Coverage Driven Distribution of Constrained Random Stimuli

Marat Teplitsky, Amit Metodi, and Raz Azaria
Cadence Design Systems, Israel
{marat | ametodi | razaz}@cadence.com

***Abstract*- Functional verification is performed by covering all the required scenarios of the device under test. To facilitate the monitoring of the verification process, these scenarios are tracked with a coverage model in the hardware verification languages. With coverage closure being the major challenge in verification, many verification teams seek better ways to automate coverage closure. In this paper, we present an approach that utilizes the simulator's random stimuli generation engine to distribute the random results to satisfy functional coverage requirements defined in the coverage model. We discuss the technology in use, and present experimental results showing the improvement in coverage convergence rate when using our recommended approach compared to the traditional constraint random approach.**

I. INTRODUCTION

Functional verification tests a hardware device by simulating device-under-test (DUT) functionality. Verification is performed by stimulating the DUT with a wide variety of inputs to ensure that all aspects of the DUT have been tested. To plan and monitor the tested functionality, methodologies based on coverage, checks, and assertions such as the Metric Driven Verification (MDV) [4] were introduced.

To express the functional requirements that should be tested, hardware Verification Languages (HVL) like the '*e*' language [1] or SystemVerilog (SV) [3], enable users to define a coverage model – a set of cover points that specify what data should be observed while examining the input, internal and output signals going through the DUT.

Coverage closure is achieved when the entire coverage space, spanned by the coverage model, has been observed. This process is the measurable closure of the entire verification effort. However, we must keep in mind that the actual goal of verification is to remove bugs, and thus coverage closure is only a well-defined and measurable derivative of that goal. Therefore, the path taken while reaching a goal is as important as reaching the goal itself.

As designs complexity grew over the years, the convergence of the verification effort became a central challenge in hardware verification. With the enormous amount of stimuli required to qualify a DUT, a constrained random verification (CRV) approach became the heart and soul of verification, and the dominant way to progress towards coverage closure. In this approach, the protocols and specifications that the stimuli must obey are modeled using a constraint language, and the verification engine provides random scenarios to stimulate the DUT. Therefore, a constraint language became an essential part of any HVL. First in the '*e*' language [1], and later on, adopted by other languages such as SystemVerilog (SV) [3] and SystemC [2].

A significant problem commonly reported by verification experts [14] is that the random approach does not achieve coverage closure because over time, the random stimulus ends up repeating without contributing to the coverage closure, thus leaving interesting scenarios uncovered. There are five methods, listed in the remainder of this section, that are being used by verification teams to address this problem. However the current alternatives are not living up to the needs of the verification teams.

The first method is the simplest, the least effective, and yet the most often practiced method to cover for the lack of convergence in the CRV approach. This method is to manually direct tests to cover each of the remaining coverage holes. While a small number of intelligent, human-crafted tests can bring a lot of value in reaching complex and interesting scenarios, on larger scale, the amount of human effort to write all the required direct tests does not scale with today's schedule and productivity requirements from the verification teams.

Alternatively (that is, in the second method) verification teams can rely on computing power and run the random tests exhaustively until a larger portion of the coverage requirements is covered. However studies and experience show that this approach is limited because the effort increases exponentially, while the test coverage and the reduction in problems found increases linearly [13].

One way to improve this exhaustive approach (the third method), is to modify the constraint model and affect the tool's distribution to achieve better coverage. This approach is significantly better than the previously mentioned methods. However, the current industry paradigm for this method is the engineer-defined distribution, meaning that

engineers rather than machines direct the random stimuli generator. This is done by modifying the constraint model, thus affecting the tool's distribution of random results using distribution directives that are part of the verification languages. Unfortunately, this paradigm leaves the critical issue of conveying the functional coverage requirements to the engineer, instead of automating the process. In addition, the task of analyzing and improving the constraint and coverage models does not scale, leaving verification teams with no productive solutions for bringing verification to closure. In section III of this paper, we take a close look at distribution directives and the obstacles they present to the verification teams.

