UCIS APPLICATIONS: IMPROVING VERIFICATION PRODUCTIVITY, SIMULATION THROUGHPUT, AND COVERAGE CLOSURE PROCESS

Ahmed Yehia
Mentor Graphics Corp.
Cairo, Egypt
ahmed_yehia@mentor.com

ABSTRACT

Given today's design sizes and complexities, complex verification environments are built to ensure high levels of quality, reliability, and stability are maintained in the DUV (Design Under Verification) before tape out. Yet, this challenges the ability to analyze and track the huge amount of data generated from today's verification environments and manage the underlying resources. Vast amounts of simulation and coverage data (due to huge regressions runs and long simulation cycles) need to be analyzed and tracked to help answer the important verification questions "Does it work?", "Am I done?", "What am I missing to get things done?", "How can I improve my productivity?"

The Accellera's Unified Coverage Interoperability Standard (UCIS) is a new, open, and industry-standard API just released in June 2012. It promises to facilitate and improve on verification productivity. It provides an application-programming interface (API) that enables sharing of coverage data across multiple tools from multiple vendors [1].

In this paper, we present several generic UCIS applications that can be easily developed and deployed to help improve the verification productivity, simulation throughput, coverage analysis, and coverage closure process.

1. INTRODUCTION

The increasing size and complexities of today's designs introduce many challenges to the verification goals, tasks, and process that result in a noticeable increase in the effort spent on verification, and has forced the necessity to increase the industries adoption of various new functional verification techniques [2]. Typical verification environments encompass automation, coverage-driven constrained random verification, assertion-based verification, and transaction level modeling, while sophisticated verification environments may include formal verification, analog-digital mixed signal verification, and software-hardware co-simulation.

Indeed, many verification technologies and methodologies adopted by the industry have succeeded in improving the verification process in general, yet many verification challenges still exist in managing the verification process, writing/maintaining tests, time to debug, reaching coverage closure, and tracking project momentum smoothly.

The Accellera UCIS committee has released the Unified Coverage Interoperability Standard (UCIS) as an open and industry-standard API that promises to facilitate and improve verification productivity, and with an ultimate goal to allow interoperability of verification metrics across different tools from multiple vendors.

In this paper, we show how UCIS can help ease some of today's verification challenges, especially:

- Tests and testbench quality improvement.
- Simulation and regression throughput.
- Analysis of verification metrics.
- Coverage closure process.
- Project tracking.

2. UNIFIED COVERAGE INTEROPERABILITY STANDARD (UCIS) OVERVIEW

The Accellera's Unified Coverage Interoperability Standard (UCIS) is an open and industry-standard API that promises to facilitate and improve on verification productivity. It provides an Application-Programming Interface (API) that enables the sharing of coverage data across software simulators, hardware accelerators, symbolic simulations, formal tools or custom verification tools [1]. The Accellera UCIS committee was formed in November 2006, many companies joined and contributed to the committee including EDA vendors and user representatives from the largest companies in the industry. Mentor Graphics donated its UCDB (Unified Coverage DataBase) technology in June 2008 as a starting point of the UCIS standard. The UCIS 1.0 standard was released in June 2012.

UCIS defines a coverage database (UCISDB) as a single repository of all coverage data from all verification processes (e.g. Functional and Code Coverage, Formal Verification, Assertion Based Verification, Emulation, verification plan goals and objectives, user-defined coverage, etc.). The two main structures that form a UCISDB are:

- *Scope*: A hierarchical node that stores hierarchical design structure. A scope can have children scopes or coveritems.
- *Coveritem*: Holds actual counts of recorded events. A coveritem is a leaf construct; it cannot have children, and is essentially an informational wrapper around an integral count.

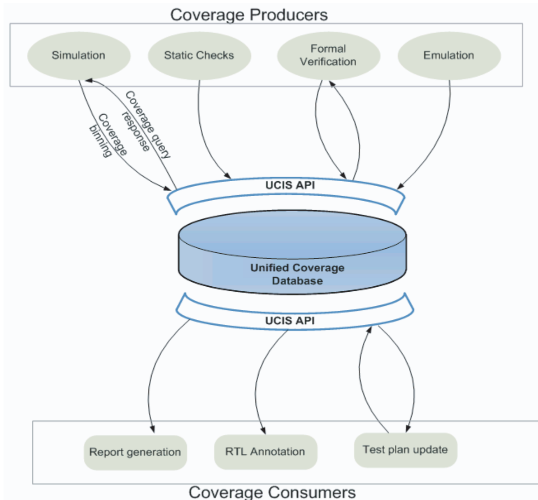

Figure 1. The UCISDB

Focusing on functional coverage[1] in this paper, and taking the IEEE SystemVerilog [3] language as a vehicle, a SystemVerilog *Covergroup*[2] would be represented in a UCISDB as a scope which has a parent scope (design unit enclosing the *Covergroup*), and child scopes (*Coverpoints*[3] and *Crosses*[4]). *Coverpoints* and *Crosses* are scopes that contain leaf nodes *Bins*[5] that in turn are represented as coveritems in a UCISDB.

The UCIS also defines specialized history nodes to describe primary coverage in a UCISDB like a *Test Record*. The test record specialization is a history node with which coverage counts may be associated. This provides a great value in attributing coverage numbers achieved to individual tests in regression run, which would help in merging, ranking of coverage data while assessing and improving the quality of tests.

The UCIS defines data structures and API functions, which are a group of C language [4] data structures and functions that enable access and manipulation of data inside a UCISDB. Once a UCISDB has been generated to hold coverage data, the UCIS API would enable opening and closing a UCISDB, navigating scopes, extraction, and manipulation of data inside a UCISDB. Out of the entire UCIS API (approximately 110 routines), we will focus on the API of interest that supports the UCIS applications presented in this paper.

- *ucis_Open()*: Creates an in-memory database, optionally populating it from a specified UCIS file. Returns a database handle.
- *ucis_Close()*: Invalidates the specified database handle and frees all memory associated with the handle.
- *ucis_MatchScopeByUniqueID():* Finds a single scope in an in-memory database by its Unique ID. The unique ID string may be the full or relative form.
- *ucis_GetScopeType()*: Returns the scope type of the specified scope.
- *ucis_ScopeIterate()*: Returns an *iterator* handle used to scan all scopes below the specified starting scope.
- *ucis_ScopeScan()*: Returns next scope handle in a specified *iterator* handle.
- *ucis_CoverIterate()*: Returns an *iterator* handle used to scan all coveritems below the specified starting scope.
- *ucis_CoverScan()*: Returns next coveritem handle in a specified *iterator* handle.
- *ucis_GetCoverData()*: Gets name, data and source information for the specified coveritem.
- *ucis_FreeIterator()*: Frees an iterator handle.
- *ucis_HistoryScan()*: Serially scans through the history nodes of the type selected by the iterator.
- *ucis_HistoryIterate()*: Returns an *iterator* handle used to scan all history nodes below the specified starting scope.
- *ucis_AttrAdd()*: Adds the specified attribute (key/value) to the specified database object or global attribute list.
- *ucis_CallBack()*: Traverses the part of the database rooted at and below the specified starting scope.
- *ucis_CreateHistoryNode()*: Creates a history node of the specified kind in the specified database.

