Blending multiple metrics from multiple verification engines for improved productivity

Darron May
Design Verification Technology Division
Mentor Graphics Inc
Newbury, UK
darron_may@mentor.com

Darren Galpin
System IP Verification Manager
Infineon Technologies
Bristol, UK
darren.galpin@infineon.com

Abstract — **Combining metrics from multiple verification engines can massively improve productivity by reducing unnecessary verification time.**

I. INTRODUCTION

Vast amounts of data are produced from today's verification environments. As a result, there is a real need for solutions that deliver capacity and performance to access and analyze this data in a timely manner. No one coverage metric can be used to measure functional verification completeness. There is a two-fold requirement: to unify both the different coverage metrics and the data from multiple tools and verification engines.

Indeed, data management forms the foundation of any verification environment. This paper will show how Infineon leveraged Mentor's UCDB (Unified Coverage DataBase) API to store multiple metrics from different sources and tools in a common database. We will discuss the custom analysis that Infineon performed that was possible as a result of having all the coverage metrics stored in a single format that can easily be accessed via an open API (Application Programming Interface).

II. DATA MANAGEMENT REQUIREMENTS

The reality of multiple tools, engines and metrics means the ideal verification database has to support more than just coverage. The database infrastructure must provide the visibility into the process across many dimensions. The major requirements of such a database are as follows:

A. Unification

No one coverage metric can measure completeness. The database has to allow the storage of a large mix of coverage metrics from many data sources including simulation, emulation, static formal analysis tools, software-driven tests and many other application-specific sources. It should be possible to combine data based on blocks, systems, instances, tests, users and time to give the most flexibility. Combining this data based on so many variables requires a flexible architecture and a capability to store details about how, where and when the coverage data was generated. This allows the verification engineer to determine how and when a particular metric was or wasn't hit. The process also should allow these metrics and measurements of certain system requirements to be associated with a verification plan, and ultimately the design specification.

B. Capacity and Performance

Unifying the verification data stored by all tools and metrics can result in huge volumes of data. The storage capacity must be able to handle the very largest of today's designs and the designs of the future. As the stored data increases, it is important to have an environment that is optimized for capacity and has the performance to manipulate and query potentially large amounts of data within workable limits. For example, such a database would be required to combine results from tests that have many millions of coverage bins. This can strain database capacity. Ideally a solution should have the ability to solve both the capacity and performance issues within the largest of projects now and in the future.

C. Visibility and Analysis

Querying stored verification data requires accessing the database. The results from many verification engine runs need to be combined and the verification engineer needs to be able to analyze which runs with which particular settings caused particular metrics to be hit. Such analysis is required to do optimization, such as figuring out redundancy in tests or isolating a particular test or set of tests of a particular feature. Combining or merging the data also necessitates an ability to query the database to find out information on the history of how the data was generated. This includes not only the command line options for generating the single tool runs but also the utilities used to add and combine data to the database across the progression of the project. Efforts to reduce and optimize data help reveal trends across the duration of the process. The verification process is dynamic. As functionality is added to the design and bugs are found and fixed, a higher level of trends helps determine if progress is being made towards completion.

D. Control

Beyond continual data analysis throughout the project, the Verification Engineer also needs to have control over the coverage model and the ability to document decisions made during the process. The database has to have the ability to manipulate coverage metrics into an overall metric showing the level of completion. It also should support tradeoffs, or specifically, the swapping of one metric for another based on importance defined by a user-controlled weighting system. As verification progresses it's also important to document any exclusions to the coverage model and the reasons why they have been excluded. These types of exclusions could be made automatically by the verification tools. An example is a static formal tool that excludes unreachable code ahead of dynamic simulation.

E. *Extensibility and Openness*

Finally, the database needs to be extensible and allow for the addition of new application-specific information or metrics, even those not currently known. An example is information or metrics from a tool yet to be developed. It also needs to be completely open and have the ability to add or remove any data with a clearly defined interface. This requirement allows any third-party tool to write data into the database or extract data, allowing the unification of data across tools from one or multiple vendors.

