Coverage Data Exchange is no robbery…or is it?

Darron May
Mentor Graphics,
Newbury, UK
darron_may@mentor.com

Samiran Laha
Mentor Graphics,
Fremont, CA
samiran_laha@mentor.com

***Abstract*-Coverage is extremely important to the modern verification flow. Most vendors have already figured that unifying data across all verification engines leads to a more efficient and integrated environment. There are many challenges to be solved unifying and sharing data across a single vendor's tool set which are further complicated when wanting to share data across multiple vendors' tool sets. The Unified Coverage Interoperability Standard (UCIS) was created to make data sharing and exchange easier by standardizing data models and data exchange. UCIS uses Extensible Markup Language (XML) as an exchange format. This paper introduces a method of exchanging data between two-UCIS compliant coverage database systems without the need for inefficient formats like ASCII or XML.**

I. INTRODUCTION

Over the last few years all the major vendors have realized that unifying the way coverage is stored in a common database allows the results of multiple verification tools to be combined and for these tools to share the data to improve coverage closure. Simulation, Emulation and Formal engines should all be using the same database to provide the user with the complete picture, and allow the analysis of all data in a common way. By defining a common data model all tools can write and read coverage and benefit from the combined strengths of all verification techniques.

Accellera's UCIS subgroup was a technical working committee formed to define a common API, data model and provide a way of exchanging data between multiple vendors and multiple tools. The data model is an extremely important aspect of data exchange and is what complicates the sharing of data amongst proprietary formats. With functional coverage this is a little more straight forward due to standards like SystemVerilog but code coverage can be more complicated, this paper will examine the difficulties with both and illustrates how there is a requirement of some intelligence within any application doing the data transfer.

XML is an excellent exchange format however because it's verbose, capacity problems often arise even when processing even average sized designs. One trick is to compress the ASCII data to reduce the disk space, but this adds to the processing time. For the best capacity and performance an application built upon an API to extract data from vendor A and passed onto a second application built upon vendor B is a better solution. The application cannot only exchange data but it can also do any manipulation required. One weakness in the XML interchange use model: when there is a mismatch in two vendors' supported XML interchange formats, it's not possible to simply import the output generated by the other. Using an application that can manipulate the data received from the source database allows changes/updates to the data before sending to the receiving end. This paper will present a complete solution for moving data from one UCIS-compliant database to another. Some third parties do not have support or have limited support for the UCIS API so this solution will also show how the same method can be applied using two different APIs to provide interchange of coverage data.

II. COVERAGE DATABASES AND DATA MODELS

Coverage databases however they are implemented contain very similar information, at the very lowest basic point it contains counts of defined events. The way these defined events are labeled or arranged within the database with a set of building blocks is defined as the data model for the particular metric.

A. *Mentor's Unified Coverage DataBase (UCDB)*

Architected in 2005 to unify coverage collection across all verification engines, UCDB was first released within Questa and ModelSim in early 2006 as a way of natively storing, analyzing and reporting on functional coverage, code coverage and assertions. The database implementation has an open C API and defines a set of standard

coverage models for all metrics generated by Mentor Graphics tools'. It also provides support of third party and user coverage, along with the ability to extend to coverage models yet to be defined by any tools. This openness has allowed users to bring coverage data from other languages and vendors' tools into the UCDB and thus use the database to completely unify their coverage storage and analysis within their flows. One example is providing the results from code coverage metrics like expression coverage to drive a formal tool to exclude certain patterns based on formal proofs [1].

From the users' point of view, the database has three primary sections. One is the coverage data collection area which is the main focus of the database. The second is for recording of test-specific information, such as test name, tool settings, CPU time, and username and so on, and is extendable to store any user-defined information. This section also includes the history of how the tests were generated, combined and/or merged together. The third section, used for testplan tracking, allows the storage of testplan items that can be linked to the coverage model and/or the test cases themselves.