In the next sections, we present a variety of UCIS applications together with testbench examples that can be deployed either during simulation runtime, or in post-run mode to help improve verification productivity, simulation throughput, coverage analysis, and the coverage closure process. The UCIS applications presented are written in C language, while testbench examples presented are written in SystemVerilog language. At the time of writing this paper, all the major vendors have publicly announced their intent to

[1] The terms “functional coverage” and “coverage” are used interchangeably in the paper to describe DUV functional coverage.
[2] A Covergroup is a construct that encapsulates the specification of a coverage model. It may contain coverpoints, crosses, and options.
[3] A Coverpoint specifies an integral expression to be covered.
[4] Specifies the cross coverage between two or more Coverpoints.
[5] Each coverage point includes a set of bins associated with its sampled values or its value transitions.

support the UCIS API. All the applications presented in this paper are tested on the Questa© simulator which currently supports the UCIS API, however due to the standardization of the UCIS and SystemVerilog, the applications presented should work on any simulator that supports the standards[6].

3. UCIS RUNTIME APPLICATIONS

In this section, we present some generic UCIS applications that can be easily deployed in any project. Making use of the UCIS API, these applications can be loaded as shared objects during simulation and be connected to the running test/testbench to feedback information of interest to the test controller at runtime. It is also possible to save test/testbench specific data of interest in a UCISDB generated from a simulation run for further post-run processing steps. An abstract summary of benefits and goals of these applications would be to:

- Maximize simulation throughput.
- On the fly change tests' runtime behavior upon collected coverage analysis.
- Track tests quality.

3.1 Testbench and Tests Preparation

Before presenting the UCIS applications that a regular test/testbench can make use of, we need to prepare the test/testbench for this usage. This section describes the preparations needed.

3.1.1 Linking a Test to its Coverage Goals and Targets

A test is normally written by a verification engineer to fulfill specific objectives and exercise specific scenarios. Scenarios, metrics and scopes in the testbench coverage model (inspired from the verification plan), normally provide in depth translation of the test/testbench goals and targets. Therefore, the first step in tracking and maximizing a test quality is to link the test implementation to its goals and objectives. To ensure modularity and reuse, this can be achieved by passing to a test at runtime the scope(s) and scenario(s) in the coverage model it needs to exercise. This information can be embedded in dynamically executable documents like the verification plan document, or the verification environment scripts holding the tests information.

Many modular ways exist to pass coverage scopes of interest to a test: (1) automatically generated systemVerilog files to be included by the test, (2) text file to be read by the test at runtime, (3) value plusarg passed to the test execution command line (CLI) as shown below. The example below shows a test based on UVM [5].

```
typedef struct {
  string scope, coverholes;
  string coveritems [];
  real coverage_score;
} coverscopesT;

class test1 extends uvm_test;
  //Coverage scopes of interest queue
  coverscopesT coverscopes [];
  function new(string name,
               uvm_component parent);
    string coverscopes_s, s;
    int cnt, i;
    byte char;
    super.new(name, parent);
    ...
    if ($value$plusargs("COVERSCOPES=%s",
                         coverscopes_s)) begin
      while((char=coverscopes_s.getc(cnt++))!=0)
        //Mark coverage scopes separators
        if (char != ",")
          s = {s, string'(char)};
        else begin
          coverscopes[i].scope = s;
          s = "";
          i++;
        end
      if (s != "") coverscopes[i].scope = s; end
  endfunction
endclass
```

As shown above, the test takes the value of the *COVERSCOPES* plusarg when passed to the CLI. If a test is targeting several coverage scopes, one can pass them all separated by commas to the *COVERSCOPES* plusarg on the CLI, the test would then parse the long string provided by the CLI and populate a list of coverage scopes of interest for ease of use later on. A test could also be interested in specific coveritems (bins) in a specific scope; in this case, one could update the *coveritems* queue of the *coverscopesT* to hold the bin names of interest for a specified scope. The simulation CLI in this case would be as follows:

```
vsim fpu_tb_top +UVM_TESTNAME=test1
     +COVERSCOPES=/19:fpu_agent_pkg/11:fcoverage
```

3.1.2 Connecting UCIS Applications to HVL Testbench

All High-level verification languages define a way to connect to foreign programming languages. The SystemVerilog language defines the Direct Programming Interface (DPI) as an interface between SystemVerilog and a foreign programming language. DPI allows direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called directly from SystemVerilog; such functions are referred to as *imported functions*. In our case, we will import the C language UCIS method, and call it directly from the SystemVerilog code. When to call the method is left to the user to define according to runtime needs.

```
import "DPI-C" function string scanCovScope(
                string dbname, string scopename)
```

[6] Processing the output data of an UCIS API call could differ from one vendor to another, since data representation inside a UCISDB is implementation and data dependent, however, the UCIS interface of a UCISDB should be common across vendors.

In the example above, *scanCovScope()* is a C function to be called directly from the SystemVerilog testbench, provided that the shared library object holding the method implementation is loaded at runtime.

3.2 Guiding Tests Behavior at Runtime using UCIS Applications

After getting tests and testbench prepared for using UCIS applications, in this section we present some applications to guide test/testbench runtime behavior based on detailed coverage analysis of the UCISDB.

3.2.1 Monitoring Coverage Score of Scopes of Interest

If a test can monitor and score the dynamically achieved coverage of its scopes of interest at runtime, then it could take interesting actions:

- If the achieved coverage of a scope (i.e. test target) hits its predefined goal, a test would then change the execution sequence to focus on other targets (if other targets are not met), or just quit the simulation to save time and resources when all coverage targets are met.
- When achieved coverage momentum of a scope stalls or falls behind a predefined threshold value, the test can change tactics to focus on the remaining coverage holes.

A UCIS application to monitor the coverage of specific scope would be as follows:

```
int checkCoverGoalMet(const char* dbname,
const char* scopename, double* scopecoverscore){
  ucisT db; /*UCIS in-memory DB handle*/
  ucisScopeT scope = NULL;/*UCIS scope handle*/
  int scopecovergoal;
  /*Populate in-memory DB from physical UCISDB*/
  db = ucis_Open(dbname);
 /*Find matching scope by name, get its handle*/
  scope = ucis_MatchScopeByUniqueID(db, NULL,
                                    scopename);
  *scopecoverscore = coverageScore(db, scope);
  scopecovergoal= ucis_GetIntProperty(db,scope,
                  -1, UCIS_INT_SCOPE_GOAL);
  /*Close DB and return list of holes string*/
  ucis_Close(db);
  return(*scopecoverscore*100 >=scopecovergoal);
}
```

In the above code we defined a C function *checkCoverGoalMet()*. The C function would take as arguments the UCISDB file name, as well as the functional coverage scope name of interest, then returns true if coverage of the specified scope achieved its predefined goal. The *checkCoverGoalMet()* function does the following:

