2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Who takes the driver seat for ISO 26262 and DO 254 verification?

Reconciling coverage driven verification with requirement based verification.

Avidan Efody, Mentor Graphics, Corp., Herzliya, Israel (avidan_efody@mentor.com)

***Abstract*— Verification teams working within an ISO 26262 or DO 254 flow have to reconcile the safety standard demands for Requirement Based Verification (RBV) with advanced verification techniques such as Coverage-Driven/Constrained-Random verification (CD/CR). The relationship between these two concepts has so far not been clearly defined or mapped, but some earlier publications that contrast the two, suggest that it is seen as one of either/or[1], or at least as one that is problematic and requires special attention[2]. This paper looks at the way these two approaches interact in greater detail in order to map the problematic places and see if and how they can be addressed. We expect it to help teams that struggle to get an ISO 26262 or DO 254 certification with advanced verification methodologies such as CD/CR.**

Keywords— requirement based verification, coverage driven verification, ISO 26262, DO 254

I. UNIQUE CHALLENGES

Automotive/avionics RnD teams required to follow ISO 26262/DO 254 face a set of unique challenges when trying to adopt Coverage-Driven/Constrained-Random (CD/CR) techniques. The flow of Requirement Based Verification (RBV) if carried out in a certain way, can lead to CD/CR techniques becoming too limited to be efficient. The methodology of CD/CR verification can make test results sensitive to verification environment modifications if not planned properly upfront. Finally, Tool and language limitations lead to gaps in requirement tracking from specifications and plans to signal level test results. In this paper we first describe the above mentioned problems in detail, and then suggest how they could be overcome, hence allowing for better and wider adoption of these proven verification techniques in the automotive and mil-aero industries.

To underline the uniqueness of the challenges, in various sections of this paper, we compare a "typical" CD/CR flow to a "typical" directed testing flow, where some of the challenges are either a non-issue or perceived as such. While doing so, we obviously make generalizations that will be in some cases inaccurate, and also ignore any shades that exist in between directed testing and CD/CR verification, such as directed testing teams that are making use of various forms of functional coverage.

A. RBV overrides CD/CR

1) The problem

The power of CD/CR verification techniques, lies, to a large extent, in their ability to generate specification compliant scenarios and combinations that RTL designers have not considered or planned for. The chances of generating such interesting test cases are in direct proportion to the solution space that the CR generator is allowed to choose from. RBV testbenches tend, on the other hand, to have a test per requirement, hence significantly limiting the allowed solution space, and reducing the power of CD/CR verification to find problems in the cross-sections between requirements.

2) Discussion

In a typical project that uses the CD/CR verification approach, separate design and verification teams get the specification for the DUT (Device Under Test), and interpret it independently of each other to create the RTL (Register Transfer Level) for the design, and the verification environment. The verification environment might further contain different independent interpretations of the specification – a verification plan, which aims to capture all aspects of the block that need be verified, and is later translated into a coverage model, a constrained random generator which should model all allowed inputs to the DUT (including illegal inputs that the DUT

should know how to handle), and a high level reference model or scoreboard used for checking. These representations of the specification and the links between them are shown in Figure 1. The main indication that all specification requirements have been met is achieved by running all the models together and making sure that they all overlap. If they do, then it is safe to assume that the specification has been interpreted and implemented correctly. However, it is often difficult to track a specific line in the specification document to any part in the RTL or testbench implementation except for the coverage model. Understanding the specification as a whole before diving into the details is a key to CD/CR efficiency, as it allows for the common parts of various features to be identified and handled together.