B. UCDB data model

Designs and testbenches are hierarchically organized. Design units (Verilog modules or VHDL entity/architectures) can be hierarchical though are not always. Test plans can be hierarchical. Even coverage data (of which the SystemVerilog covergroup is the best example) can be hierarchically organized. Hierarchical structures are handled using scopes (also referred to as hierarchical nodes), which store hierarchical structures (i.e., elements of a database that can have children). Coverage and assertion data are stored as counters, which indicate how many times something happens in the design. The basic design of a coverage hierarchy with scopes and coveritems is shown below in Figure 1.

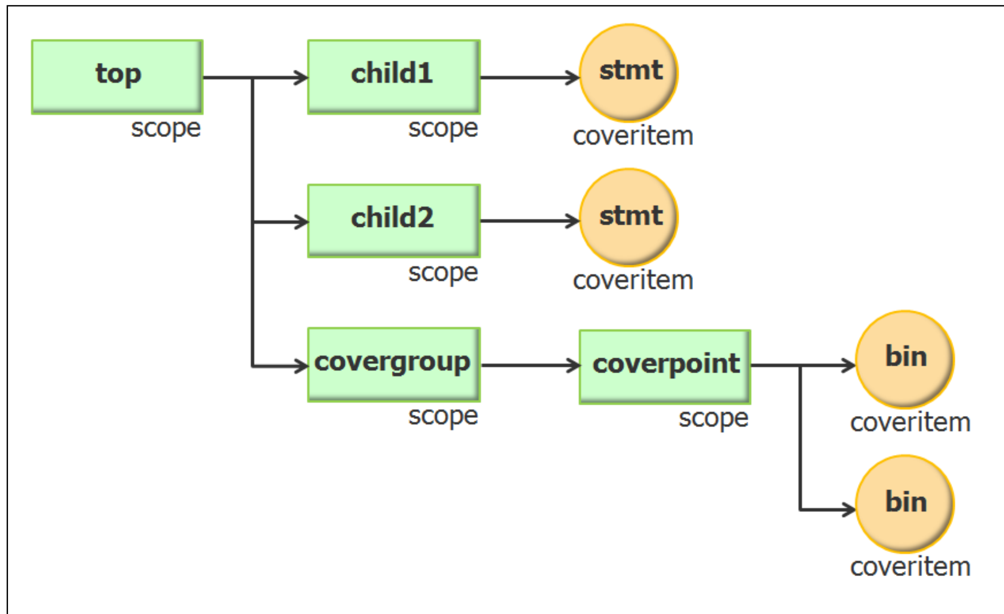

Figure 1 - Basic building blocks

Note that 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. The *scope* and *coveritem* can be decorated with attributes and flags which carry the detailed information about what type of coverage data is being represented. Using these two building blocks, and arranging them into a hierarchy with scoping rules and adding attributes and flags that are predefined to represent known kinds of coverage data allows all known coverage metrics to be supported. They can also be rearranged and additions can be made to attributes to support any type of coverage for extendibility.

C. Coverage model examples

As the coverage model is a transformation of the source code there is a strong connection between the objects and certain aspects of the code. These relationships are stronger for say code coverage metrics, for instance statement or branch coverage where the attributes will carry the information on source code path name, file name, line and token information.

Due to the fact that code coverage is not standard and differs in implementation, trying to match metrics from different vendors can be very tricky. A straight interchange is sometimes not enough. This paper will show how some of these problems can be tackled in the later sections. You would expect coverage models to be exactly the same across implementations when looking at functional coverage from a standard like SystemVerilog, but not even this is true. There are still a few things within the standard that are ambiguous, which means that there can be subtle differences between vendors. In general, SystemVerilog is probably one of the most straightforward coverage models to move between two databases. However, because of some of the possible differences in the ways vendors represent the data, even with SystemVerilog in some instances there is likely the need for some intelligence when moving objects between databases.