- Loads and populates an existent physical UCISDB file in memory.
- Finds a matching scope with same name as the *scopename* argument and gets a handle for it. If a "NULL" argument was passed, then the coverage scoring will be applied to the entire UCISDB which requires the *ucis_CallBack()* shown in section 4.2.
- Calculate the current coverage percentage of the scope of interest using the *coverageScore()* method presented below[7].
- Returns TRUE if coverage goal was met otherwise returns FALSE.

```
double coverageScore(ucisT db,ucisScopeT scope){
 ucisScopeTypeT scopetype; /*Type of a scope*/
 double total_coverage = 0;
 int weight, total_weight;
 if (scope) {
  /*Determine the type of the found scope*/
  scopetype = ucis_GetScopeType(db, scope);
  if((scopetype == UCIS_COVERPOINT)||
     (scopetype == UCIS_CROSS)){
    /*Coverpoint or a cross scope*/
    total_coverage=coverageScoreCore(db,scope);

  } else if(scopetype == UCIS_COVERGROUP)  {
    /*Covergroup scope: Loop on all sub-scopes*/
    ucisScopeT subscope = NULL;
    ucisIteratorT iterator     = ucis_ScopeIterate(
                         db, scope, -1);
    while(subscope = ucis_ScopeScan(db,
                                iterator)){
      weight = ucis_GetIntProperty(db,subscope,
                -1, UCIS_INT_SCOPE_WEIGHT);
      total_coverage += coverageScoreCore(db,
                           subscope) * weight;
      total_weight   += weight;
    }
    /*Free the iterator handle*/
    ucis_FreeIterator(db, iterator);
    total_coverage = total_weight ?
             total_coverage / total_weight : 0;
  }
 } else {/*NULL scope, score entire UCISDB*/}
  return total_coverage;
}/*coverageScore*/
double coverageScoreCore (ucisT db,
                     ucisScopeT scope){
  int coveritems_hit, coveritems_num;
  iterator = ucis_CoverIterate (db, scope,
                            UCIS_ALL_BINS);
  while((binindex = ucis_CoverScan (db,
                 iterator))!= -1){
    ucis_GetCoverData(db, scope, binindex,
               &binname, &cvdata, &srcinfo);
    if ((cvdata.type == UCIS_ILLEGALBIN) ||
        (cvdata.type == UCIS_IGNOREBIN)) {
      continue;
    }
    coveritems_num++;
    /*If bin is covered. Assume 64-bit UCISDB*/
    if(cvdata.data.int64 >= cvdata.goal){
      coveritems_hit++;
    }
  }
  /*Free the iterator handle*/
  ucis_FreeIterator (db, iterator);
  return (double) coveritems_hit/coveritems_num;
}/*coverageScoreCore*/
```

The SystemVerilog test would then make use of the C *checkCoverGoalMet()* method as shown below. In its *run_phase()* method:

[7] Assuming coverage scope types and option.merge_instances=1.

- A continuous loop to monitor coverage score of scopes of interest starts by saving the runtime coverage data, residing inside the memory allocated by the simulator, in a UCISDB. The SystemVerilog language does not provide means to access memory handles of the coverage information at runtime, hence the need to use the UCIS API. The SystemVerilog language defines the *$coverage_save()* method to save coverage runtime information in a database, however the method specification is limited to code and assertion coverage types and does not extend to functional coverage types. The *$coverage_save_mti()* method used below is a simulator specific method that can be considered as a super set of *$coverage_save()*, extending the specification to functional coverage types, and allowing to save coverage information underneath specific scope of interest.
- When one of the scopes coverage score meets its predefined goal, a corresponding UVM event is triggered and propagated to all the testbench components. Testbench components (i.e. sequencers, sequences, drivers, etc.) can stop and/or alter execution of current execution threads and focus on meeting coverage goals for next targets.
- When all tests' targets are met, then there is no need to continue running the test; terminating the test at this point would save time and free resources required for simulation runs.

```
import "DPI-C" function int checkCoverGoalMet
(string dbname, string scopename,
 output real scopecoverscore);

task test1::run_phase(uvm_phase phase);
  uvm_event target_met_e;
  phase.raise_objection (this, "test1");
  fork
    //Coverage score monitor loop
    while (1) begin
      //When no more targets to cover break
      if (i >= coverscopes.size())  break;
      wait_for_next_transaction();
      //Save snapshot of runtime coverage
      assert(!$coverage_save_mti("test1.ucisdb",
                  "/fpu_agent_pkg/fcoverage"));
      if (checkCoverGoalMet("test1.ucisdb",
          coverscopes[i].scope,
          coverscopes[i].coverage_score) > 0)
      begin
        //Notify testbench components when
        //specific coverage target is met
        uvm_config_db # (uvm_event)::get (agent,
       "", {coverscopes[i++].scope, "_goalmet"},
         target_met_e);
        target_met_e.trigger();
      end
    end
    //Start main test virtual sequence
    test1_v_seq.start(m_env.m_v_sqr);
  join_any
  phase.drop_objection (this, "test1");
endtask
```

3.2.2 *Inspecting Coverage Momentum and Extracting Coverage Holes of Scopes of Interest*

A test monitoring the coverage momentum of its targets, may decide to terminate active threads and/or initiate others when coverage momentum stalls or falls behind a pre-defined threshold, this indirectly improves simulation throughput. Another decision would be to fetch remaining coverage holes (uncovered coveritems) from the coverage model, this would be useful for other testbench components to re-define the tactics of sequences execution or even modify the constrained random data generated of the system transactions to fully cover the test targets.

```
import "DPI-C" function string getCoverHoles
                     (string dbname,
                      string scopename);

task test1::run_phase(uvm_phase phase);
  //Coverage monitor loop
  while (1) begin
    assert(!$coverage_save_mti("test1.ucisdb"));
    if (checkCoverGoalMet("test1.ucisdb",
        coverscopes[i].scope,
        coverscopes[i].coverage_score) > 0)
      ...
    else begin
     //compute coverage momentum when coverage
     //goal is not met
     coverage_momentum =
               coverscopes[i].coverage_score /
               num_of_trans;
     if(coverage_momentum < threshold_momentum)
     begin
       coverscopes[i].coverholes =
          getCoverHoles("test1.ucisdb",
                        coverscopes[i].scope);
       uvm_config_db # (uvm_event)::get (agent,
        "", {coverscopes[i].scope, "_holes"},
        coverholes_update_e);
       coverholes_update_e.trigger();
     end
    end
  end
endtask
```

The code above shows how a test checks coverage momentum, i.e. coverage score w.r.t. number of transactions[8] generated. When coverage momentum decreases below a threshold value, the test calls *getCoverHoles()* to extract a list of coverage holes of the scope of interest then it triggers a UVM event to notify all interested testbench components (i.e. sequences and stimuli generators) that coverage holes of current scope/target are available. Appendix A shows a hypothetical example of how stimuli generators can make use of the extracted coverage holes string to guide the constrained randomization attempts of the system transactions, on the fly during runtime, to focus on covering uncovered coveritems. The implementation of *getCoverHoles()* is as follows:

[8] A transaction is a single transfer of high-level representation of control or data from random stimuli generator to DUV.

```
char* getCoverHoles(const char* dbname,
                    const char* scopename){
  ucisT db; /*UCIS in-memory DB handle*/
  ucisScopeT scope = NULL; /*UCIS scope handle*/
  ucisScopeTypeT scopetype; /*Type of a scope*/

  char * holeslist; /*List of holes string*/

  /*Populate in-memory DB from physical UCISDB*/
  db = ucis_Open(dbname);

 /*Find matching scope by name, get its handle*/
  scope = ucis_MatchScopeByUniqueID(db, NULL,
                                    scopename);
  /*return if required scope is not found*/
  if(scope == NULL){return NULL;}

  /*Determine the type of the found scope*/
  scopetype = ucis_GetScopeType(db, scope);

  if((scopetype == UCIS_COVERPOINT)||
     (scopetype == UCIS_CROSS)){
    /*Coverpoint or a cross scope*/
    populateHolesList (db, scope, &holeslist);

  } else if(scopetype == UCIS_COVERGROUP)  {
    /*Covergroup scope: Loop on all sub-scopes*/
    ucisScopeT subscope = NULL;
    ucisIteratorT iterator = ucis_ScopeIterate(
                             db, scope, -1);
    while(subscope = ucis_ScopeScan(db,
                                    iterator)){
      populateHolesList(db,subscope,&holeslist);
    }
    /*Free the iterator handle*/
    ucis_FreeIterator(db, iterator);

  } else {/*Handle other FCOV scopetypes...*/}

  /*Close DB and return list of holes string*/
  ucis_Close(db);
  return holeslist;
}/*getCoverHoles*/
```

In the above code we defined a C function *getCoverHoles()* to return the list of coverage holes in a scope. The C function would take as arguments the UCISDB file name, as well as the functional coverage scope name of interest, then returns a string holding the list of coverage holes (i.e. uncovered coveritems names) in the given scope. This list can be fed to testbench components of interest to focus on generating the uncovered scenarios. The *getCoverHoles* function does the following:

- Loads and populates an existent physical UCISDB file in memory.
- Finds a matching scope with same name as the *scopename* argument and gets a handle for it.
- Once a matching scope is found, gets its type.
- If the scope type is a Coverpoint or a Cross, this means that scope does not contain child scopes and would only contain coveritems, then it calls the C function *populateHolesList()* to loop on all coveritems (bins) enclosed.
- If the type is a Covergroup, this means that scope contains child scopes (i.e. Coverpoints and Crosses) and hence we have to loop on enclosed scopes and for each call the *populateHolesList()* function.
- Return the list of coverage holes of the scope as a string.

The *populateHolesList()* can implemented be as follows:

```
void populateHolesList(ucisT db,
                       ucisScopeT scope,
                       char** holeslist){
  ...
  /*Add start of scope flag, scope name and type
  to holeslist str*/
  strcat (*holeslist, "%");
  strcat (*holeslist, ucis_GetStringProperty
          (db, scope, -1,UCIS_STR_SCOPE_NAME));
  strcat (*holeslist, ":");
  strcat (*holeslist, scopetypename);
  strcat (*holeslist, ":");

  /*Iterator handle: loop on all enclosed bins*/
  iterator = ucis_CoverIterate (db, scope,
                                UCIS_ALL_BINS);
  while((binindex = ucis_CoverScan (db,
                     iterator))!= -1){
    ucis_GetCoverData(db, scope, binindex,
                  &binname, &cvdata, &srcinfo);
    ...
    /*If bin is not covered.
    Assume 64-bit UCISDB*/
    if (cvdata.data.int64 < cvdata.goal){
      ...
      /*If bin not covered, add to holes list*/
      strcat (*holeslist, binname);
      strcat (*holeslist,"|");/*Bins separator*/
    }
  }
  /*Free the iterator handle*/
  ucis_FreeIterator (db, iterator);
  /*Add end of scope flag to holeslist str*/
  strcat (*holeslist, "%");
}
```

As shown above, *populateHolesList()* takes a scope handle, database handle, and a reference to the *holeslist* string. The function then loops on all bins inside a scope checking if a bin was covered or not. When not covered, the bin name is added to the list of coverage holes. Extra coding may be required to ease the parsing of the returned *holeslist* string (e.g. begin of scope flag, scope name/type, reduce clutter of bin names specially the auto-generated ones to something more meaningful for stimuli generators, etc.).

3.3 Save Functional and Runtime Attributes of a Test

Beside coverage metrics, a test may want to save specific functional and/or runtime metrics for future post-run analysis that would help determine tests quality, cost-benefit with means for improvement, as well as project momentum and general trends. Test name, exit status, simulation time, cpu time, memory footprint, simulation CLI, seed, date, and username are all examples of runtime metrics. Test objective(s), coverage targets met/missed, execution paths,

sequences exercised, and user-defined metrics are all examples of functional metrics. The UCIS defines *attributes* as name-value pairs that may be associated with history nodes, scopes, or coveritems in a UCISDB.

```
typedef enum {
        ATTR_INT,
        ATTR_FLOAT,
        ATTR_DOUBLE,
        ATTR_STRING
} AttrTypeT;
typedef struct {
        AttrTypeT type_; /* Value type */
        int    ivalue;   /* Integer value */
        real   rvalue;   /* Real value */
        string svalue;   /* String value */
} AttrValueT;
import "DPI-C" function void saveAttr (
                    string dbname,
                    string key,
                    output AttrValueT value);
function void test1::extract_phase(uvm_phase
                                    phase);
  AttrValueT value;

  value.type_   = ATTR_STRING;
  value.svalue = "PASSED";
  saveAttr("test1.ucisdb", "TESTSTATUS",
                           value);

  value.svalue = "YES";
  for (int i=0; i<coverscopes.size(); i++)
    if (checkCoverGoalMet("test1.ucisdb",
          coverscopes[i].scope,
          coverscopes[i].coverage_score) == 0)
    begin
      value.svalue = "NO";
      coverholes ={coverholes,
                   getCoverHoles("test1.ucisdb",
                     coverscopes[i].scope)};
    end
  saveAttr("test1.ucisdb", "COVER_TARGETS_MET",
                           value);

  value.svalue = coverholes;
  saveAttr("test1.ucisdb", "COVER_HOLES",
                           value);
  ...
endfunction
```

As shown above, before the test exists (in its post-run phase), a test could save its functional and runtime metrics as attributes. The code above is an example of saving three important attributes. The *TESTSTATUS* attribute holds the functional pass/fail information of the test, this is important for regression summary population and faster time to debug. The *COVER_TARGETS_MET* attribute is linked to what has been presented in previous sections of linking a test to its coverage targets and objectives. A test may pass on the functional level, but may fail to achieve its coverage targets, having the information handy and populated easily could help verification engineer take actions for improvement (e.g. enhance the test sequence library, restructure testbench, write more new directed tests, explore benefits of other technologies like formal verification, graph-based test coverage, etc.). One could also save the remaining coverage holes for more details as shown above. The more important functional and runtime metrics are saved in the UCISDB, the easier the analysis and tracking of regression runs and coverage closure will be. Simulators supporting UCIS should save runtime metrics by default in the UCISDB, allowing verification engineers to focus on saving tests functional metrics.