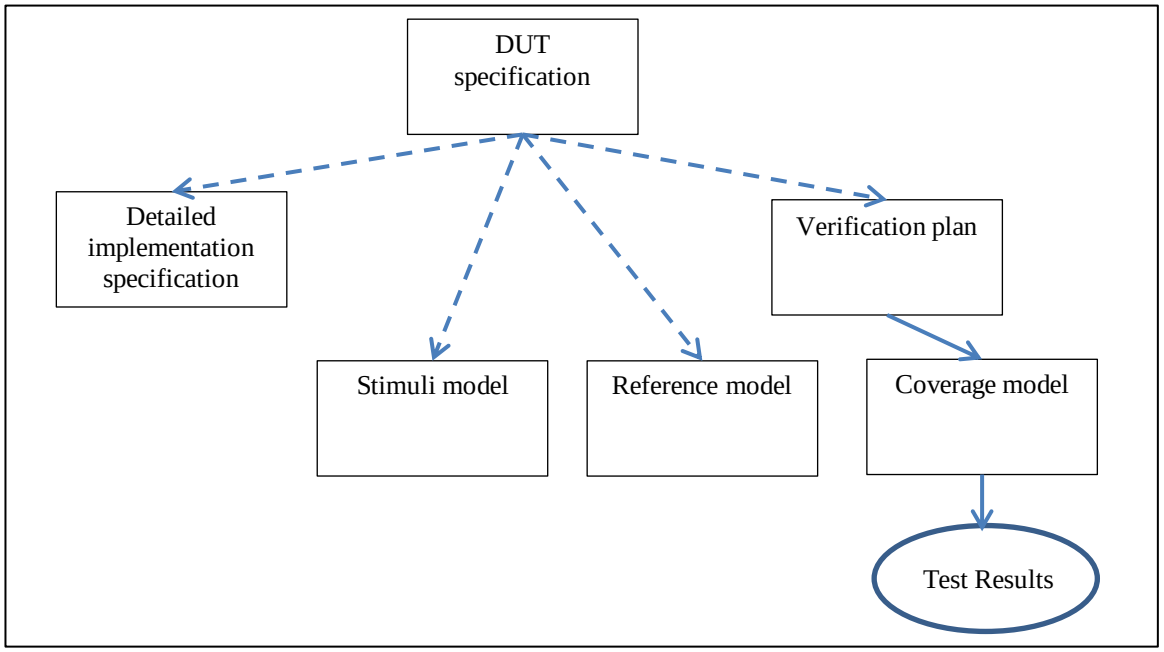

Figure 1: DUT specification interpretation tree in a typical CD/CR verification environment. Broken arrows denote few links to high level elements, while continuous arrows denote a multitude of links to low level elements. Functional coverage is the main type of test results collected

With requirement based verification, the document structure described above can be similar, but the links between the various documents and models would be much stronger. The specification would have specific sections that are marked and uniquely identified as requirements, and these would be then linked into the verification plan document and functional coverage elements. On the design side, they might be linked into an implementation document, and into parts of the actual code that implement certain features. Unlike the CD/CR verification flow, the confidence that the implementation is correct isn't derived by making sure that multiple independent interpretations overlap, but rather from the fact that all requirements outlined by the specification have been implemented by both design and verification team. This could in some cases lead to requirements being tracked from the verification plan into elements such as the stimuli or reference models, as shown in Figure 2, hence limiting their degree of freedom. If the links into the stimuli model are strong enough, the stimuli model might end up turning into a list of directed test, each of which targeting a specific detailed requirements.