III. THE UNIFIED COVERAGE DATABASE

A. *Overview*

The Unified Coverage Database (UCDB) has been architected from the ground up to meet the requirements outlined above. The UCDB has been the default coverage database format for storing code coverage and functional coverage metrics in both ModelSim and the Questa Advanced Simulator for a number of years. In addition, UCDB extensibility allows for combining test data, assertions and coverage results from many other engines, including Static Formal Verification, power-aware simulation, Analog Simulation and emulation. In addition to its coverage storage abilities, the UCDB also stores verification plans and test-specific data, making it a solid anchor for any verification team that intends to adopt a verification methodology driven from verification plans, design and/or requirements specification documents. One of the biggest verification challenges is having the ability to bring together the data and benefits from multiple verification techniques. The UCDB merge algorithms have been developed to take into consideration data from both formal (static) and dynamic verification engines. It has the ability to combine results and report on any conflicts that may occur when comparing static and dynamic techniques, as well as allowing a static formal engine to exclude coverage from the dynamic simulation engine flagged as unreachable. Leveraging the unique test-associated merging capability it is possible for a verification team to maintain a single merged database that contains merged coverage data from multiple verification runs or simulations in a regression. A record of the attributes, commands and settings of any tool are associated with each test or testcase, giving it a unique label to allow test association with coverage data. The architecture allows verification plans to be imported and linked with multiple coverage metrics or tests. This single database has enough information within it to help figure out the test(s) that incremented a specific coverage bin.

B. *Architecture Details*

From the users point of view the database has three sections as shown below in Figure 1, three sections of the UCDB. The first

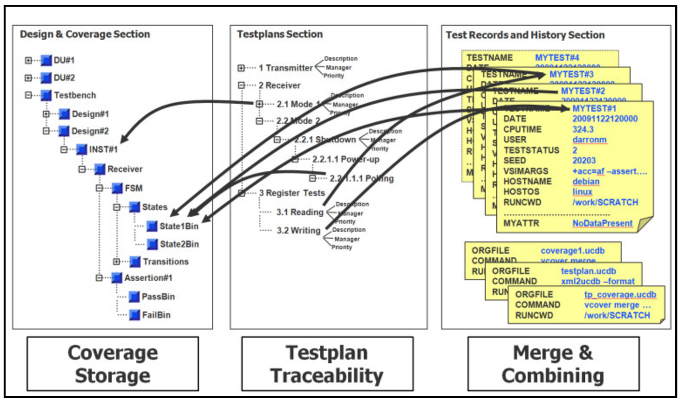


Figure 1, three sections of the UCDB

section is the coverage data collection. The second is the recording of test-specific information, such as test name, tool settings, CPU time, username, etc., along with the history of how the tests were generated, combined and/or merged together. The last is the testplan section used for testplan tracking, which allows the storage of testplan items that can be linked to the coverage model and/or the test cases themselves.

Designs and testbenches are hierarchically organized. Design units (Verilog modules or VHDL entity/architectures) can be hierarchical, though they are not always. Test plans can be hierarchical. Even coverage data (of which the SystemVerilog covergroup is the best example) can be hierarchical. Therefore, the UCDB needs some general way to store hierarchical structures. The UCDB has scopes (also referred to as hierarchical nodes), which store hierarchical structures (i.e., elements of a database that can have children). Coverage data and assertion data are stored as counters, which indicate how many times something happened in the design. In

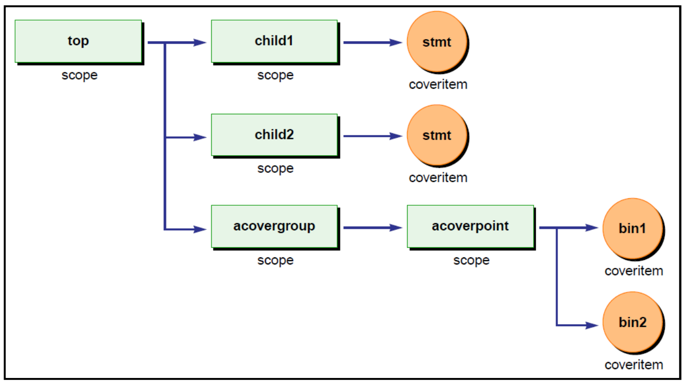


Figure 2, the basic design, coverage hierarchy

Figure 2, the basic design, coverage hierarchy is shown. For example, the counters count how many times a sequence completed, how many times a bin incremented or how many times a statement executed. In UCDB terminology, these types of counters and some associated data are called coveritems. These counters are database leaf nodes, which cannot have children. In the diagram the scopes top, child1, and child2 represent the module instances of this design hierarchy however there is also a need to access the data from the design unit point of view. Coverage associated with the design unit and accessed this way will be the union of coverage from the instances of the design unit. Source file information is stored with the design unit as attributes and from each module instance scope, its corresponding design unit may be accessed; in fact, the design unit must exist prior to creating any instances.