The C implementation of the saveAttr() method used by the test above is shown below. Since the attributes passed to the method are linked to the test, the method saves them in the test record. Global attributes can be saved in the UCISDB if *NULL* was passed instead of *test* in the *ucis_AttrAdd()* method call below.

```
void saveAttr (const char* dbname,
               const char * key,
               AttrValueT * value){
  ucisT db; /*UCIS in-memory DB handle*/
  ucisAttrValueT value_;
  ucisHistoryNodeT test;
  ucisIteratorT iterator;

  db = ucis_Open(dbname);

 /*Get handle to the test record inside UCISDB*/
  iterator = ucis_HistoryIterate
             (db, NULL, UCIS_HISTORYNODE_TEST);
  while (test = ucis_HistoryScan (db, iterator))
  {
   /*Register the type of the attribute (string,
   int, float, double...)*/
   value_.type = (ucisAttrTypeT) value->type_;

  /*Save the corresponding value of the attr*/
   if (value_.type == UCIS_ATTR_STRING) {
     value_.u.svalue = value->svalue;

   } else if (value_.type == UCIS_ATTR_INT) {
     value_.u.ivalue = value->ivalue;

   } else if((value_.type == UCIS_ATTR_FLOAT) ||
             (value_.type == UCIS_ATTR_DOUBLE)){
     value_.u.dvalue = value->rvalue;
   }
   assert(!ucis_AttrAdd(db,test,-1,
                        key,&value_));
  }
  ucis_FreeIterator (db, iterator);

  /*Save UCISDB after adding the new attribute*/
  ucis_Write(db, dbname, NULL, 1, -1);
  ucis_Close(db);
}/*saveAttr*/
```

4. UCIS POST-RUN APPLICATIONS

After demonstrating UCIS applications at runtime, either to enhance test quality and performance on the fly, or to help identify tests unachieved targets for further post-run analysis, we now present some standalone UCIS applications which can be used to analyze tests and regression runs status, achieved coverage and guide the next steps to improve tests and regression quality and throughput. Applications presented here require UCISDBs generated prior to simulation run termination; a test should save a UCISDB holding its run metrics. Application would be compiled using C compiler (e.g. gcc) then executed on required UCISDBs.

4.1 Reporting Regression and Tests Runs Summary

At the end of a regression run, it is always useful for a verification engineer to provide summary of the regression runs with relevant details about pass/fail tests with failure reasons, tests achieved/did not achieve coverage targets (reporting corresponding coverage holes if any), and general simulation run information.

The standalone application presented below starts by taking all regression UCISDBs, which correspond to tests runs, as input arguments. It then loops on each UCISDB extracting the corresponding test runtime data and attributes. The *ucisTestDataT* is a UCIS built-in struct holding the test status, simulation run cputime, seed, simulation time (and unit), CLI, etc. The application then gets the specific attributes added by the test; as an example below it shows if the test succeeded to meet its coverage targets and the corresponding coverage holes if not.

```
typedef struct {
  ucisTestDataT testdata;
  char *testname, *coverholes, *fail_msg,
       *covertargets, *teststatus,
       *covertargets_met;
  double coverage_score;
} reportDataT;

void testSummaryReport (char * filename) {
  ucisT db;
  ucisHistoryNodeT test;
  ucisIteratorT iterator;
  ucisAttrValueT * value;
  reportDataT data;
  db = ucis_Open(filename);
  iterator = ucis_HistoryIterate (db, NULL,
                       UCIS_HISTORYNODE_TEST);
  while (test = ucis_HistoryScan (db, iterator))
  {
    data.testname = ucis_GetStringProperty(db,
              test, -1, UCIS_STR_TEST_NAME);
    ucis_GetTestData(db, test, & data.testdata);
    /*If coverage targets are met*/
    value = ucis_AttrMatch(db, test, -1,
                        "COVER_TARGETS_MET");
    data.covertargets_met = value->u.svalue;
    /*Coverage holes if any*/
    value = ucis_AttrMatch(db, test, -1,
                        "COVER_HOLES");
     data.coverholes = value->u.svalue;
    /*Coverage score*/
    data.coverage_score=coverageScore(db,NULL);
    /*Test status*/
    value = ucis_AttrMatch(db, test, -1,
                      "TESTSTATUS");
    data.teststatus = value->u.svalue; ...
    reportTestCore(data);
  }
  ucis_FreeIterator (db, iterator);
  ucis_Close(db);
}

int main(int argc, char* argv[]) {
  for (i = 1; i < argc ; i++) {
    testSummaryReport (argv[i]);
  }
  regressionSummaryReport();
}
```

The application reports the summary data for each test then a summary of the whole regression. The implementation of reporting methods (*reportTestCore()* and *regressionSummaryReport()*) is left to the user to define. The format could be ASCII, XML, HTML or in this example we chose to generate a spreadsheet table. Different formats can be used to exchange and interface information with other consumers in the flow. The spreadsheet for the report could look as follows:

TEST	Coverage Targets	CLI Args	Pass/Fail Status	Fail Reason	Fail Time	Coverage Targets Met	Coverage Holes	CPUTime (Sec)	Peak Memory Consumed(MB)
Wishbone_if_compliance	wishbone_cvg	+UVM_TESTNAME=Wishbone_if_compliance	PASS	-	-	YES	-	423.67	550.4
Wishbone_tx_rx_fifo	fifo_statistic, fifo_cornercases	+UVM_TESTNAME=Wishbone_tx_rx_fifo	PASS	-	-	NO	fifo_cornercases:C1, fifo_cornercases:C3,	1890.3	450.5
buffer_desc_1	BdRam_Cvg	+UVM_TESTNAME=buffer_desc_1	FAIL	** FATAL: randomization() of primitive class failure	18349 ns	NO	BdRam_Cvg:wrXadd, BdRam_Cvg:wrXdin, num_bd_cvg:TXBDs, txbd_frm_cvg:under_run, txbd_frm_cvg:retry_cnt, txbd_frm_cvg:retran_limit, txbd_frm_cvg:cs_lost,	120.6	234.4
ethmac_rand_rxtx_test	rx_tx_cvg	+UVM_TESTNAME=ethmac_rand_rxtx_test	PASS	-	-	NO	rxtx_seq_item_cg::rxtx_size_cross::<scenario_type[RXTX],min_sz, min_sz>, <scenario_type[RXTX],	845.5	332.9

Figure 2. Tests and regression report snippet

As shown above, the generated report could help in post-run analysis as follows:

- Provides general information about the test; name, objective, category (e.g. simulation, formal, emulation, positive, negative, etc.), runtime seed used, CLI args and coverage targets.
- Provides pass/fail information of the test. Upon a test failure, it prints out the failing reason and the simulation time at which it occurs. All these attributes can be added to a test's UCISDB as user defined attributes at runtime. This helps in grouping common failures to identify the number of unique issues that need to be fixed in the DUV or testbench.
- Provides information about coverage targets missed (coverage holes) in coverage scopes of interest. This could help a verification engineer to revisit the test for enhancements. Simulation stimuli patterns can be generated easily from coverage holes to be applied at runtime if needed to elevate coverage numbers[9].
- Provides runtime details related to performance (e.g. cputime, peak memory consumed), this could help determining bottleneck tests and provide means for future regression throughput improvements (e.g. long running tests that do not meet their targets may be re-architected, split, or omitted).

[9] This requires visiting the source code, and it assumes that direct correlation between the coverage model and the system input transactions can be established.

- A general regression summary; e.g. number of passing/failing tests w.r.t. total number of tests, number of tests that meet (or do not meet) their coverage targets, total regression time, etc.