Eventually, some EDA providers took up the challenge of coverage closure. One approach to increase the reached coverage and reduce the dependency in distribution directives is by increasing the diversity of the functional scenarios created natively by the random stimuli generators. Random generators can do it by understanding the functional intention of the constraint model, and directing the random results accordingly [17]. However, while this result is significantly better than the traditional uniform-distribution approach, it cannot bridge the gap of lacking the actual coverage goals.

Another approach is by using tools to create a pool of stimuli that achieve the defined functional coverage requirements on a specific class, which is later injected into the simulation engine. The creation of the pool is done by iterating over the defined coverage space, and generating stimuli externally (outside of the simulation context) that cover that space. During simulation, stimuli from that pool are injected as direct tests in order to fulfill the functional coverage requirements.

This external stimuli creation approach has inherent limitations. A major one is an inability to react to the run-time state of the DUT and verification environment, thus limiting the ability to reach functional coverage requirements that are too complex to project and plan the exact stimuli that can reach them. Other limitations include the runtime and memory overhead with vast solution space.

In this paper, we challenge the engineer-defined distribution paradigm and propose a new paradigm of coverage-driven distribution that overcomes all drawbacks of the currently practiced methods, described above. In the coverage-driven-distribution paradigm, the distribution of the random stimuli is directed towards the user's functional coverage requirements. This is done by deriving the desired distribution from the user's coverage model and using it to influence the random stimuli generator distribution during stimulation. A random stimuli generator that applies coverage driven distribution provides better out of the box distribution targeting functional intention, and improves coverage convergence rate compared to traditional constraint random approach.

The proposed paradigm was implemented in the 14.2 release of Cadence Incisive® solution with Specman® '*e*' [6], within the IntelliGen [7] constrained random stimuli generator. IntelliGen is designed to work with environments with hundreds of thousands of variables and constraints, and is the tool of choice for many of the most demanding verification environments in the industry. One of the key capabilities of IntelliGen is that it provides out of the box distribution that reflects the user intentions. Applying coverage-driven distribution in IntelliGen improves its out of the box distribution even further.

II. Problem Definition

Functional verification tests a hardware device by a simulating or emulating the behavior of a device-under-test (DUT). To bring verification to closure, the constrained random test generation approach became the heart and soul of verification. In this approach, verification engineers model the protocols and specifications that the stimuli must obey, using a constraint language which is part of a Hardware Verification Language (HVL), such as SystemVerilog or '*e*'. The constraint model is then used to generate diverse random stimuli to produce interesting test scenarios.

However, when verification teams report that the constraint random approach does not bring closure to the verification coverage and effort, and verification experts and bloggers recommend falling back to heavy direct testing to compensate [14, 15], a deeper and more critical look at the constrained random test generation approach is required.

A key to the quality of constrained random approach is the distribution of the random results provided by the verification tools. The random stimuli generator's distribution determines not only what verification scenarios will be covered, but also directly impacts how fast the verification effort will be completed. The question of "What is a good distribution?" has been discussed in length in earlier works [8, 9, 10]. While the intuitive answer is uniform distribution, and indeed SystemVerilog IEEE standard [3] requires the random stimuli generator to generate a solution that uniformly distribute over the solution space, this solution is rarely effective in real life scenarios.

In [11], we claimed that "the purpose of randomization is to cover as many functional scenarios as possible, and that uniform distribution over the solution space does not reflect this intention". Moreover, the random distribution provided by most verification tools is derived from the constraint model, and is not aligned with the coverage model. Therefore, the verification tools are usually unable to fulfill the objectives requested by the coverage model. This problem was identified and explored by researchers and EDA providers in the previous decade [5, 16].

In the following example, we use two HVLs, SystemVerilog and '*e*', to illustrate how verification engineers may describe the constraint and coverage models, and illustrate the gap between the solution space defined by the constraint model and the coverage space defined by the coverage model.

Suppose that verification engineers need to generate a set of packets that vary in size, to be written into a memory. The packets can be classified into one of three sizes, SMALL, MEDIUM or LARGE, derived from the packet size, *size* field. The packets are written to memory starting from the address *s_addr* field, and ending in the address, *e_addr* field. Figure 1describes the constraint model of packet and its fields.