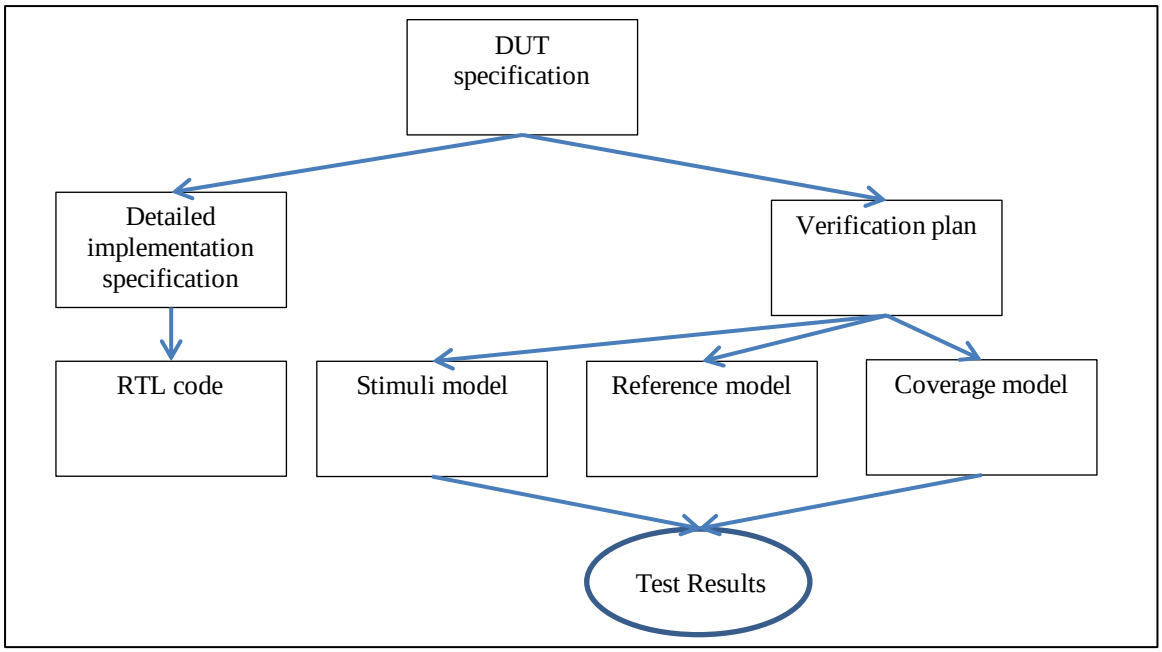

Figure 2: Possible DUT specification interpretation tree in an RBV environment. Requirements are sometimes tracked into RTL code as well as stimuli and reference model, hence turning the stimuli model into a test list. Test results might not be only functional coverage but also log/wave files of directed tests.

While from a requirement based flow stand-point it would be optimal if all results and code could be mapped one to one to requirements, in practice this is not always beneficial. Functional coverage can have good correlation to individual requirements, since its native use model is to be linked to verification plans, and covergroups and coverpoints are orthogonal analysis elements that don't influence similar elements. RTL code, on the other hand, will usually be much harder to map to individual requirements, since during the design process groups of requirements will normally be split and merged in order to achieve an optimized implementation that supports the whole specification. Quality design must take into account the commonalities between requirements and where they allow resources to be optimized and shared. The exact same applies to stimuli generators and checkers: Building a stimuli generator or a checker that address every requirement separately would be costly and result in repetitive code that is difficult to debug and maintain. Generally speaking while the requirement based approach aims to split the specification into manageable parts in order to simplify tracking, some implementations try to consider it as a whole in order to reach the best optimization possible. To make sure that a CD/CR approach remains efficient even within an RBV flow, the right balance between these two conflicting needs should be carefully maintained.

3) Example

Consider a DUT that is required to support a few versions of a given protocol, for example, a DMA (Direct Memory Access) controller that must support a few common ARM™ AMBA™ AXI™[3] versions (i.e. AXI3™, AXI4™, AXI4Lite™, ACE™) according to a given configuration parameter. Such a requirement is very common for AMBA and for other protocols, since it allows a given block or an entire SoC to be used in a wider range of systems. The specification for the interface part of this DUT is likely to be made of the specifications for the supported standards (AXI3, AXI4, AXI4Lite, ACE) augmented by an integration guide that explains how interface modes are switched, their parameterization, implementation of optional features etc.

In such a project, a very common first step for RTL design and verification engineers, would be to gain a good understanding of the entire protocol suite, and then map the commonalities between the various protocols that need to be supported to make sure that code is shared and reused wherever possible. Failing to do so on the design side would result in wasteful redundant implementation, and hence in more bugs. On the stimuli generation side the result might be a bloated test suite where different tests for practically identical features in multiple protocol versions exist. This can lead not only to longer regression runs, but also to coverage holes and missed bugs. On the checker side, similar to the RTL side, the code might not reach a stable state, because bug

fixes will have to be repeated over and over again. This in turn might lead to low checker credibility and checker errors being ignored even when correct.

Because the various AMBA protocol versions specifications are not incremental (i.e. they describe protocol functionality that fully overlaps with that of earlier versions), Trying to establish a one to one mapping from the relevant specifications to tests, checks, or RTL code segments in such a case would push RTL design, stimuli generation and checker implementation in the undesirable direction of separate implementations for each protocol version. However, as mentioned earlier, trying to establish a one to one mapping from specification sections to functional coverage elements such a coverpoints and bins, is feasible and if done correctly, will not have any negative impacts. Features such as cross coverage which allow identical code to provide analysis for several distinct modes, were designed to serve exactly that purpose.