As the UCDB needs to distinguish between module instances, design units, and even other scopes like those for covergroups and coverpoints, the UCDB has a scope type associated with every scope. Scope types are in one of the following categories an HDL scope, a Design unit scope, a Cover scope, a Group scope or a Test plan scope. Each scope can hold extra attributes about the nature of the scope. Within the hierarchy of the scopes there are relationships that must exist between certain scope types for example HDL scopes where an instance scope must have a corresponding DU scope, and a generate scope must be within an instance scope. These relationships are defined within the HDL languages. The UCDB API is a very general one that creates certain objects – such as scopes, coveritems, and test data records – with certain names, types, and attributes. This allows creation of many different potential data models. The data models are important because they capture assumptions about the data structure for a given kind of coverage. Other tools might be able to read and make sense of different data structures, the UCDB API itself is more general purpose and many different kinds of coverage hierarchies could be created through the API.

C. *Coverage Data Model Examples*

The coverage metrics that were of interest in the Infineon design flow were functional coverage, assertions, and code coverage such as statement, branch, expression, and toggle coverage. The UCDB has data models for all these types of coverage allowing applications to be written that can access these types for both analysis and updating during the verification process. The following is an overview of two of the types of coverage that are covered within this paper and how they are stored within the UCDB so that a better understanding can be gained on some of the details in the latter part of this paper.

The covergroup type data model is part of the subtree rooted at the "cg" (UCDB_COVERGROUP) scope – specifically, the subtree containing the UCDB_COVERPOINT and UCDB_CROSS scope children. The covergroup instance is the subtree rooted at the UCDB_COVERINSTANCE node. It is a mirror of the type subtree. When there are multiple instances, the number of coverpoint and cross children must be the same among all instances, but the numbers of bins can be different depending on the definition of the covergroup options. The coveritems are at the lowest level and are the bins that store the counts for the number of times they have been hit. Any SystemVerilog covergroup can be modeled using the combination of scopes defined above. The diagram in Figure 3, covergroup data model shows an example of a covergroup implemented in scopes in the UCDB. In this example a top level instance scope is the parent to a covergroup scope called "cg', the covergroup scope has children scopes for each of the coverpoints and crosses and each of the coverpoints and crosses have coveritems for the bin storage.

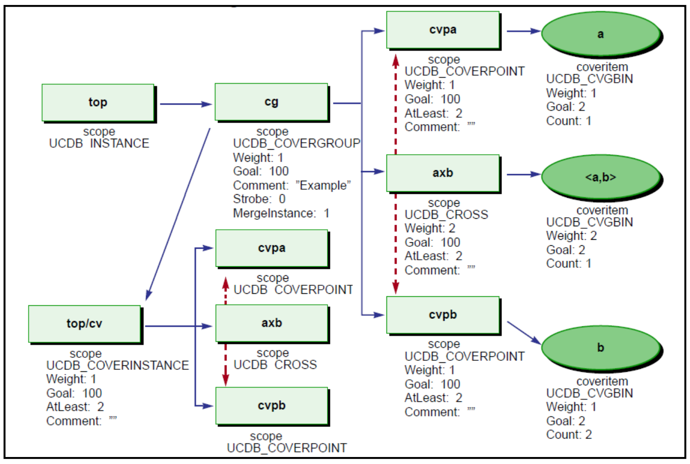


Figure 3, covergroup data model

If the covergroup has instances then a cover instance is added as a child of the covergroup and then the cover instance will have scopes that include the instance coverpoints and crosses. Certain attributes and flags are stored on the scopes and bins that give meaning to the scope, some of these are shown in the example such as weight and goal. One of the important flags is the exclusion flag which allows the coverage object to be excluded from coverage calculations.

The branch coverage data model decomposes the branching of an "if" or "case" statement into scopes that can be used to count the number of times a certain branch has been taken. The diagram in Figure 4, branch coverage data model shows an example of an "if", "elsif" VHDL branch statement.