III. WHAT UCIS PROVIDES THE INDUSTRY

Accellera's UCIS subgroup was formed in 2006 with three defined goals combined to encourage user and EDA technology advancement. These goals were to identify interoperability opportunities for coverage, to define standard coverage models for the industry and to define an operability standard for data exchange. The 1.0 standard itself was released in June 2012 [2].

A. UCIS database compatibility

For an implementer to be fully compliant to the UCIS standard, they are required to support all possible coverage data models and all of the C API routines defined for accessing the data defined in the specification. The UCDB is the binary implementation of Mentor's coverage database that stores data internally compliant with the UCIS coverage data models. This binary representation supports the ability to access the data using either the proprietary UCDB C API or the standard UCIS C API. When a vendor supports UCIS users often assume that the implementations are binary compatible, which is not correct. Figure below shows the different combinations of UCIS support; note that the implementers of coverage databases require a persistent form of the data to be stored in an implementation-specific manner. (In the case of Mentor this is a UCDB file stored on a disk.)

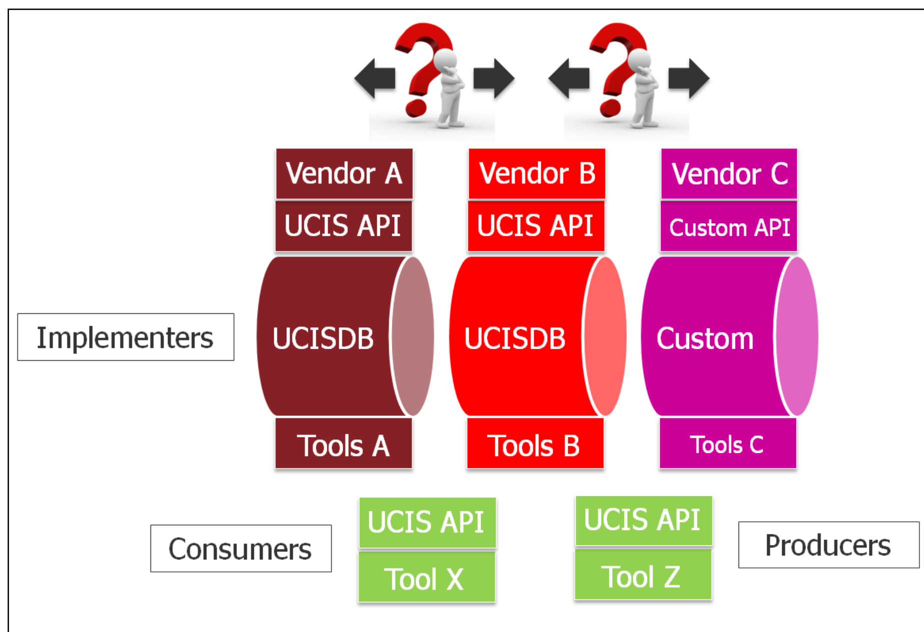

Figure 2 - UCIS compatibility

It is not possible to read vendor A's binary coverage database with a tool compiled to read vendor B's coverage database even if they are both UCIS-compliant data models. The same situation stands if trying to use vendor C's tools to read vendor B's coverage databases. As a consumer of coverage, it is possible to write an application using the UCIS API and then compile and link that with vendor A's UCIS library to produce an application to read and process vendor A's databases. Likewise, the same application could be compiled and linked with vendor B's UCIS library to produce an application to read and process vendor B's databases. This allows common applications to be written by both consumers and producers and compiled with each UCIS implementation [3]. This would not be the case for vendor C, where a custom API application needs to be written.

B. *UCIS interoperability*

UCIS defines all known coverage models and its recommendations on how these models should be represented with the scope and coveritem building blocks plus attributes and flags. The standard provides an XML exchange format allowing vendor A to output XML in a defined format and vendor B to read this XML format and import the data into its database. There are a number of disadvantages in the use of XML. First, it may be very easy for users to read, parse and understand but for the amount of data it has to represent within a typical verification environment, capacity and performance becomes an issue even for average sized designs. Second, and more important, there are some non-overlapping inconsistencies between the internal data representation of the UCIS API and the XML format which currently means that a vendor has to implement and maintain two formats. To facilitate the first release of the standard these inconsistencies were deferred to a future version of the standard. Table 1 below shows the performance and capacity comparisons between using an XML format and an internal binary format for some typical user designs in the range of 4 to 17 million bins.