The verification engineers may suspect that the design suffers from bugs when the packets are written to the edges of the memory address (that is, when the start address is close to 0 or the end address is close to 0xfffffff). To ensure that the set of generated packets covers those corner cases, the verification engineer defines a coverage model as described in Figure 2.

To achieve complete coverage closure, packets from each size (SMALL, MEDIUM and LARGE) must be generated and for each size, packets with start address and end address from each of the bins declared for them.

Unfortunately, when verification engineers use a random stimuli generator to generate large amount of packets, most objectives in the coverage model are rarely achieved. This is due to the extremely low probability of generating a needed packets (in this case, a packet different than a large packet with start address *s_addr* greater than 47, and with end address *e_addr* smaller than 0xfffffd0) when assuming the solutions distributed uniformly over the solution space. Figure 3a illustrates the distribution of packets when using uniform distribution over the solution space. Figure 3b illustrates the desired distribution – uniform distribution over the coverage space. (Note that Figure 3b reflects the bias enforced by the constraint model reflected by biased cover of several points while targeting only one.)

The essence is that the coverage model (Figure 2) focuses on the values in the edges of the memory address, but the constraint model (Figure 1), does not contain any directives that can be interpreted to direct the random stimuli generator towards those objectives. Therefore, the solutions generated by the random stimuli generator are distributed differently than desired.

```
type sz_type_e : [SMALL, MEDIUM, LARGE];

struct packet {

   sz_type : sz_type_e;
   s_addr : uint;
   e_addr : uint;
   size : uint;

   keep s_addr <= e_addr;
   keep size == (e_addr - s_addr);
   keep (sz_type==SMALL) => (size < 1k);
   keep (sz_type==MEDIUM) => (size in [1k..64k]);
   keep (sz_type==LARGE) => (size > 64k);
};
```

```
typedef int unsigned t_uint32;
typedef enum  {SMALL,MEDIUM,LARGE} size_type_e;
class packet;
rand size_type_e sz_type;
rand t_uint32 s_addr;
rand t_uint32 e_addr;
rand t_uint32 size;
constraint legal {
   s_addr <= e_addr;
   size == (e_addr-s_addr);
   (sz_type==SMALL) -> (size < 1024);
   (sz_type==MEDIUM) ->  (size inside { [1024:65536] });
   (sz_type==LARGE) -> (size > 65536);
}
endclass
```

Figure 1: Constraint Model in 'e' and SystemVerilog

```
cover pkt_gen is {
   item s_addr using
     ranges = {
       range([0..0], "BIN_S_1");
       range([1..7], "BIN_S_2");
       range([8..47], "BIN_S_3");
       range([48..MAX_UINT], "BIN_S_4");
   };
   item e_addr using
     ranges = {
       range([0x00000000..0xffffffcf], "BIN_E_1");
       range([0xffffffd0..0xfffffff7], "BIN_E_2");
       range([0xfffffff8..0xfffffffe], "BIN_E_3");
       range([0xffffffff..0xffffffff], "BIN_E_4");
   };
   item sz_type;
   cross s_addr, sz_type;
   cross e_addr, sz_type;
};
```

```
covergroup pkt_gen;

 s_addr: coverpoint t_uint32'(s_addr) {
   bins BIN_S_1 = {[0:0]};
   bins BIN_S_2 = {[1:7]};
   bins BIN_S_3 = {[8:47]};
   bins BIN_S_4 = {[48:32'd4294967295]};
 }
 e_addr: coverpoint t_uint32'(e_addr) {
   bins BIN_E_1 = {[0: 'hffffffcf]};
   bins BIN_E_2 = {['hffffffd0: 'hfffffff7]};
   bins BIN_E_3 = {['hfffffff8: 'hffffffe]};
   bins BIN_E_4 = {['hffffffff: 'hffffffff]};
 }
 sz_type : coverpoint size_type_e'(sz_type);
 cross_s_adde__sz_type : cross s_addr, sz_type {}
 cross_e_adde__sz_type : cross e_addr, sz_type {}

endgroup
```