4.2 Generate Regression Coverage Summary by Merging Individual Tests Coverage

In order to get insight about overall regression coverage score achieved and compare it w.r.t. the project's verification plan target and objectives, an application is required to merge tests individual coverage results altogether. The resulting information would help answer questions "Are we done?", or suggest additional steps required; e.g. enhance or write new tests to cover uncovered scenarios, or try out new technologies as discussed before.

```
typedef struct { /*UCISDB Carrier for callback*/
  ucisT db;
} dbCarrierT;

void mergeScope (ucisCBDataT* cbdata,
       dbCarrierT * nextdb, ucisScopeT scope_m){
  ...
  char * scopename_m = ucis_GetStringProperty (
   cbdata->db, scope_m, -1, UCIS_STR_UNIQUE_ID);

  ucisScopeT scope2 = ucis_MatchScopeByUniqueID(
           nextdb->db, NULL, scopename_m);

  ucisIteratorT iterator = ucis_CoverIterate
      (cbdata->db, scope_m , UCIS_ALL_BINS);
  while((coverindex_m = ucis_CoverScan (
              cbdata->db, iterator))!= -1){
    /*Get coveritems data for same scope in both
    UCISDBs (master and next UCISdb in list)
    Assuming identical UCISDBs in structure*/
    ucis_GetCoverData(cbdata->db, scope_m,
                  coverindex_m, &binname_m,
                  &cvdata_m, &srcinfo_m);
    ucis_GetCoverData (nextdb->db, scope2,
  coverindex_m, &binname2, &cvdata2, &srcinfo2);
    /*Increment coverages scope of coveritem in
master UCISDB by coverage score in next UCISDB*/
    ucis_IncrementCover (cbdata->db, scope_m,
         coverindex_m, cvdata2.data.int64);
  }
  ucis_FreeIterator (cbdata->db, iterator);
}

ucisCBReturnT callback(void* userdata,
                       ucisCBDataT* cbdata) {
  dbCarrierT* nextdb = (dbCarrierT*) userdata;
  if(cbdata->reason == UCIS_REASON_SCOPE) {
    if (UCIS_COVERGROUP & (scopetype =
    ucis_GetScopeType(cbdata->db,cbdata->obj))){
      /*Covergroup scope loop on all subscopes*/
      ucisScopeT subscope = NULL;
      ucisIteratorT iterator =ucis_ScopeIterate(
                cbdata->db, cbdata->obj, -1);
      while(subscope =ucis_ScopeScan(cbdata->db,
                                  iterator)){
        mergeScope(cbdata, nextdb, subscope);
      }
      ucis_FreeIterator(cbdata->db, iterator);
    }
  }
  return UCIS_SCAN_CONTINUE;
}
```

```
double mergeUcisdb (int argc, char* argv[]){
startScopeName = argv[1];
outfilename    = argv[2];
 /*First UCISDB treated as master DB */
 printf ("* Merging file  %s (Master)\n",
            argv[3]);
 dbmaster = ucis_Open(argv[3]);
 startScope=ucis_MatchScopeByUniqueID(dbmaster,
                         NULL, startScopeName);
 mergenode = ucis_CreateHistoryNode(dbmaster,
           NULL, "TopHistoryNode", outfilename,
            UCIS_HISTORYNODE_MERGE);
/*Master UCISDB as basis for merge output*/
 iterator = ucis_HistoryIterate(dbmaster,
            NULL, UCIS_HISTORYNODE_TEST);
 while(test=ucis_HistoryScan(dbmaster,iterator))
 {
   /*All test nodes are children of mergenode*/
   ucis_SetHistoryNodeParent(dbmaster, test,
                           mergenode);
 }
 ucis_FreeIterator (dbmaster, iterator);

 for (i = 4; i < argc ; i++) {
   /*Loop on all UCISDBs and create
   corresponding test records in master UCISDB*/
   printf("* Merging file %s\n", argv[i]);
   db = ucis_Open(argv[i]);
   iterator = ucis_HistoryIterate(db, NULL,
                    UCIS_HISTORYNODE_TEST);
   while(test = ucis_HistoryScan(db, iterator)){
     testname = ucis_GetStringProperty(db, test,
                 -1, UCIS_STR_TEST_NAME);
     ucis_GetTestData(db, test, &testdata);
     testnode = ucis_CreateHistoryNode(dbmaster,
                  mergenode, testname,
                 argv[i],
                 UCIS_HISTORYNODE_TEST);
     if (testnode) ucis_SetTestData(dbmaster,
              testnode, &testdata);
   }
   ucis_FreeIterator (db, iterator);
   nextdb.db = db;
   ucis_CallBack(dbmaster, startScope,
              callback, &nextdb);
   ucis_Close(db);
 }
 ucis_Write(dbmaster, outfilename, NULL, 1, -1);
 mergescore=coverageScore(dbmaster, startScope);
 ucis_Close(dbmaster);
 return mergescore;
}
/*Prepare Linked lists for input UCISDBs files*/
int main(int argc, char* argv[]) {
  mergescore = mergeUcisdb (argc, argv);
  return 0;
}
```

The application works as follows:

- Take as arguments: (1) the required scope to merge the input UCISDBs upon (i.e. start scope; passing NULL would mean to merge all scopes of input UCISDBs), (2) the output UCISDB holding the merged data, (3) all input UCISDBs to be merged.
- The *mergeUcisdb()* method would take the first input UCISDB as a master UCISDB and uses it as the basis for the merge output. It would loop on all test records in all input UCISDBs and create

corresponding test records as children of a parent merge node record in the merge output UCISDB.

- The method then use the *ucis_CallBack()* to loop on all scopes in the master UCISDB under the start scope. The *callback()* method once spots a *UCIS_COVERGROUP* scope (corresponds to a SystemVerilog Covergroup), it loops on all sub scopes (e.g. Coverpoints and Crosses) and for each call the *mergeScope()* method.
- The *mergeScope()* method loops on all coveritems in the master UCISDB and increments the coverage score of each with the coverage score of the corresponding coveritem in the next input UCISDB.

The application above is simple for demonstration purposes assuming identically structured UCISDBs given as inputs, also the first UCISDB is being treated as a master UCISDB, and hence the merge algorithm would be performed only on the scopes found in the master UCISDB. The merge algorithm applied in this application is a *totals merge*, in which all the coveritems coverage scores in all UCISDBs are aggregated and written in the final UCISDB holding the result. This would be a relatively fast application and would result in a compact UCISDB holding the merge result. The catch here is that no specific information about which test hit which coveritem is retained in the merge output UCISDB, which could be useful for analysis purposes. Such UCISDB can be generated by the *test-associated merge*, which is more complex and hence requires additional coding and resources than the totals merge, and results in a larger output UCISDB due to the extra information. There are many trade-offs in the merging of coverage data that an application needs to understand and give flexibility to the user. Merge modes such as *totals* versus *test-association*, *union* versus *master* mode, coverage types to include, block into system, multithreading, 'or' versus 'anding' flags and attributes, etc.[10]

4.3 Ranking Regression Most Contributing Tests to Coverage

After a regression run, it is often required to value each test in the regression individually according to its contribution to the overall regression coverage. This could be useful to:

- Abandon redundant tests that do not contribute to overall coverage and consume huge amount of resources. This indirectly minimizes resources utilization and boosts regression throughput.
- Identify highly contributing tests in the regression; this helps in constructing an acceptance sanity checking regression subset, which would act as quick and effective subset used for sanitizing recent code changes instead of waiting for days/weeks for a full regression run.
- Ranking can also take into consideration the runtime metrics of tests aiming to boost regression performance.

```
int rankUcisdb(int argc, char* argv[]) {
  /*Get best coverage score of all i/p UCISDBs
  Winner will be basis for subsequent merges*/
  for (i = 2; i < argc ; i++) {
   checkCoverGoalMet (argv[i], argv[1],
                     &score);
   if (score > max_score) {
     max_score        = score;
     base_ucisdb_name = argv[i];
   }
  }
  printf("*** RANKING RESULTS ***\n");
  printf("1. %s: %f\n", base_ucisdb_name,
                      max_score*100);
  ucisdb_pool[0] = base_ucisdb_name;
  ucisdb_pool_size = 1;
  db = ucis_Open(base_ucisdb_name);
  ucis_Write("db","merged.ucisdb", NULL, 1, -1);
  ucis_Close(db);

  /*iterative merge based on winner UCISDB*/
  for(itr=1; itr<=argc-3; itr++) {
    for (i = 2; i < argc ; i++) {
      found = 0;
      for (j = 0; j < ucisdb_pool_size ; j++) {
        if (!strcmp (argv[i], ucisdb_pool[j])){
          found = 1;
          break;
        }
      }
      if (found) continue;
      sprintf(curr_mergefilename,"%d.ucisdb",i);
      score = mergeUcisdb(5, (char*[]) {argv[0],
                  argv[1], curr_mergefilename,
                  "merged.ucisdb", argv[i]});
      if (score > max_score) {
        max_score          = score;
        next_ucisdb_name   = argv[i];
        strcpy (merged_ucisdb_name,
                curr_mergefilename;
      }
    }
    db = ucis_Open(merged_ucisdb_name);
    ucis_Write(db, "merged.ucisdb",
               NULL, 1, -1);
    ucis_Close(db);
    remove(merged_ucisdb_name);
    printf("%d. %s: %f\n", itr+1,
                  next_ucisdb_name, max_score);
    ucisdb_pool[itr] = next_ucisdb_name;
    ucisdb_pool_size +=1;
  }
  return 0;
}

int main(int argc, char* argv[]) {
  return rankUcisdb (argc, argv[]);
}
```

The ranking presented in this application is based on the greatest coverage score a UCISDB can offer w.r.t. the overall aggregated coverage score when all UCISDBs are merged together. The application offers *iterative ranking* which ranks

[10] The list is endless but the merge application is the value that tools like Questa© Verification Management add to the flow.

each individual test UCISDB by performing an iteration of merges on the input UCISDBs.

Iterative ranking is a greedy ranking algorithm, (1) it requires $\binom{N^2+N}{2}$ merges (where *N* is the number of tests) hence can be very time consuming, (2) ranking is performed w.r.t. the entire coverage space, hence it requires identically structured input UCISDBs to work effectively otherwise the ranking result will not be accurate. The *test-associated* ranking performs a single merge of the databases and proceeds to rank based upon a test-associated merge results held in memory, it is much more complex to implement however, it super exceeds iterative ranking in terms of performance[11]. The application presented above shows the simple iterative ranking approach as follows:

- Take as arguments: (1) the required scope to rank the input UCISDBs upon (i.e. start scope; passing NULL would mean to rank based on the entire UCISDB score), (2) all input UCISDBs to be ranked.
- Start by computing the greatest coverage score of all inputs UCISDBs, the outcome UCISDB will be the basis for all future merges (i.e. base UCISDB).
- In an iterative loop of length *(N-2)*, where *N* is the number of input UCISDBs, it performs iterative 1-to-1 merge between the merge base UCISDB and the next UCISDB in the loop, and picks the UCISDB that contributes most to the merge output. It then takes the output of the merge process of iteration *M*, as the basis for the merge process for iteration *M+1*, till all required iterations are executed.

Output of the application can be as follows:

Rank	TEST	Seed	Coverage Score	Contributing
1	ethmac_rand_rxtx_test	17	56.8	YES
2	Wishbone_tx_rx_fifo	20301	72.3	YES
3	buffer_desc_1	1	79.7	YES
4	wishbone_if_compliance	0	85.4	YES
5	reg_read_writes	35	89.3	YES
6	reg_resets	2298773	89.5	Yes
7	ethmac_rand_simple_sanity	1529813	89.6	NO
8	patternset_ip	1856469	89.6	NO

Figure 3. Tests ranking report snippet

As can be seen, from the above ranking report one could put hands on noncontributing tests to overall regression coverage score. This is useful from a verification engineer perspective for inspections and enhancements purposes on these tests, or eliminating them from regression runs to improve the regression throughput.

[11] Again, Questa© Verification Management includes algorithms that use techniques to speed up the ranking of databases, including test association, quick and multi-threading.

5. CONCLUSION

The paper started by giving some background about the verification process challenges for today's complex designs. The paper touched upon tests and testbench quality improvement, single simulation and regression throughput, analysis of verification metrics, faster coverage closure, and project tracking.

The paper makes use of the UCIS and shows how the capabilities of such standard must not be underestimated in terms of the value it could bring to the verification process. It was shown how the UCIS open API can bring a huge value in building applications that will not only help in post-run analysis and tracking, however they can help during runtime as well.

We presented several runtime UCIS applications that would help maximize the simulation throughput, on the fly change tests' runtime behavior upon collected coverage analysis, and register tests specific runtime and functional metrics for post-run track of tests quality. Then we presented several post-run applications which can be used to analyze tests and regression runs status, achieved coverage, and guide the next steps to improve tests quality and regression throughput.

6. ACKNOWLEDGMENTS

The author would like to thank Abigail Moorhouse, Samiran Laha, and Darron May, of Mentor Graphics Corp., for their significant contribution to the UCIS standard and for their help getting some of the applications presented here up and running.

7. REFERENCES

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

[2] Wilson Research Group, *2010 Functional Verification Study*.

[3] IEEE Standard for SystemVerilog, *Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2012, 2012.

[4] PJ Plauger: *The Standard C Library, Prentice Hall*, 1992.

[5] UVM User Manual, uvmworld.org.

APPENDIX A. – HYPOTHETICAL EXAMPLE, SHOWING STIMULI GENERATORS MAKING USE OF COVERAGE HOLES EXTRACTED AT RUNTIME

```
// $Id: fcc_lib.sv,v 1.0 2011/09/13 Ayehia Exp$
//--------------------------------------------------
// Ahmed Yehia ahmed_yehia@mentor.com
// Copyright 2005-2013 Mentor Graphics Corporation
// All Rights Reserved Worldwide
//
// Licensed under the Apache License, Version 2.0
// (the "License"); you may not use this file
// except in compliance with the License. You may
// obtain a copy of the License at
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to
// in writing, software distributed under the
// License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
// either express or implied. See
// the License for the specific language governing
// permissions and limitations under the License.
//--------------------------------------------------

/*Hypothetical example showing difficulties to
close coverage of simple transaction coverage
model with 3 random variables a,b,&d. The
coverage model consists of a coverpoint for each
of rand variable, and a cross between a&b&d.
Normal ways need several thousands of cycles to
close coverage, whereas proposed method can
close coverage in few hundred cycles.
User added code does not exceed 50 lines, while
way to coverage closure shortened dramatically.*/
```