TABLE I

SOME TYPICAL DESIGNS WITH DATABASE SIZE AND PERFORMANCE MEASUREMENTS

Design	Measurements							
	Size (Millions Bins)	UCDB Size (Mbytes)	XML Size (GBytes)	UCDB size / XML size	Compressed XML (Mbytes)	XML Comp / UCDB size	Gzip time (seconds)	Gunzip time (seconds)
1	4.5	14.0	1.8	128.5	37.0	2.6	8	15
2	5.8	19.7	2.3	116.7	68.8	3.5	48	24
3	7.9	42.5	3.9	91.7	106.0	2.5	53	37
4	17.4	63.0	19.2	304.7	254.4	4.0	262	233

From the table we can see that the XML files are on average 160 times larger than the binary representation of a design. With design sizes increasing, tests numbering in the thousands to tens of thousands, and many regressions run on a daily basis, the result can be a huge amount of data. Even compressed XML files (note that compression is not part of the UCIS process) are on average three times larger, not much of a relief considering that this approach also adds an extra burden of compressing/uncompressing any time the data needs to be processed.

IV. AN ALTERNATIVE DATA EXCHANGE METHOD

A. *Using the API*

Exchanging data without the need for XML is straightforward when the two implementations exchanging data have full implementations of both the coverage models and API as defined within the standard. This section introduces the basic blocks of a UCIS-to-UCIS exchange application; it requires both sides to support the data models for the coverage metrics being exchanged.

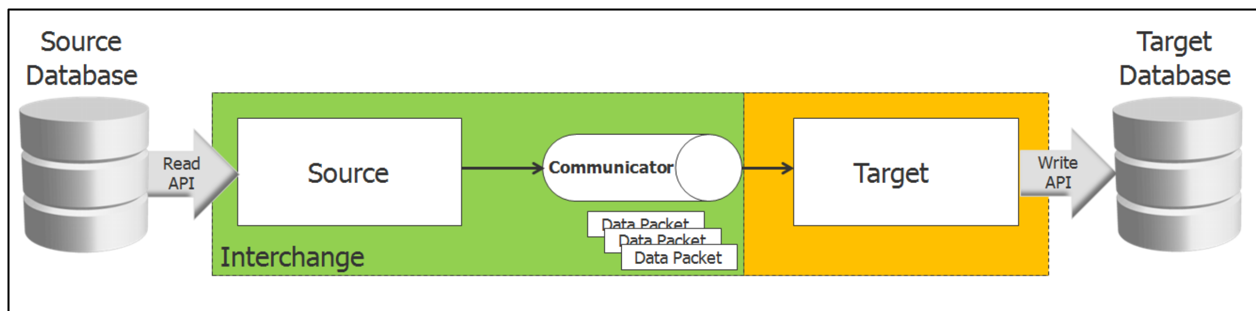

Figure 3 - Environment that allows two UCIS API implementations to be compiled into a single executable that can read vendor A's binary database and write vendor B's binary database.

The 'Source' block reads data from the source database using the vendor-specific UCIS API, which created the source database. The 'Source' block is created by linking the UCIS C library supplied by the vendor which contains the definition of UCIS API routines for that vendor. By linking the application against other vendor-supplied UCIS C libraries, the same application can be used by any implementation. Data read by the 'Source' block is passed to the 'Communicator' block by calling some specific C routines defined by the 'Communicator' block.

The 'Communicator' block is responsible for accepting data in its own format. It supplies a small set of C library routines which the 'Source' block calls to send data to the 'Communicator'. The 'Communicator' receives data from

‘Source’, packages it up and sends that over to the ‘Target’ block. There is another set of small C library routines provided by the ‘Communicator’ block which is called by the ‘Communicator’ to send data to ‘Target’.