Figure 2: Coverage model in 'e' and SystemVerilog

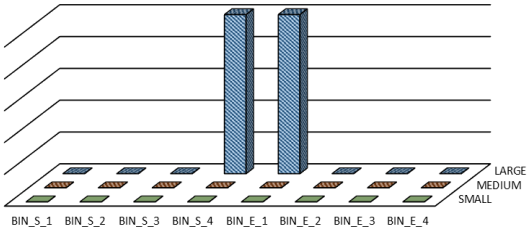

Figure 3a: Constraint distribution

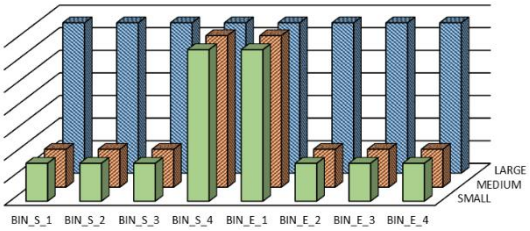

Figure 3b: Coverage distribution

In the next section, we discuss distribution directives – the solution provided by HVLs to influence the distribution of the random stimuli generators.

III. Distribution Directives

Generation of random stimuli relies heavily on the random distribution of the verification tool in use. In many cases, the distribution provided by the random stimuli generator is not aligned with the user's intentions. This is because in most verification tools, the random distribution is based solely on its interpretation of the solution space of the constraint model, and therefore overlooks the user's functional intentions and functional coverage requirements. To compensate for the imperfection of the random stimuli generators, and to direct the random generation towards the proper distribution, distribution-directives are used.

The current paradigm in functional verification is that it is the engineers' task to define the distribution of the verification tool, using distribution directives. In this section we describe the use of distribution directives, and the difficulties that the engineer-defined-distribution paradigm poses for the verification teams.

There are two main ways to control distribution: through distribution constraints and through order directives. Distribution constraints, such as 'select' in '*e*' and 'dist' in SystemVerilog, are constraints which direct a field or an expression to be generated according to a given distribution. On the other hand, order directives, such as 'solve...before' in SystemVerilog, are constraints which direct the random generator to solve a specific field or fields before other fields. However, many problems arise from using distribution directives in verification.

First of all, the verification engineers are required to have a deep understanding of the HVL distribution directives semantic and the technical solving process. For instance, in HVLs the distribution constraints have order semantics which must be considered. Failing to understand the specific order semantic of the distribution constraints causes the distribution be different than intended.

In addition, the verification engineers need to be familiar with the entire constraint model since in large designs the relations between fields are tangled, and adding a distribution directive on one field can influence the distribution of other fields. Adding distribution directives increases the complexity of the model, and eventually harms the distribution, not really helping to verify the design.

Lastly, the engineers need also to be familiar with the entire coverage model since not all objectives can be achieved simultaneously, and therefore they cannot be trivially projected at the constraint model without them overriding each-other.

To demonstrate the use of the user defined distribution paradigm, let's now reconsider the example from Section II, with its constraint model in Figure 1and the coverage model in Figure 2. To direct the random stimuli generator towards the desired functional coverage requirements, the verification engineer must add distribution directives to the constraint model. Figure 4 contains a possible set of distribution constraints to influence the random stimuli generator to generate solutions in a distribution which uniformly covers the objectives in the coverage model.

A closer look at the distribution constraints in Figure 4 reveals that the verification engineer had to repeat the information from the coverage model in Figure 2 and write a distribution constraint for each one of the coverage items.

Even for the given example, adding distribution directives to achieve coverage closure is not a simple task. For instance, if the order of the distribution constraints in Figure 4 would have been modified, it would have caused the distribution to be different than intended. Furthermore, SystemVerilog code requires adding the order directive, 'solve…before' to achieve the desired distribution.

Since the process of adding distribution directives to achieve coverage closure is not productive and requires deep understanding of many aspects, verification teams are left with bad alternatives, such as falling back to direct testing, or running exhaustive random simulations, as discussed in Section I.