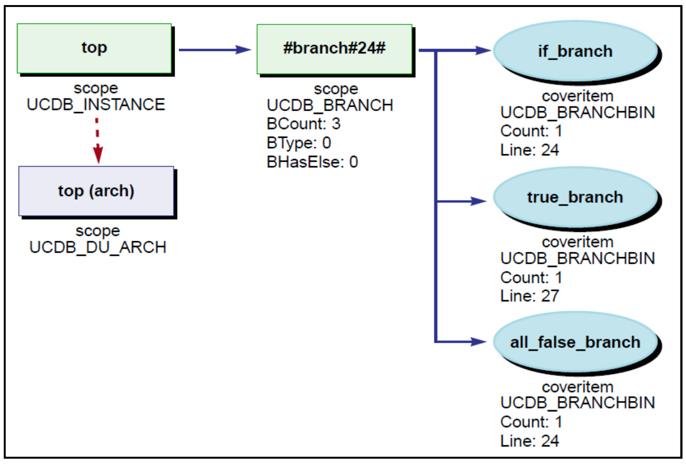


Figure 4, branch coverage data model

The HDL instance scope is parent to the branch scope that has a bin for the "if" statement being true, the "elseif" statement being true and lastly the all false branch when neither the "if" or "elsif" are true. Again the scopes have attributes and flags to carry certain information about both the source of the coverage and its uniqueness.

As has been mentioned the scope type within the database testplan sections can also be modeled. Shown in the diagram in Figure 6, the testplan model. The testplan has two sections, one and two, which could have been written in any format. Once they are imported into the UCDB they can be linked to any coverage that exists in the database by the use of the tagging mechanism. The testplan scopes and the coverage scopes are tagged with a matching string which makes a virtual link that can be used to calculate testplan coverage based upon the

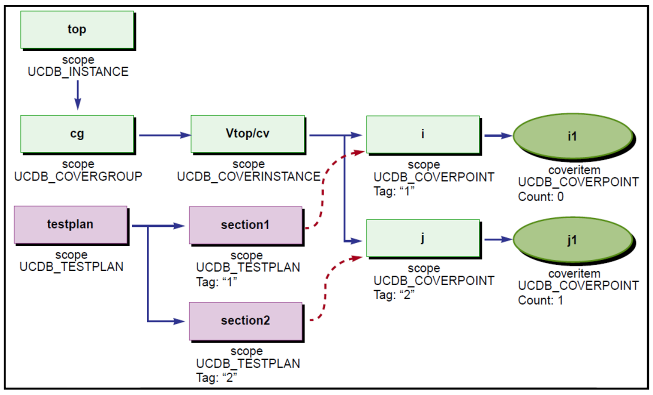


Figure 6, the testplan model

coverage that is attached. This coverage would be based upon the coverage that is needed to ensure a particular requirement or part of the testplan has been successfully verified.

The calculation of the coverage within the database is the roll-up of all the coverage bins within an instance. It is possible to define what coverage needs to be included in this roll-up if separate numbers are needed for each of the coverage metric. This also has the built-in feature that an overall coverage number can be gained from for the complete design or testplan hierarchy.

D. Standardization

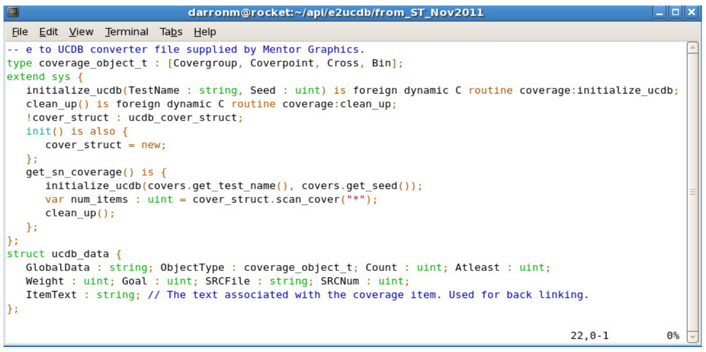
Soon after the first implementation of the UCDB was being stressed in its use on real projects, Accellera formed the UCIS (Unified Coverage Interoperability Standard) working group. The goal: to develop a standard for coverage interchange between vendors. The group is made up of both EDA vendors and user representatives from the largest companies in the industry. Mentor Graphics donated its technology as a starting point for the standard. This triggered other donations from other sources. The UCDB API was chosen as the basis of the standard after a lengthy period of analysis of the donated technologies. With the standardization process well under way, users will start to benefit from this pioneering work and database optimizations, particularly as other vendors introduce solutions based on the UCIS.