The ‘Target’ block has to register callbacks for each of the ‘Communicator’ block’s C function handles to receive callbacks when data arrives in the ‘Communicator’ block destined for the ‘Target’ block. The ‘Target’ block registers those callbacks at the beginning and then waits for receiving data via any of the callback routines. After receiving the data, the ‘Target’ block writes the corresponding vendor-specific UCIS database depending on the linked vendor’s UCIS C library. This architecture allows the ‘Source’ block and ‘Target’ block to be linked with any two vendors’ UCIS C libraries and thus to convert one vendor’s database to another vendor’s database.

B. The implementation

This section explains the implementation and how the blocks allow customization on both the read and write sides of the exchange. This implementation uses a read streaming mode to extract data on the source side and writes data using the write streaming mode on the target side.

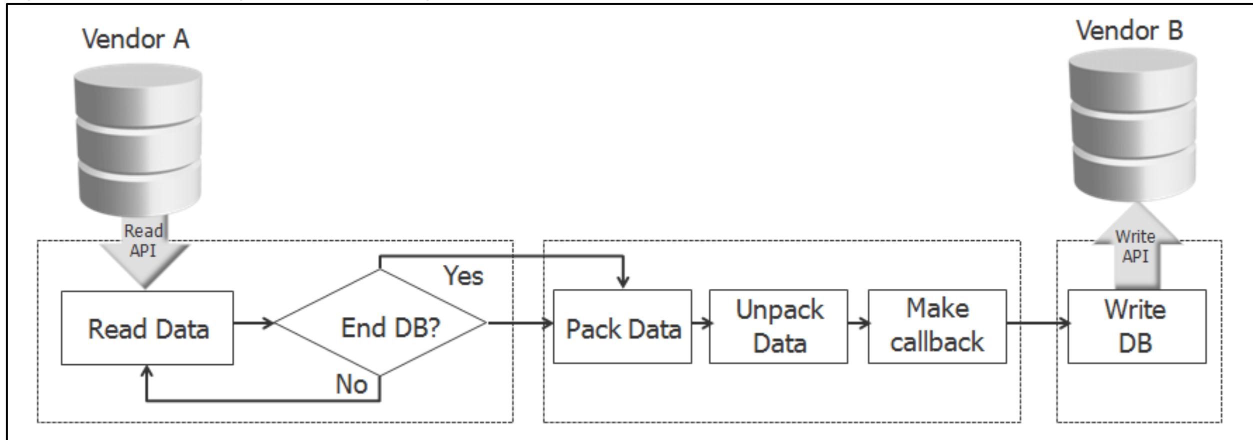

Figure 4 – Data path diagram

The ‘Source’ block is linked with a UCIS C library provided by Vendor A to convert the database created by Vendor A. The ‘Target’ is linked with the UCIS C library provided by Vendor B to create a Vendor B database.

The ‘Source’ block calls the UCIS standard ucis_OpenReadStream() API routine to read the UCIS database in read streaming mode. Note that the UCIS read streaming mode is a faster and more memory-efficient mode for scanning through the database objects. This function scans data from the UCIS database file which the source block sends to the communicator block by calling the a set of C routines provided by the communicator block for the major building blocks of the coverage database (i.e send_testdata(), send_cross_scope(), send_cover() etc.). The ‘Communicator’ receives data from those function calls made by the source block. It then packages the data one at a time and sends the packaged data to the ‘Target’ block. The other part of the ‘Communicator’ block is linked to the front of the target block, which receives packaged data, unpacks the data and then sends it to the ‘Target’. The ‘Communicator’ block provides callback registrations for ‘Target’ block to register the functions (i.e. receive_testdata(), receive_cross_scope(), receive_cover() etc.).

The ‘Target’ block registers callbacks for those functions by using the register_comm_callback() communicator supplied routine with the provided function handles for each object.