4) Solutions

Awareness to the contradictory needs of requirement tracking and CD/CR effectiveness is the first step towards a solution. Once the trade-offs are understood, a variety of means can be taken to insure that the right balance for a given project is found and maintained.

a) Link requirements to functional coverage only

As mentioned above trying to create a one to one mapping from requirements to stimuli breaks CR generation into small chunks that prevent it from being optimized and reduce its capability of finding bugs that lay in the cross-sections of requirements. To avoid such a situation it is recommended that requirements in the verification plan are always linked to coverage, and as little as possible to stimuli. The sections that discuss the verification plan document in the ISO 26262 and DO 254 specification support this option.

Within ISO 26262 the main reference is section 8-9.4.2.1[4] which states:

> The verification specification shall select and specify the methods to be used for the verification, and shall include:
>
> a) review or analysis checklists; or
>
> b) simulation scenarios; or
>
> c) test cases, test data and test objects.

For the purpose of this discussion b) is the most relevant since a) refers mainly to the planning phase and c) to post-silicon testing. Linking of the "verification specification" to "simulation scenarios" might be done by linking it to actual tests driving these scenarios, but it can also be done by linking it to coverage points that are filled when these "simulation scenarios" are identified within the regression. The latter method is not only better from a CD/CR efficiency perspective, but also because it links requirements to actual test results, and not to the test's intended results.

Within DO 254 section 10.1.4[5] describes verification plan contents and makes the following requirement about the link between verification plan and verification data.

> verification plan should include:
>
> Verification Data: Identification and description of the evidence to be produced as a result of the hardware verification process.

This guideline could be met, by linking the verification plan to coverage results alone, and doesn't require any linking to the stimuli model.

b) Improve tracking from coverage to results

This is the flip side of the previous solution. When coverage collected can't be directly linked into signal level test results such as log files and the actual times in which events happened, users will fall-back to linking requirements to test results via actual tests. This will, in turn, lead to the situation of one test per requirement, which, as discussed above, tends to reduce CD/CR effectiveness. This paper discusses tracking from coverage to results and how it could be improved in section 1.C.

c) Strengthen verification team position

The experience of the verification team in CD/CR methodologies can play a big role in creating a healthy balance between requirement tracking and CD/CR needs. An experienced verification team will identify scoping issues and reduced efficiency earlier, and will be able to better communicate to management the measures required to rectify those. Confidence, derived from experience, will assure that the verification team input is taken into account during verification flow design, and doesn't end up imposing restrictions that would end up reducing CD/CR efficiency.

B. *Result instability*

1) The problem

Achieving ISO 26262 and DO 254 certification implies a repeatable and reproducible process. When the verification process is reproducible, it is simple to know which requirements were tested with each version, auditing becomes straightforward, and any bugs discovered after delivery can be analyzed against the regression to determine liability and understand why they remained undetected. Since constrained random testing is often perceived as unstable and non-deterministic, teams that have to follow ISO 26262 and DO 254 flows are sometimes encouraged to adopt methodologies that offer a more predictable link between results and the tests that are being run.

2) Discussion

To understand if this concern is a valid one, and to be able to address it if it is, we need to take a deeper look at the reasons that might cause a CR testbench to change its results and compare those against those that might impact alternative methodologies such as directed testing. First we need to remember that as long as the code for the test, the testbench and DUT, the simulation seed, and the simulator itself (i.e. same version) remain constant, the results produced by a CR testbench will remain the exact same for any given run. Cases where this rule doesn't hold are by definition a simulator bug. Such bugs exist of course, but with SystemVerilog CR generation being supported for more than 10 years on most major simulators, are not very likely[6]. However, to minimize the chances of tool bug it is recommended to use only CR features that have been around for some time (for example regular constraints, and not soft constraints[1]), and to emphasize that aspect during tool qualification (or ask the vendor to do it). Tool bugs aside, as long as the four above mentioned items are kept constant, CR results should be as stable as those of any directed test, the only difference being that a test/seed will need to be maintained per requirement, as opposed to test only with directed testing.