In the next section, we introduce a new paradigm that eases the burden on achieving coverage closure. This is the coverage-driven-distribution paradigm, which tunes the random distribution of the random stimuli generator to achieve a better coverage closure, without the verification engineers need to add additional coding layers.

```
keep soft s_addr == select {
   1 : [0..0];
   1 : [1..7];
   1 : [8..47];
   1 : [48..0xffffffff];
};
keep soft e_addr == select {
   1 : [0x00000000..0xfffffcf];
   1 : [0xfffffd0..0xfffffff7];
   1 : [0xffffff8..0xffffffe];
   1 : [0xffffffff..0xffffffff];
   4 : pass;
};
keep soft sz_type == select {
   1 : SMALL;
   1 : MEDUIM;
   1 : LARGE;
};
```

```
constraint dists {
   s_addr dist {
      [0:0] :/ 1,
      [1:7] :/ 1,
      [8:47] :/ 1,
      [48:'hffffffff] :/ 1
   };
   e_addr dist {
     ['h00000000:'hfffffcf] :/ 1,
     ['hfffffd0:'hfffffff7] :/ 1,
     ['hffffff8:'hffffffe] :/ 1,
     ['hffffffff:'hffffffff] :/ 1
   };
   sz_type dist {
      [SMALL: LARGE] := 1,
   };
   solve sz_type before e_addr, s_addr, size;
   solve s_addr, e_addr before size;
}
```

Figure 4: Distribution Constraints in 'e' and SystemVerilog

IV. COVERAGE DRIVEN DISTRIBUTION

A common paradigm asserts that it is the job of a verification engineer to direct the random stimuli generator towards the desired results, using distribution directives that are part of the verification languages, as discussed in Section III. In this section, we introduce a new paradigm called coverage-driven-distribution (CDD) that challenges the commonly practiced, engineer-defined distribution. In the coverage-driven-distribution paradigm, the random stimuli generator reads the functional coverage requirements, in addition to the constraint model, and derives the desired distribution to achieve them. The derived distribution complements the default distribution of the random generator and directs the generator toward "a better distribution" than the one understood from analyzing the constraint model alone.

The question of "What is a good distribution?" rises once again. In response to the traditional approach in the verification community to accept the uniform distribution over the solution space as the proper distribution for constrained random verification, we asserted in [11] "that the purpose of randomization is to cover as many functional scenarios as possible, and that uniform distribution over the solution space does not reflect this intention."

Taking also the functional coverage requirements in consideration, we can refine our claim [11] that "purpose of randomization is to cover as many functional scenarios as possible" and claim that the best distribution to achieve coverage closure is a uniform distribution of stimuli over the *coverage* space. There are several ways to achieve uniform distribution over the coverage space, discussed below.

In the previous section, we discussed that verification engineers control the random stimuli generator's distribution by manually writing distribution directives. This approach has many conceptual drawbacks as previously described, and it eventually does not scale to the needs of verification teams.

A better approach, used by other EDA tools, is to generate the stimuli independently of the simulation engine, and to inject them iteratively to the simulation. However, these are insensitive to the environment, the run-time inputs, and the DUT state.

A third, new approach, presented here, is of Coverage-driven-distribution (CDD).This approach overcomes the deficits of the previous approaches in the following manner.

First, the random stimuli generator automatically derives the proper distribution directives from the functional coverage requirements which become a part of the constraint model. This is done automatically by the verification tool, without the verification engineer having to be familiar with the entire constraint and coverage models, or having to repeat the functional coverage requirements in the form of directive constraints.

In addition, by working inside the simulator and being a part of the random constrained process, the CDD paradigm provides a more flexible verification flow. Unlike the approach that uses predefined stimuli, CDD achieves the coverage requirements while still being random, being responsive to run-time variables such as DUT state, and being capable of giving up on certain coverage goals and pursuing others when needed. By doing that, CDD is much more suited to reach interesting scenarios and find bugs on its way to covering the coverage goals. And this, as we said earlier, is the true goal of verification.

Among the other benefits of using CDD as part of the constraint model is the ability to understand, debug, and reproduce the verification results. Moreover, the constraint model allows continued enforcement of other directives such as soft constraints, react to DUT state, and pick the functional coverage requirements that can be achieved in the current state without failing or bringing the DUT to illegal states.