IV. Combining Functional Coverage

Infineon's verification environments involved a combination of VIP using both 'e' and SystemVerilog. This meant there was a need to combine functional coverage from both languages to allow analysis to be carried out on the complete coverage model. This was achieved by converting the 'e' functional coverage into the SystemVerilog equivalents and combining the two in the UCDB. The UCDB does have the ability to store the data model for 'e' coverage, but as the same analysis tools needed to be used for both types of functional coverage, the 'e'-to-SystemVerilog conversion made sense. This seems to go against the whole theory of a unified coverage database but it actually goes to the very heart of the problem of combining coverage metrics that are similar but not quite the same. It is easy to store data into a standard database when the model is known and understood, analysis tools are able to read this model and analyze based on the stored data. However there are two types of analysis tool, one is generic and can analyze any combination of coverage because it can combine the scores and bins of say SV functional coverage, statement, branch and other coverage metrics, for example a ranking algorithm or similar. The other type of analysis tool may be specific to the coverage metric itself for instance in this case of a SV functional coverage browser. This second case reads and expects to see coverage and attributes specific to the coverage type, like coverpoints, crosses, SV options etc. In our case with functional coverage in both SV and 'e' we wanted to use the analysis tools built for SV to analysis them together in the same place. Hence transforming the 'e' coverage to SV coverage model in the UCDB made complete sense. Storing 'e' data in the UCDB would be as simple as adding slightly different attributes to the storage bins.

The basic setup for extracting the 'e' coverage was to extend the 'e' coverage API to call C functions written using the UCDB API as foreign routines. These routines allowed the coverage within 'e' to be parsed and used to generate the equivalent functional coverage objects within a UCDB. This coverage was then combined with the SystemVerilog coverage that was generated natively by Questa.

There were three routines added to the 'e' coverage class. The first was an initialization routine called "initialize_ucdb,"


darronm@rocket:~/api/e2ucdb/from_ST_Nov2011
File Edit View Terminal Tabs Help

```
-- e to UCDB converter file supplied by Mentor Graphics.
type coverage_object_t : [Covergroup, Coverpoint, Cross, Bin];
extend sys {
   initialize_ucdb(TestName : string, Seed : uint) is foreign dynamic C routine coverage:initialize_ucdb;
   clean_up() is foreign dynamic C routine coverage:clean_up;
   !cover_struct : ucdb_cover_struct;
   init() is also {
      cover_struct = new;
   };
   get_sn_coverage() is {
      initialize_ucdb(covers.get_test_name(), covers.get_seed());
      var num_items : uint = cover_struct.scan_cover("*");
      clean_up();
   };
};
struct ucdb_data {
   GlobalData : string; ObjectType : coverage_object_t; Count : uint; Atleast : uint;
   Weight : uint; Goal : uint; SRCFile : string; SRCNum : uint;
   ItemText : string; // The text associated with the coverage item. Used for back linking.
};
                                                                      22,0-1        0%
```

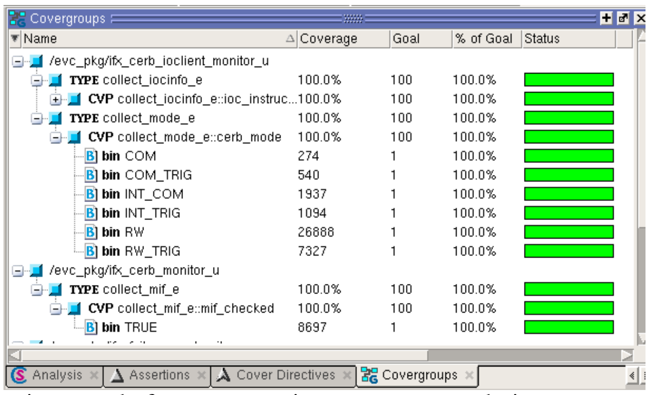
Figure 5, fragments of extended 'e' code

which is called under the constructor as a foreign function and in turn calls the UCDB API routine written to open a new UCDB and add the test record for the gathered coverage. The test record is used to hold as much information as possible about how the test was run. In this case, the seed and name of test were passed but this will be extended in the future to pass more information about the 'e' test. The second routine, "add_type_coverage_data," was used to transfer the detailed information about the coverpoints and bins within the 'e' coverage model. This routine is called on every coverage object as the 'e' coverage API transverses the coverage model. The routine calls the UCDB API with all the relevant information so that the equivalent SystemVerilog coverage can be written by the UCDB API routine that is called by the extended function. The complete instance name, the coverage counts, the at least values, weight, goal, testplan linking details and source code information are all passed with each call to this routine. Finally, the third routine, "clean_up," closes and writes the UCDB data that has been built up in memory to a persistent UCDB data file. In Figure 5, fragments of extended 'e' code can be seen..

