $Z > 0$
    Find the index, I, and height, Δ, of the next "stair"
    Let $\delta = \min\{\Delta, Z/I\}$
    For $i = 1 \ldots I,\ z_i = z_i + \delta$
    $Z = Z - \Delta \cdot I$
end
$\tilde{\mathbf{p}} = normalize(\{z_1, \ldots, z_n\})$

Figure 6. The "Water-Filling" algorithm

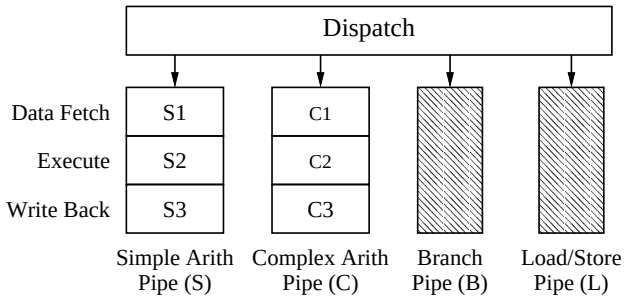

Figure 7. Structure of the NorthStar pipeline

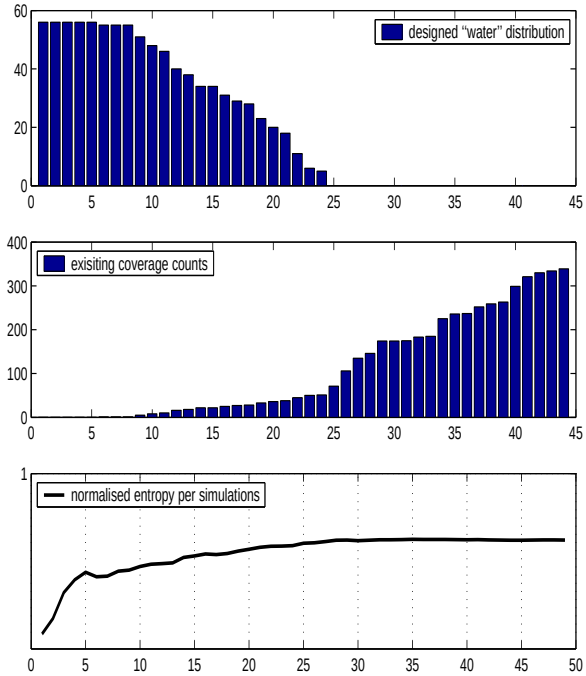

Figure 8. Biasing the counts distribution at the NorthStar coverage space

Experimental Results

The following experiment demonstrates use of the "Water-Filling" approach to bias coverage space distribution. We conducted the experiment on a model of the pipeline of NorthStar, an advanced PowerPC processor. The pipeline of NorthStar contains four execution units and a dispatch unit that dispatches instructions to the execution units. Figure 7 illustrates the general structure of the NorthStar pipeline. Each execution unit consists of three pipeline stages:

- Data fetch stage, in which the data of the instruction is fetched
- Execute stage, in which the instruction is executed
- Write back stage, where the result is written back to the target register

We used a coverage model for two of the execution pipelines, the simple arithmetic and complex arithmetic pipelines. The coverage model examines the state of the execution pipelines and properties of the instructions in them. It consists of five attributes: the type of instruction at stage 1 of the simple and complex arithmetic pipelines, flags indicating whether stage 2 of the pipelines are occupied, and a flag indicating whether the instruction at stage 2 of the simple arithmetic pipeline uses the condition register. The total number of legal coverage tasks in the model is 54. The model of the NortStar pipeline and the coverage model used in the experiment are identical to those used in [4].

The goal of the experiment was to achieve the most uniform distribution possible for event counts in the coverage space. To evaluate performances, we used the entropy measure [2], which measures the "similarity" of a given probability distribution $\mathbf{p} = \{p_1, \ldots, p_n\}$ to the uniform distribution

$$H(\mathbf{p}) = -\sum_i p_i \log p_i \qquad (2)$$

Starting from an empty coverage, a feedback-based CDG process was executed, such that at each iteration the current distribution of counts was used by the "Water-Filling" algorithm to design a probability distribution over the coverage events, which in turn biased the accumulated counts towards uniform distribution. The designed "Water-Filling" distribution was fed to the CDG engine, which produced the appropriate directives to the simulation environment. The upper and middle panels of Figure 8 provide a snapshot of this procedure after 50 simulations. The middle panel presents the sorted counts distribution of the least frequent 44 (out of the 54) events, while the upper panel shows the requested distribution of counts for the same events, as designed by the "Water-Filling" algorithm and fed to the CDG engine. Finally, the lower panel presents the progress of the normalized entropy across the first 50 simulations, where the normalized entropy is the entropy measure of a given probability distribution divided by the entropy measure of the uniform distribution. Since the uniform distribution maximizes the entropy (cf. [2]), the normalized entropy ranges between 0 and 1 - The closer it gets to 1, the more similar to uniform is the given probability distribution.

Concluding Remarks and Future Work

In this paper we described two algorithms that make use of a CDG engine to better control and improve efficiency of the covering process. For both algorithms, we provided some experimental results on simple test cases. These results illustrate how the proposed algorithms work and their benefits. Although our experiments were conducted using BN-based CDG engines, the algorithms described in this paper are not limited in that sense, and can be used with any other CDG engine. Developing techniques that exploit the advanced capabilities of the BN-based CDG engine is left for further study.

References

[1] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[2] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, 1991.

[3] R. O. Duda and P. E. Hart. *Pattern Classification and scene analysis*. Wiley, 1973.

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

[5] 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 (DATE)*, 1999.

[6] A. Hajjar, T. Chen, I. Munn, A. Andrews, and M. Bjorkman. High quality behavioral verification using statistical stopping criteria. In *Proceedings of the 2001 Design, Automation and Test in Europe Conference (DATE)*, pages 411–418, March 2001.

[7] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[8] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

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

[10] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Network of Plausible Inference*. Morgan Kaufmann, 1988.

[11] F. C. Pereira, N. Tishby, and L. Lee. Distributional clustering of English words. In *Proceedings of the 31st meeting of the Association for Computational Linguistics*, pages 183–190, 1993.