Since the list of tests and matching seeds to achieve a given coverage target is not available for the first regression run, the common flow is to first run a "blind" regression with a big number of different seeds per test. Once the results are available, they can be analyzed to extract the minimal list of tests/seeds that hit all functional coverage, hence removing any redundant tests that are not contributing to overall coverage. This list can then be saved, as the regression list for a given version, and each test/seed in it will be guaranteed to cover the exact same requirements as long as test and testbench code, seed, and simulator version remain the same. Since the "blind" regression run and the extraction of a test/seed list can be time consuming processes it is preferable to run "blind" regressions as infrequently as possible. Figure 3 describes this flow.

[1] Soft constraints were introduced only in the 2012[7] SystemVerilog standard LRM and are therefore a relatively new feature compared to regular constraints introduced at 2005[6].

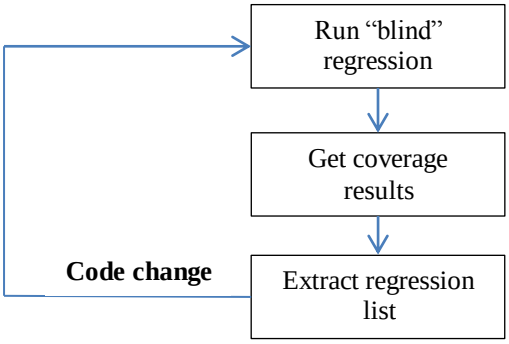

Figure 3: Extracting a minimal regression list after code change

Of course, during the life time of a project, the code is guaranteed to change quite often. In general, any changes made to the common parts of a CR environment, or to the common parts of a directed testing environment, can impact all test results and make them invalid. However, with a CR environment that is not designed properly, the impact is likely to be much more felt, because code changes in one part of the testbench might impact stimuli generation in parts that are seemingly unrelated. This happens because random number generation management is done for the testbench as a whole, and ignores functional or hierarchical dividing lines.

3) example

This example demonstrates how minor modifications in CR testbenches could produce big impact on results, functional coverage and requirement coverage. Consider an AMBA AXI4 wrapping burst address generator. Such address generator should generate addresses that match the following requirement (slightly rephrased from the specification):

> The start address of a wrapping burst must be aligned to the size of each transfer. Allowed transfer sizes are 1, 2, 4, 8, 16, 32, 64 or 128 bytes

The matching functional coverage for this requirement would cover the allowed size values on the bus. Assuming bus size of 1024 bits, all 8 size values are allowed. If the size value has uniform distribution coverage closure should be achieved within 22 transactions on average[2]. If distribution is not uniform, which will usually be true for cases similar to the above example[3], the number of transactions to achieve full coverage could be much larger. Once a “blind” regression has been run once, however, regression analysis will provide exactly 8 seeds that cover all legal size values.

As mentioned above, this optimal regression set will hold only until a modification that impacts random number generation happens. For example, if a new instance or a new thread are added into the generator, they might impact address and size values for the 8 seeds in the optimal regression list. If this happens the functional coverage is expected to drop to about 40% (i.e 8/22), forcing a rerun of “blind” regression to find a new set of optimal seeds. From RBV perspective it means that a requirement that has been covered is not covered anymore.

With a real life testbench, significant fluctuations in functional coverage mapped to requirements, not only lead to additional “blind” regressions that waste time and resources, but can also lower the credibility of a CR testbench from a management perspective, hence triggering a push towards methodologies that seem to provide better result guarantee. It is therefore important that adequate measures are taken to lower result sensitivity to code modifications.

[2] https://en.wikipedia.org/wiki/Coupon_collector%27s_problem

[3] Smaller size values have a wider range of address values to choose from, and therefore a higher likelihood of being selected. SystemVerilog distribution constraints could be added to compensate for this distortion, but this is rarely done, since it is usually not trivial to calculate the distribution of values for random variables.