The "add_type_coverage_data" routine written using the UCDB API does all the heavy lifting. It gets pasted to it the complete instance name within the 'e' coverage model that includes the class, the path to the coverage object and any instance information. Along with this it is passed the coverage object type, the counts, weights, goals and source code information. The UCDB API calls within this routine then add the objects to the UCDB in-memory following the data model that is defined earlier in Figure 3, covergroup data model. The format of the instance path information is as follows.

<ClassName>::<CoveragePath>(InstanceName==Value)

The routine is called repeatedly by the 'e' API and requires that covergroups are defined before coverpoints, crosses or cover instances, and that coverpoints and crosses are defined before bins. This results in ensuring that the cover group is in existence before coverpoints, crosses and instances, and coverpoints and crosses exist before the bins are stored. All the counts, weights, at least values and testplan linking information is also transferred into the UCDB. When running the testbench with both the 'e' VIP (Verification Intellectual Property) and the SystemVerilog VIP, a UCDB file containing the 'e' functional coverage written from the extended 'e' code can be combined or merged with a UCDB file containing the SystemVerilog coverage written by Questa. Standard product utilities were used to merge the two UCDBs together and with the test-associated merge of the UCDB, all of the analysis tools are able to query which coverage bin was hit by which test. This resulted in being able to tell if the coverage was hit by 'e' or SystemVerilog. The viewing of the 'e' coverage can be seen alongside the SystemVerilog coverage in Figure 7, converted 'e' coverage.



Name	Coverage	Goal	% of Goal	Status
/evc_pkg/ifx_cerb_ioclient_monitor_u				
TYPE collect_iocinfo_e	100.0%	100	100.0%	🟩
CVP collect_iocinfo_e::ioc_instruc...	100.0%	100	100.0%	🟩
TYPE collect_mode_e	100.0%	100	100.0%	🟩
CVP collect_mode_e::cerb_mode	100.0%	100	100.0%	🟩
bin COM	274	1	100.0%	🟩
bin COM_TRIG	540	1	100.0%	🟩
bin INT_COM	1937	1	100.0%	🟩
bin INT_TRIG	1094	1	100.0%	🟩
bin RW	26888	1	100.0%	🟩
bin RW_TRIG	7327	1	100.0%	🟩
/evc_pkg/ifx_cerb_monitor_u				
TYPE collect_mif_e	100.0%	100	100.0%	🟩
CVP collect_mif_e::mif_checked	100.0%	100	100.0%	🟩
bin TRUE	8697	1	100.0%	🟩

Figure 7, converted 'e' coverage

V. COMBINING FORMAL AND CODE COVERAGE

The following section shows how the results from simulation and formal were combined to allow for exclusion of certain metrics from simulation using the results of static analysis.

A. *Finding Dead Code*

Switching on code coverage within Questa will cause the enabled code coverage metrics to be stored within the same UCDB that stores the SystemVerilog coverage. Running the same source code on the formal tool, OneSpin in this case, allowed for generation of a dead code report. Using the output from the formal tool report, exclusion commands were generated to apply to the UCDB to exclude the unreachable statements and branches. This is achieved by generating a set of simulator commands, which apply the exclusions and automatically update the UCDB. The commands are in the following format:

coverage exclude –src <filename> -line <number>

The user currently sources this separately once the dead code checks have been run, but it would also be possible to modify the process which generates the excludes to again call a separate UCDB API application directly and apply the exclusions to the UCDB. The UCDB has an exclusion mechanism that is implemented as a flag on the coverage object within the data model. The coverage exclude command modifies this flag for the defined object. The API has direct access to all the information within the UCDB, so the code could be modified to set this flag for the object instead of outputting exclusion commands to run as a separate process.

B. *Unreachable Expressions*

Focused Expression Coverage (FEC) is a metric that details which inputs have been toggled causing the output of the expression to change. To gain full FEC coverage, every input must have toggled with the other inputs held at a level that allows the toggled input to cause the output of the expression to change state. After dynamic simulation is run, missing FEC coverage can be seen within the UCDB because each input toggle has a bin that is incremented on the occurrence of a pattern that satisfies the occurrence of the output toggle. The patterns themselves are also stored within the UCDB FEC model as attributes, so the vector required to hit a particular FEC point as well as the terms that this refers to allows inductive properties to be created to prove whether the FEC point can actually be hit, i.e:

assume: at t: not(<vector and expression to be hit>)

prove: at t+1: not(<vector and expression to be hit>)

This then gives you a list of FEC points which can never be hit. Those that can be hit formally may require an illegal state combination or input signal combination to get there, so they can always be rerun with added restrictions on the design inputs. The results of these can then be fed back into the UCDB to filter the unhittable points by again excluding them within the UCDB using exclusion commands or setting the exclusion flag.