One of the simplest and most straightforward ways to obtain functional coverage requirements, and the one we highlight in this paper, is to derive them directly from the coverage model. Other methods to derive them include reading real-time coverage results or using other coverage planning methods, and are not in the scope of this paper.

A naïve implementation of CDD would be to automatically convert each element from the coverage model to a standard distribution directive which will be added to the constraint model, as done in the example in Figure 1, Figure 2, and Figure 4. However, while automatic conversion of coverage model to a set of standard distribution directives seems straightforward for the given example, applying such conversion in a more complex environment and coverage model might raise some difficulties.

Converting the coverage model to a set of distribution directives to achieve desired distribution requires understanding of the HVL being used. When using '*e*', one must understand of the order semantics of the distribution constraints; and it might require adding additional policies to some distribution constraints. When using SystemVerilog, the use of order directives, 'solve…before', might be required and a wrong usage can cause undesired behavior of the random stimuli. But one of the major drawbacks is that distribution constraints are defined on a single expression (field or a set of fields that are connected through relation, function or mathematical operators) and when translating a coverage model containing cross coverage to distribution directives, a distribution constraint on a set of expressions is required. When the coverage item/cross contains an "ignore" attribute which should come into consideration, it makes the translation more complex.

Moreover, the constraint model might contain user-defined soft constraints and distribution directives. Adding distribution directives derived from the coverage model raises a question regarding enforcement order. While enforcing the derived distribution directives first might improve coverage distribution, it might also override user-defined soft constraints and distribution directives.

Another aspect need to be considered when adding distribution directives to the constraint model is performance. Since generation is affected by inputs (values sampled during stimulation), some functional coverage requirements cannot be achieved since they depend certain input values. Trying to enforce the derived distribution directives as a top priority can consume memory and take a lot of time.

The proposed paradigm was implemented in the 14.2 release of Cadence Incisive solution with Specman '*e*' within the IntelliGen constrained random stimuli generator.

Specman reads the coverage model in addition to the constraint model, extracts objectives from the coverage model and creates a set of distribution directives which become part of the constraint model. To overcome the '*e*' order semantic of 'select' distribution constraints, Specman has a new internal kind of distribution directives, named as "coverage directives". The coverage directives are similar to 'select' distribution constraints in functionality but do not have the '*e*' soft semantics. In addition, coverage directives can be also be defined on set of expressions to support cross coverage which defined on more than one element.

Existing user-defined 'select' and soft constraints need to come into play when searching for valid random stimuli. Specman enforces user-defined soft constraints prior to any coverage directives by default. In addition, Specman may give up on some coverage directives if it detects that the effort to fulfill them in the current state is too big.

One of the key features of Specman is debugability [12], the ability to debug the solving process. Debugging a solving process which involves the new coverage directives is possible and natural since the coverage source code is presented in the debugger as the reason to the relevant distribution choices, similarly to 'select' distribution constraints.

To summarize, our solution describes a way to improve the out of the box distribution of the random generation based on the overlooked information in the coverage model. Moreover, this solution doesn't contradict the use of higher level solutions based on the random constraint generator, and can be integrated to any other higher level solution. In the next section, we demonstrate how using the coverage-driven-distribution improves the coverage convergence rate compared to the traditional constraint random approach.

V. Experimental Results

In this section, we demonstrate the improvement in coverage closure achieved by using coverage-driven-distribution paradigm. The proposed paradigm was implemented in the 14.2 release of Cadence Incisive solution with Specman '*e*', within the IntelliGen constrained random stimuli generator.

The experiments compares coverage score in three run modes:

a. SystemVerilog constraint model using Cadence Incisive Enterprise Simulator, referred to here as 'SV CR'. Note that the SystemVerilog constraint random stimuli generator generates solutions in uniform distribution over the solution space, as the SystemVerilog IEEE standard requires.
b. '*e*' constraint model using Cadence IES Specman with IntelliGen constrained random stimuli generator, referred to here as 'e CR'. Note that unlike SystemVerilog, '*e*' IEEE standard does not specify a required

distribution of the random generator. IntelliGen constrained random stimuli generator provides out of the box distribution that tries to reflects the user's functional intentions.

c. '*e*' constraint model and '*e*' coverage model using Cadence IES Specman with IntelliGen constrained random stimuli generator while applying Coverage Driven Distribution paradigm, referred to here as 'e CDD'.