After receiving the first data via receive_initdb() callback, the target block calls the standard UCIS C API routine ucis_OpenWriteStream() to open a database file intended to write data in streaming mode. Note that the UCIS write stream mode is a faster and more memory-efficient mode for writing data in a database file directly. After receiving each data packet via any of those callback routines, the target block calls the appropriate UCIS C API routine to write the corresponding data into the database file. After receiving the receive_enddb() callback, the target block calls the ucis_Close() routine to finalize the writing of database, the database conversion becomes complete, and the program terminates. The vendor-B-created UCIS database is then ready to be read by any Vendor-B-supplied database processing tool.

C. Supporting multi coverage metrics

This section explains how SystemVerilog functional coverage and toggle coverage metrics are supported within the system. There is also the question of what will be done with the data once it is in the target database. Compared to just analyzing the data coming from the source alone, there are further requirements on the exchange of data if the

data is to be combined or merged with similar data from other sources. An example with a module top has one toggle coverage scope which has two bins, and a SV covergroup scope, which has two coverpoints and one cross scope. The coverpoints and the cross have one bin each. The source block opens the input database in read streaming mode using the ucis_OpenReadStream() API to scan through the above coverage data hierarchy. It receives the following list of callbacks from that API routine for the coverage data stored in the input database, and calls the appropriate communicator routine as shown below:

- database_init_callback: calls send_initdb() and send_attribute() for each global attribute
- test_data_callback: calls send_testdata() and send_attribute() for each test data attribute
- module_inst_scope_callback: calls send_module_instance_scope() and send_attribute() for each scope attribute
- toggle_scope_callback: calls send_toggle_scope() and send_attribute() for each toggle scope attribute
- toggle_bin_callback: calls send_cover() and send_attribute() for each bin attribute
- toggle_bin_callback: calls send_cover() and send_attribute() for each bin attribute
- toggle_endscope_callback: calls send_endscope()
- covergroup_scope_callback: calls send_general_scope() and send_attribute() for each scope attribute
- coverpoint_scope_callback: calls send_general_scope() and send_attribute() for each scope attribute
- bin_scope_callback: calls send_cover() and send_attribute() for each bin attribute
- coverpoint_endscope_callback: calls send_endscope()
- coverpoint_scope_callback: calls send_general_scope() and send_attribute() for each scope attribute
- bin_scope_callback: calls send_cover() and send_attribute() for each bin attribute
- coverpoint_endscope_callback: calls send_endscope()
- cross_scope_callback: calls send_cross_scope() and send_attribute() for each scope attribute
- bin_scope_callback: calls send_cover() and send_attribute() for each bin attribute
- cross_endscope_callback: calls send_endscope()
- covergroup_endscope_callback: calls send_endscope()
- module_instance_endscope_callback: calls send_endscope()
- end_of_database_callback: calls send_enddb()

The ‘Communicator’ packages the received data as it receives, and sends that to the ‘Target’. The ‘Communicator’ block on the other side unpacks data and calls the appropriate callback routine registered by the ‘Target’ block. The communicator calls the registered callback routines registered by the ‘Target’ block. The following callbacks are made in the given order for this coverage database.

- receive_initdb() followed by some receive_attribute()
  - It then calls ucis_OpenWriteStream() API routine to start writing the converted database
- receive_testdata() followed by some receive_attribute()
  - It then calls ucis_CreateHistoryNode() to add the test data record in new db
- receive_module_instance_scope() followed by some receive_attribute()
  - It then calls ucis_CreateInstanceByName() to add the module instance scope in new db
- receive_toggle_scope() followed by some receive_attribute()
  - It then calls ucis_CreateToggle() to add the toggle scope in new db
- receive_cover() followed by some receive_attribute()
  - It then calls ucis_CreateNextCover() to add the toggle bin in new db
- receive_cover() followed by some receive_attribute()
  - It then calls ucis_CreateNextCover() to add the second toggle bin in new db
- receive_endscope()
  - It then calls ucis_WriteStreamScope() to finish writing the toggle scope and pop to the parent module instance scope
- receive_general_scope() followed by some receive_attribute()
  - It then calls ucis_CreateScope() to add the covergroup scope in new db
- receive_general_scope() followed by some receive_attribute()
  - It then calls ucis_CreateScope() to add the coverpoint scope in new db
- receive_cover() followed by some receive_attribute()
  - It then calls ucis_CreateNextCover() to add the coverpoint bin in new db
- receive_endscope()