4) solutions

a) Designing a random stable environment

To reduce the impact of code changes and limit its scope the concept of **random stability** must be understood, and the environment must be planned to be random stable, that is, to be planned so that any changes in code can influence random results only locally and to the minimal extent possible. If done correctly, this can make a CR environment as stable as one where no randomness is used. Planning an environment to be random stable is outside the scope of this paper, but has been discussed at length elsewhere [8]. One of the main things that can help in building such an environment is the use of a class library that provides the infrastructure for random stability behind the scenes. UVM (Universal Verification Methodology)[9], especially in later versions (above 1.1d) is one such class library. It basically makes all component threads and sequence threads in the testbench environment independent of each other, so that a modification of any single thread, does not affect the others. This can make some modifications have minimal or no impact on overall coverage, and reduce the number of "blind" regressions that need to be run to a considerable extent.

b) Continuous monitoring of coverage

Even with a well-planned random stable environment, there might be cases where, after a given modification, some requirements that were covered earlier, remain uncovered. To get an early warning of such cases, it is a good practice to keep a trace of all coverage reports, and to diff and analyze differences on a regular basis. An even better solution is to keep the coverage report trace linked to a code repository trace, so that any changes in coverage are easily linked to the respective code changes that triggered them. Advanced coverage and requirement databases allow users to perform trend analysis in order to detect such issues as well as a variety of other issues, and to plan their verification better.

c) Alternative generation methods

Over the last decade stimuli generation methods that combine the advantages of CR with those of directed testing have evolved. One such method is graph based generation, which lets users define, manage and prioritize an extensive suite of directed tests, while preserving the capability of generating random scenarios. Since these methods are external to the testbench they are not impacted by testbench modifications and guaranteed to provide stable stimuli [10].

C. Functional coverage tracking gap

a) The problem

ISO 26262 and DO 254 standards mandate that requirements are linked to verification results[4]. As shown in Figure 1 above in a CD/CR flow, the main verification results produced are functional coverage reports. Since raw information coming out of the design goes through some processing in order to turn it into functional coverage, users might be required to show that this processing is in fact reliable by linking from functional coverage collected to low level signal values and events stored in various simulation log files. Due to SystemVerilog language limitations users often struggle with this linking.

2) Discussion

[4] According to DO 254 10.4.1 "A correlation between the requirements, including derived hardware requirements, and detailed design data and the verification procedures and results" has to be established. According ISO 26262-8-9.4.3.2 "The evaluation of the verification results shall contain the following information: … the reference to the verification plan and verification specification"

RTL verification tools usually support three types of coverage: code coverage, assertion coverage, and functional coverage. All of these types of coverage could be used to drive verification, and are normally used to some extent in any CD/CR verification project. Each of them is best suited for a different type of application as shown in table 1. Code coverage is automatically generated, and doesn't capture requirements or design intent. Hence, it doesn't need to be linked to any test results that prove that a requirement or design intent has been tested. Assertion coverage captures requirements and in most tools is strongly linked to actual test results, that is, to the exact time in which a group of signals behaved in a certain way. This makes it a powerful tool in an ISO 26262 and DO 254 flow. However, it doesn't extend well beyond the signal level, and practically can't be used to cover complex data relationships. Functional coverage is best at covering the internal relations in individual data items (i.e a single transaction), but gets harder to code as soon as sequential behavior over time needs to be covered, since all items that need to be covered must exist at the time when a covergroup is sampled. Though it can usually be linked to specific tests, it stores no timing information, and since it brings all items to a single point in time for sampling, by definition it can't link to sequences of events. Hence from a DO 254 and ISO 26262 perspective it is lacking.

Type of coverage	Best for:	Links to test results
Code coverage	State machines, arbitration schemes, FIFOs and queues, etc.	Doesn't capture design intent Doesn't need to be linked
Assertion coverage	State machines, arbitration schemes, FIFOs and queues,	Strong link to test results Only practical for signal level
Functional coverage	Handling transactions: requests and responses, packets, CPU instructions etc.	Weak link to test results Impractical for complex sequential behavior

Table 1: Types of coverage, their application areas and how they link to actual results

3) Example

Consider a AMBA AXI4 bus as discussed in earlier examples. To show that a given requirement has been implemented, for example that all read accesses to a secure zone get an ok response with garbage data, a user might implement a covergroup that crosses given request phase parameters, such as address, with given response phase parameters such as response type and data. The point at which the covergroup that contains the cross coverage item will be sampled, will usually be the time at which the response arrives. Such coverage is too complicated to implement using assertions, so it is likely that SystemVerilog functional coverage will be used.