Note that, the SystemVerilog constraint and coverage models are equivalent to the '*e*' models.

Let's review the example from Section II. In that example, we demonstrated the generation of 1000 instances of the type packet from the constraint model in Figure 1; the coverage model from Figure 2contains 3 cover items, 2 cover crosses and a total number of 35 bins.

When using 'SV CR' for generation, all sz_type packets were generated to LARGE, s_addr is in [48..0xffffffff] and e_addr in [0x00000000..0xffffffcf], as seen in b. When using 'e CR' for generation, the generated sz_type packet values distributed evenly over the possible values (SMALL, MEDIUM and LARGE), while s_addr is in [48..0xffffffff] and e_addr in [0x00000000..0xffffffcf] as in SystemVerilog. However, in both cases, most of the objectives in the coverage model (Figure 2) were left uncovered.

When adding the distribution directives from Figure 4, both cases achieved full coverage closure.

By applying coverage-driven-distribution ('e CDD') on the constraint model from Figure 1 and the coverage model from Figure 2, without the distribution directives from Figure 4, full and uniform coverage of the coverage space is achieved as illustrated in Figure 5.

Figure 5 demonstrates that applying CDD on section II's example directs the random stimuli generator towards the coverage objectives, and the generated packets distribute in the desired distribution as illustrated in Figure 3b.

To demonstrate the power of the coverage-driven-distribution paradigm, we examine it using the Cadence Denali AMBA AXI4 protocol implementation, and perform 1000 randomize calls of an AXI transaction in the three run modes 'SV CR', 'e CR', and 'e CDD'. Figure 5shows the coverage score achieved per number of instances generated.

The SystemVerilog and the '*e*' constraint and coverage models are equivalent. The constraint model doesn't contain distribution constraints and the coverage model contains 33 cover items, 13 cover crosses and the total number of bins is 1651. (To estimate the convergence rate from the experimental results, in Figure 6 we interpolate the exponential convergence formula of $cov = 1 - \alpha e^{-\beta x}$, where α is a linear factor, β is the coverage convergence rate exponent, and x is the number of transactions.)

As can be seen in the experiment on the AXI model (Figure 6), the CDD approach achieves exponentially faster coverage of the verification goals than the alternatives: 0.003 for CDD, vs 0.001 and 0.0008 for 'e CR' and 'SV CR' respectively. And indeed. 'e CDD' achieved better coverage score than the other two during every step of the experiment, until reaching a score of 97.51% after generation of only 1000 AXI transactions. This is while 'e CR' and 'SV CR' get 79.51% and 68.95% coverage rate respectively.

An intriguing result on the plot in Figure. 6 is the crossover between SV and 'e' after 160 transactions. While not a point of study in this paper, we can estimate that 'e' has a higher exponent of convergence due to its ability to predict the functional intentions in the constraint language as described in [11], and that the very early crossover after only 160 transactions may be due to statistical or implementation factors.