C. Stuck at signals

OneSpin also reports which signals toggle during analysis and produces a log file of these results. The file can be converted into PSL assertions within the UCDB, stating which stuck-at, initialization and dead code checks pass and fail. The user can then iterate over the failing stuck-at PSL assertions and generate a set of exclusions, which can be applied to the structural toggle coverage already within the UCDB, excluding the signals which cannot be toggled. Currently this is done manually within Infineon, but it is possible to automate this step.

PSL assertions were used rather than tool specific data-types in order to aid reuse of the resultant UCDB between multiple vendors and their tools. PSL assertions are a commonly used construct between simulators and formal tools, and any tool providing coverage of them will understand the results. This then allows the user to utilize any specific features within a coverage tool to analyze the resulting data set, rather than relying on a given tool understanding a vendor specific data construct.

The first step is to convert the log file from the OneSpin analysis which gave stuck-at, initialization and dead code checks into PSL assertions written within the UCDB again using the UCDB API. Now, when trawling over the toggle coverage structural results it's possible to generate exclusion commands to exclude stuck-at signals proven by the formal tool.

The conversion from OneSpin log files to UCDB is done via a scripting file which calls a C object to create the resultant UCDB. Each of the different check types is grepped out of the resulting log file, and the name of each check together with its pass/fail result are added into an array. The resulting array is then iterated over, and each entry is then passed to the UCDB API as a PSL assertion with the associated formal result and the information stored. This process is then repeated for each check type.

Running these checks and this UCDB creation before simulation is started can also reduce the amount of simulation required. By removing code that cannot possibly be hit from the set of coverage to be hit before simulation starts, the user can reduce the number of regression cycles run trying to hit things which are unachievable, and also reduce the time spent manually trying to justify why certain structural coverage points cannot be hit. The latter is often a time sink just to justify why coverage points cannot be hit, complementing the static analysis of formal and excluding this coverage in dynamic simulation can be very productive.

D. Formal property results

Finally, to get a complete picture of the verification result in the UCDB, we took a log file from OneSpin detailing the properties run and the pass/fail/vacuous results, converting these into PSL property results and storing them within the UCDB. The log file from OneSpin details the property results in a tabular format as follows:

ITL Object	type	result	validity
Basic_read	property	pass	uptodate
Basic_write	property	fail (1)	uptodate

This log file was read into an array, and then iterated around, with each property name becoming a PSL property object. The result was added to the property object sub-field. Again, PSL property objects were used to maintain the commonality between tool sets, and in this case, VHDL PSL properties were used to maintain commonality between the DUT language and the other verification approaches. Note that the actual properties were written in the proprietary ITL language within OneSpin, although the approach would also work should SystemVerilog Assertions (SVA) or PSL properties be used within the OneSpin environment.

Although properties are not directly equivalent to structural coverage, they are analogous to functional coverage, in that they are testing functions within the actual DUT. If a property within the UCDB is passing, it should be possible to link this to functional coverage defined elsewhere within the UCDB and thus reduce the amount of required simulation. (Currently this is done manually.)

VI. DATA ANALYSIS

Having all metrics stored within a single UCDB results in a single source, which gives the ability to analyze data using a number of standard utilities. One example: applying a new ranking algorithm across all metrics to get a complete view of the data for all tests and all metrics. The UCDB stores the name of the test, which hits each structural and functional coverage point, and each added property can be equally treated as a test, as can the PSL assertions added from the formal consistency checking. All of this information can be viewed as a whole for analysis within the same tool. As the information is now available in the one location, we can run the inbuilt ranking tool to obtain a list of tests which hit the coverage metrics in the shortest possible time. The only difference now is that the range of coverage metrics is larger. Previously, this would have only been done for the structural coverage within Questa, or for structural and functional coverage if using SystemVerilog. The resultant output is a list of tests (including assertions and properties) and the total of the various coverage metrics that they have achieved as a regression suite. This then forms the golden regression for the test environment.

Example output from UCDB ranking:.

Metric	Items	Covered%
Statements	617	70.82
Branches	421	57.00
Expressions	334	81.13

```
Conditions          365            33.69
ToggleNodes         4234           42.18
AssertPasses        2708           100.00
FecExpressions      444            61.03
FecConditions       524            33.20
```

All 4 input files have been ranked.