    - It then calls the ucis_WriteStreamScope() to finish writing the coverpoint scope and pop to the parent covergroup scope
- receive_general_scope() followed by some receive_attribute()
    - It then calls ucis_CreateScope() to add the second coverpoint scope in new db
- receive_cover() followed by some receive_attribute()
    - It then calls ucis_CreateNextCover() to add the coverpoint bin in new db
- receive_endscope()
    - It then calls the ucis_WriteStreamScope() to finish writing the current coverpoint scope and pop to the parent covergroup scope
- receive_cross_scope() followed by some receive_attribute()
    - It then calls ucis_CreateCrossByName() to add the cross scope in new db
- receive_cover() followed by some receive_attribute()
    - It then calls ucis_CreateNextCover() to add the cross bin in new db
- receive_endscope()
    - It then calls the ucis_WriteStreamScope() to finish writing the cross scope and pop to the parent covergroup scope
- receive_endscope()
    - It then calls the ucis_WriteStreamScope() to finish writing the covergroup scope and pop to the parent module instance scope
- receive_endscope()
    - It then calls the ucis_WriteStreamScope() to finish writing the top-level module instance scope
- receive_enddb()
    - It then calls ucis_Close() to finish writing the new database, and the database conversion completes

Once the data is available to the 'Target' block, it can do whatever it wants instead of just creating a new database. For example, it could generate a report out of that data. It could also merge the received data directly into another opened UCIS database, etc.

V. SUPPORTING NON UCIS COVERAGE

Some coverage vendors do not support the UCIS API so there is a need to allow mixing of APIs and the addition of extra customization code to make adjustments between the two formats being exchanged. The architecture of this exchanger is able to plug and play with any coverage API on either the source of target sides.

A. Using UCDB API for the target

Due to the fact the UCDB is able to support dual APIs, this section details how the source block was replaced with an application implemented using the UCDB API. This allows the architecture to be proved by reading a Questa-generated UCDB file using the UCDB API and writing a new database binary using the UCIS API and comparing the two databases to ensure that the data was transferred successfully.

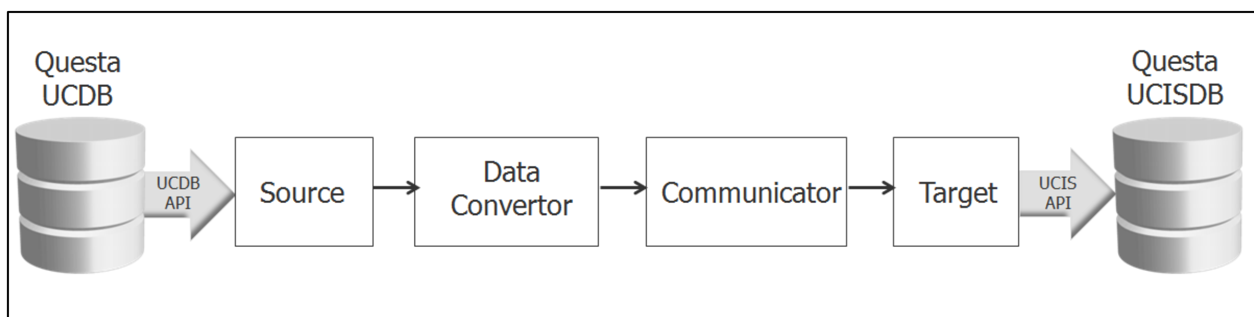

Figure 7 - Block diagram showing Data Converter in Source block

The additional Data Converter block is placed between the 'Source' and 'Communicator' blocks. That block makes the necessary data manipulation to map the UCDB data types to the corresponding data types, if they are different. The communicator understands UCIS data types, so when the Data Converter converts a data of a different type that can go directly to a UCISDB without any further modification. In this case only one Data Converter is

needed on the source side as the target is generating UCISDB. There were some additions to the UCDB that were not standardized; this architecture allows those additions to be handled.