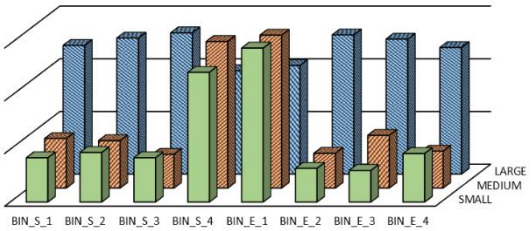

Figure 5: Coverage-Driven-Distribution distribution

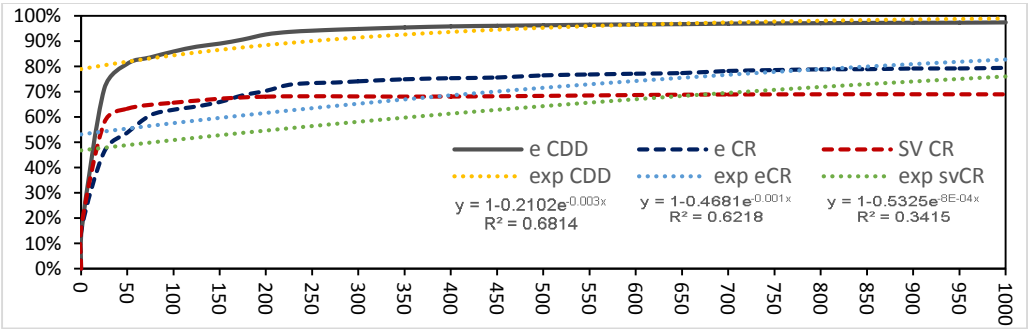

Figure 6: Coverage score per number of instances of AXI transaction

VI. Summary

In this paper, we reviewed the problem of not reaching coverage closure when using the constrained random verification approach. Modern HVLs give the verification engineers tools, such as distribution directives, to direct the naïve random generators toward function coverage requirements. But the effort required to achieve coverage closure using distribution directives is very large and becomes even larger as the design complexity grows.

Automatic coverage closure tools were developed by EDA providers to assist verification teams by automating the process. However, as of today, the value and usability of these tools has been limited, both because of the immature technology, and the hard abstraction, modeling and maintenance work that it requires for each piece of the verification model.

In this paper, we introduced the coverage-driven-distribution paradigm. In this paradigm, the constrained random stimuli generator analyzes the functional coverage requirements, in addition to the constraint model, derives the desired distribution, and then directs its default distribution in order to improve coverage convergence rate and coverage closure.

We discussed the difficulties of converting the coverage model to distribution directives and how our implementation, which derives a new special kind of distribution directives from the coverage model, overcomes them. By applying our implementation on existing constraint and coverage models, such as AXI4 transaction, we demonstrated how CDD's improves the out of the box distribution of the random generator and can bring engineers a step closer towards coverage closure.

In addition, by working inside the simulator and being a part of the random constrained process, the CDD paradigm provides a more subtle and flexible verification, making the journey towards the coverage goals better suited to finding bugs.

With CDD's improved coverage convergence from inside the simulator, we believe that CDD should be the basic and underlying technology for tools aiming to fully automate coverage closure.

References

[1] "Ieee standard for the functional verification language e," *IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008)*, pp. 1 –495, 2011.

[2] "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1 –638, 2012.

[3] "Ieee standard for systemverilogunified hardware design, specification, and verification language," *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pp. 1 –1315, 2013.

[4] H. B. Carter and S. Hemmady, "Metric-driven design," 2007.

[5] A. Piziali, *Functional verification coverage measurement and analysis*. Springer, 2004.

[6] D. Robinson, *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*. Morgan Kaufmann, 2010.

[7] V. Lagoon, "The challenges of constraint-based test generation," in *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*. ACM, 2011, pp. 1–2.

[8] V. Gogate and R. Dechter, "A new algorithm for sampling csp solutions uniformly at random," in *Principles and Practice of Constraint Programming-CP 2006*. Springer, 2006, pp. 711–715.

[9] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," *in Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2007, pp. 258–265.

[10] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, p. 13, 2007.

[11] M. Teplitsky, A. Metodi, and R. Naveh, "On coverage and debug in random stimulus generation," in *14th International Workshop on Microprocessor Test and Verification*, 2013.

[12] G. Alexandron, V. Lagoon, R. Naveh, and A. Rich, "Gendebugger: An explanation-based constraint debugger," *2010 TRICS*, p. 35, 2010.

[13] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on Empirical Software*. IEEE, 2009, pp. 291–301.

[14] N. Johnson, "Constrained random or directed? it's not an either/or choice for verification," http://chipdesignmag.com/display.php?articleId=4913.

[15] Arrow-Devices, "How to balance directed and constraint random testing," http://www.arrowdevices.com/blog/how-to-balance-directed-andconstraint-random-testing/ .

[16] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of Design Automation Conference, 2003.* IEEE, 2003, pp. 286–291.

[17] R. Naveh and A. Metodi, "Beyond feasibility: Cp usage in constrained random functional hardware verification," in *Principles and Practice of Constra int Programming*. Springer, 2013, pp. 823–831.