Ranking summary:

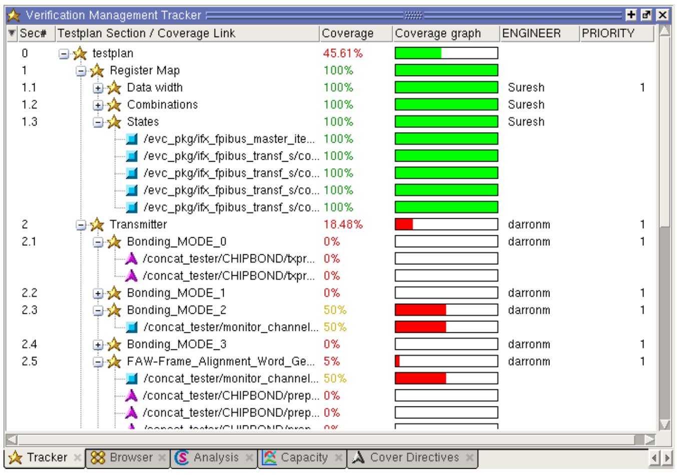
Total CPU time = 196.97

Total SIM time = 464920.00 ns

Test order = onespin_property.ucdb, testcase.e_1106318176.ucdb, testcase.e_102558471.ucdb, testcase.e_1106318190.ucdb

In the above example, the PSL assertions generated by the formal consistency check have been treated as coming from one test, as only one tool run is done to generate them. Thus the reason why only one test is in the test list while there are 2708 assertion items in the ranking report.

With all the coverage stored within a single source it is possible to link any coverage object or testcase to a testplan for traceability reporting. The UCDB has the ability to import testplans from all popular editing tools, such as Word, Excel, Calc, Framemaker or even requirements capture systems such as DOORs or ReqTracer. The testplan defines a particular requirement be tested with the combination of a coverpoint, cross, cover directive or any other coverage metric stored



Sec#	Testplan Section / Coverage Link	Coverage	Coverage graph	ENGINEER	PRIORITY
0	testplan	45.61%			
1	Register Map	100%			
1.1	Data width	100%		Suresh	1
1.2	Combinations	100%		Suresh	
1.3	States	100%		Suresh	
	/evc_pkg/ifx_fpibus_master_ite...	100%			
	/evc_pkg/ifx_fpibus_transf_s/co...	100%			
	/evc_pkg/ifx_fpibus_transf_s/co...	100%			
	/evc_pkg/ifx_fpibus_transf_s/co...	100%			
	/evc_pkg/ifx_fpibus_transf_s/co...	100%			
2	Transmitter	18.48%		darronm	1
2.1	Bonding_MODE_0	0%		darronm	1
	/concat_tester/CHIPBOND/txpr...	0%			
	/concat_tester/CHIPBOND/txpr...	0%			
2.2	Bonding_MODE_1	0%		darronm	1
2.3	Bonding_MODE_2	50%		darronm	1
	/concat_tester/monitor_channel...	50%			
2.4	Bonding_MODE_3	0%		darronm	1
2.5	FAW-Frame_Alignment_Word_Ge...	5%		darronm	1
	/concat_tester/monitor_channel...	50%			
	/concat_tester/CHIPBOND/prep...	0%			
	/concat_tester/CHIPBOND/prep...	0%			

Figure 8, testplan tracking

within the database.

The fact that any object within the merged UCDB can be linked means that it's possible to link to coverage from the VIPs within both SystemVerilog and 'e', or even a combination of the two. The graphic in Figure 8, testplan tracking, shows the testplan tracker and its link to both SystemVerilog and the coverage from the 'e' VIP. The tracker also allows the analysis of any testplan sections to find out which test has the best and worst coverage, and also can run the ranking algorithm to find the best set of tests for a particular section. This allows the traceability of requirements testing all the way back to the test that covers the requirement.

VII. CONCLUSION

The UCDB allowed for combining of all verification data to give a complete picture of coverage across all metrics. Using formal methods to complement dynamic simulation means gave a more actuate measure of where we were in the process, and there was a massive increase in productivity based on the fact that time was not wasted trying to close on coverage that was not possible to cover. Further improvements to this process are possible by automating some of the steps that have been detailed in this paper. For example, instead of generating exclusion commands to be excluded in each step, the API application could have set the exclusion flags to the objects directly. In short, having a database technology that allows all coverage to be stored in one place and an open API to both read and write data to the database enables a more productive verification process.

ACKNOWLEDGMENT

Avidan Efody, Mentor Graphics for his help on implementing the 'e' coverage extensions.