B. Using UCAPI and Unicov API

The ‘Target’ and ‘Source’ blocks can be replaced with an application using any coverage database API, thus making it possible to exchange data. This of course means that the differences in the data representation and data models have to be taken care of within the ‘Target’ and ‘Source’ blocks. Moving coverage data from one vendor to another requires two Data Converter blocks, one on the source side and another on the target side, as both the source and target are using non-UCIS coverage databases. The source side could be replaced with an application written with the UCAPI API to allow Synopsys coverage databases to be read, and the target side could be replaced with an application written using the Unicov API to allow a Cadence coverage database to be written. The Data Converter block on the source side converts the UCAPI coverage data model to a UCIS-compliant data model, which is then transported over the target side. The Data Converter block on the target side converts the UCIS-compliant data model to a UCD-compliant data model which the Target Block understands. The Target Block uses Unicov API and writes coverage data in UCD format.

One general example of moving coverage data that could be represented differently by two different vendors is SV covergroup array bins. A coverpoint under a covergroup may have an array of bins that may be represented differently by using two different data models. In one model it could be a flat list of bin objects under the parent coverpoint scope, while in the other model there could be an additional Userbin-arrray scope holding all the bins within it and placed under the parent coverpoint. Both of these methods of storing data are defined and allowed within the UCIS standard. The flat data model can go from source to target without any modification, but the Converter Block on the target side needs to create an extra scope for the Userbin-array and move all the bins from coverpoint scope to the new Userbin-array scope while putting the new scope under the coverpoint scope. Then the data model supported by the target API becomes the same, and the Target Block can write the coverage data in its own format. This would not be possible with the exchange of data using fixed XML.

C. Problems supporting differing coverage models

Functional coverage can be adjusted between coverage database implementations due to the fact that they are representing the same SystemVerilog standard. But as there are no standards for code coverage, this may produce more difficulties as different vendors implement different data models. Here is a possible list of different code coverage data models which could be very hard to transform from one to another.

- Statement and Branch coverage: Line coverage vs. block coverage
    - In this case additional data is needed to convert a line coverage data model to a block coverage data model for identifying blocks properly. A vendor supporting line coverage data model may not have that extra data.
- Expression coverage: Flat list of nodes and variables vs. hierarchically represented sub-expressions
    - In this case bin counts in both models could be totally independent, so conversion of one data model to another could be very difficult.
- Toggle coverage: Simple transitions among 0, 1, and Z vs. merging Z with 0 or 1 based on some mode
    - In this case also the bin counts could be totally unrelated; hence conversion may not be possible.

VI. Conclusions

A basic exchange of coverage data between vendors using XML is limited not only with respect to performance and capacity, but also in that data models can vary even between vendors who have chosen to implement based on the standard. Therefore a different method is required that allows adjustments to be made during the data exchange process. This paper detailed the history of the UCIS and how data is represented within coverage databases. It also explained a very flexible method that is currently being employed to move coverage data between vendors using proprietary APIs that allow full read/write access to the vendor’s coverage data.

References

[1] “Blending multiple metrics from multiple verification engines for improved productivity”, Darron May and Darren Galpin, DVCon 2012.

[2] Accellera Unified Coverage Interoperability Standard (UCIS) Version 1.0, June 2, 2012.

[3] “UCIS applications: improving verification productivity, simulation throughput and coverage closure tracking”, Ahmed Yehia, DVCon 2013.