Now assume that to prove the correctness of the functional coverage collected, the user is required to show that the functional coverage report matches a simulation log file that contains the actual signals values on the AXI4 interface. This might present the following difficulties:

1. The time at which the covergroup is sampled is not logged[5].
2. If coverage is collected at response arrival, the time at which the request occurred is not logged.
3. Links to relevant signals are not logged.

Since none of the above information is available via the covergroup itself, users might need to log "out of band" information and use some additional tools to allow for quick linking from functional coverage to signal level results.

4) solutions

a) Logging of "out of band" information

As mentioned above, to solve this problem within traditional SystemVerilog testbenches it is possible to log some "out of band" information. For example advanced coverage databases might implement a non-standard feature which logs the simulation times in which a given covergroup bin is hit, hence providing an initial link into simulation log files. Using UVM's transaction recording and linking, this point in time can be used as a key to

[5] Some simulators might log this information but this is vendor specific and not required by SystemVerilog specification

obtain all phases of a given transaction. Off the shelf verification IPs that have built in support for recording and linking, might make the task of linking from functional coverage to signal level easier.

b) Dynamic/Statistical coverage tools

The ideal coverage tool that would allow bridging the "last mile" into an actual point in a test should be able to store multiple events, just like assertion coverage, and multiple data items. If a requirement can be specified as a chain of events and associated data items leading to another chain of events and associated data items, i.e. :

dataIn1@eventIn1 + ***dataIn2@eventIn2*** + .. + ***dataInN@eventInN*** -> ***dataOut1@eventOut1*** + ***dataOut2@eventOut2*** + ... + ***dataOutN@eventOutN***

Then the coverage tool must be able to capture all of these data items and events. Such a tool would allow users to trace any requirement to a chain of events and associated data items within a test.

The kind of linking described above is not possible with SystemVerilog or with any language/tool that are widely in use today, but there are some efforts to make it happen. Some vendors are offering system level analysis tools that log large amounts of regression signal level information into a database which can then be queried by users for relations between events in the time domain and data domain[11]. If requirements could be coded into queries, users could then go from the query directly into the corresponding test results. Apart from better linking, such a tool could also shorten turn-around time for getting new coverage. If a new coverage item is needed because a new requirement is added, the existing data base could be queried, to see if the requirement is already met. With existing technologies based on SystemVerilog covergroups, the whole regression would need to be rerun to get data for the new coverage item.

II. SUMMARY

With ISO 26262 and DO 254 gaining momentum more and more verification teams are expected to adhere to a requirement based flow, while trying to remain as productive as possible using advanced verification techniques. As we have seen above combining the two is not always straightforward, and requires awareness of the specific challenges as well as good amount of planning upfront. However, if done successfully, it can enable teams to enjoy both worlds, and end up with a flow that is better than both its individual components. Improvement requirements coming from these teams will likely drive EDA vendors as well to enhance their tools, for example, by providing better coverage tracking capabilities and new types of coverage that supersede those that exist in SystemVerilog.

III. REFERENCES

[1] Serrie-Justine C., Galpin D., Bartley M. "Requirements Driven Verification Methodology (for Standards Compliance)", DVCon-Europe 2014
[2] Serrie-Justine C., Bartley M. "Requirement Driven Verification Methodology Tutorial" DVCon-US 2015
[3] AXI4 standard
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0022/t_self/oAMBA%204%20AXI%20&%20ACE%20protocol%20specification
[4] ISO 26262-1:2011 Road Vehicles -- Functional safety
[5] RTCA/DO-254:2000 Design Assurance Guidance for Airborne Electronic Hardware"
[6] IEEE Standard for System Verilog- Unified Hardware, Design,Specification and Verificaction Language", IEEE std 1800-2005, 2005.
[7] IEEE Standard for System Verilog- Unified Hardware, Design,Specification and Verificaction Language", IEEE std 1800-2012, 2012.
[8] Efody, A. "UVM Random Stability : Don't Leave it to Chance" DVCon-US 2012
[9] UVM standard http://accellera.org/downloads/standards/uvm
[10] http://www.accellera.org/activities/working-groups/portable-stimulus
[11] Hunter A., Meyer A., Fredieu R. "Understanding the Effectiveness of Your System-Level SoC Stimulus Suite", DVCon-Europe 2015