<transaction.svh>

```
/*As shown below no changes required in the
transaction definition except including file that
would be generated/written to apply the proposed
method. Transaction makes use of macros defined in
fcc_macros.svh to define rand variables */
`include "fcc_macros.svh"
import fcc_pkg::*;
class mytrans extends fcc_transaction;
 /*Replace "rand bit[3:0] a,b,d" with a macro
defines rand objs of type and ID passed,  and
queues of same type to hold object cover holes.*/
  `fcc_def_rand(bit[3:0], a)
  `fcc_def_rand(bit[3:0], b)
  `fcc_def_rand(bit[1:0], d)
  covergroup cvg;
    /*Auto-generated bins for A and B CVPs
    e.g. "auto[0]", "auto[1]", "auto[2]"*/
    A: coverpoint a;
    B: coverpoint b;
    D: coverpoint d {
     //User-Defined bins for D CVP
      bins D_1 = {0,1};
      bins D_2 = {2,3};
    }
    A_B_D_CROSS: cross A, B, D;
  endgroup
  function new();
    cvg = new;
  endfunction
  /*Insert/Write code to apply proposed method
  Code can be automatically generated*/
  `include "mytrans__fcc_generated.sv"
endclass
```

<test.sv>

```
/*Few lines were added to the test to start the
proposed method by saving a coverage database then
extracting the list of coverage holes as a string.
A loop is then active, as long as there are more
holes to cover, inside which normal test operation
is cloned with the exception of calling
fcc_randomize() instead of normal SystemVerilog
randomize() for doing the actual transaction
randomization. The fcc_randomize() would be a
simple generic method that loops on all coverage
holes and for each call the core_randomize()
method that would do the real work, transforming
from strings to additional values constraints: do
the transaction generation(i.e. randomization).*/
import "DPI-C" function string getCoverHoles
                       (string dbname,
                        string scopename);

`include "uvm_macros.svh"
import uvm_pkg::*;
import fcc_pkg::*;

class test1 extends uvm_test;
  `uvm_component_utils (test1)
   mytrans tr;
   real coverage_momentum, threshold_momentum=1,
        normal_max_method_score;
   int num_of_trans, goal_met;
   coverscopesT coverscopes[];
   byte char;

   function new(string name,uvm_component parent);
     super.new (name, parent);
     tr = new;
     coverscopes = new [1];
     coverscopes[0].scope =
                  "/19:my_pkg/11:mytrans/12:cvg";
   endfunction

   task run_phase(uvm_phase phase);
     phase.raise_objection (this, "test1");
     `uvm_info("test1", "NORMAL TEST RUN: SEEKING
COVERAGE CLOSURE!", UVM_LOW);
     //Coverage monitor loop
     while (1) begin
      //Drive "tr" to DUV and sample Covergroup.
       assert (tr.randomize());
        tr.cvg.sample();
        num_of_trans += 1;

        coverscopes[0].coverage_score =
                       $get_coverage();
        coverage_momentum =
              coverscopes[0].coverage_score /
              num_of_trans;
        `uvm_info("test1", $sformatf("ITERATION
#%0d, COVERAGE=%0f, MOMENTUM=%0f", num_of_trans,
coverscopes[0].coverage_score, coverage_momentum),
UVM_HIGH);
        if (coverscopes[0].coverage_score < 100)
         begin
          //compute momentum when goal is not met
          if(coverage_momentum <threshold_momentum)
          begin
            $coverage_save_mti("test1.ucisdb");
             coverscopes[0].coverholes=getCoverHoles
                            ("test1.ucisdb",
                             coverscopes[0].scope);
            break;
          end
         end else begin goal_met = 1; break; end
     end
     if (!goal_met) begin
       `uvm_info("test1", "ADVANCED TEST RUN:
SEEKING COVERAGE CLOSURE!", UVM_LOW);
```

```
      /*While coverage holes exist.
      Extract next coverage hole and randomize
      transaction accordingly*/
      while(coverscopes[0].coverholes!="") begin
        tr.fcc_randomize(
                     coverscopes[0].coverholes);
        //Drive "tr" to DUV, sample Covergroup.
        tr.cvg.sample();
        num_of_trans +=1;
        coverscopes[0].coverage_score =
                                 $get_coverage();
        `uvm_info("test1", $sformatf("ITERATION
#%0d, COVERAGE=%0f", num_of_trans,
coverscopes[0].coverage_score), UVM_HIGH);
      end
    end
    phase.drop_objection (this, "test1");
  endtask
endclass

module top;
  initial
    run_test();
endmodule
```

<fcc_macros.svh>

```
/*Library of handy macros re-used to smoothen the
transformation from string bin names to meant
object values and defines randomization process
to generate them.
Written once and re-used in each testbench*/
`define fcc_def_rand(T, ID) \
  rand T ID;\
  T ID``_fcc_q [$];\
  int ID``_fcc_transition_cnt;

`define fcc_update_queue(ID,VAL)\
  ID``_fcc_q = VAL;

`define fcc_constraint(ID)\
  ID inside {ID``_fcc_q};

`define fcc_do_rand(ID)\
  assert (this.randomize() with \
  {`fcc_constraint(ID)});

`define fcc_do_rand3(ID1, ID2, ID3)\
  assert (this.randomize() with \
  {`fcc_constraint(ID1)\
   `fcc_constraint(ID2)\
   `fcc_constraint(ID3)});
```

<fcc_pkg.sv>

```
/*Package defining transaction (class)
infrastructure that can be utilized by
transactions just importing this package. Written
once and re-used in each testbench.*/
package fcc_pkg;
 import uvm_pkg::*;
 class fcc_transaction extends uvm_sequence_item;
   ...
   virtual function void fcc_randomize (ref string
                                       holes_list);
     ...
     while ... begin
        //Loop on coverage holes string to extract
        //next hole and corresponding scope name
        //Then pass them to core_randomize() to do
        //the actual randomization/stim generation
        ...
       core_randomize(scopename, hole);
       break;
       ...
     end
     ...
  endfunction
  /*User to implement. Represents the mapping from
  The bin name string to the meant value to be
  applied to corresponding object.*/
  virtual function void core_randomize (
    string scope_name, string hole,
    bit coming_from_cross=0);
  endfunction
 endclass
endpackage
```

<mytrans_fcc_generated.sv>

```
/* This code generates an additional constraint
applied to the next transaction generation that
corresponds to a coverage hole of a corresponding
object.  The code makes use of macros introduced
above making it compact, straightforward and has
direct correlation with the coverage model
definition; hence can be automatically
generated.*/
function void core_randomize (string scope_name,
           string hole, bit coming_from_cross=0);
  byte char;
  case (scope_name)
    //Auto-generated bin name clutter was
    //removed by C code to something like "0"
    //making atoi() all what one needs to
    //transform bin name to meant value.
    "A": begin
      `fcc_update_queue(a, {hole.atoi()})
      if (!coming_from_cross) `fcc_do_rand(a)
    end
    "B": begin
      `fcc_update_queue(b, {hole.atoi()})
      if (!coming_from_cross) `fcc_do_rand(b)
    end
    "D": begin
      //User defined bins for Coverpoint D needs
      //special handling.
      case (hole)
        "D_1": `fcc_update_queue(d, {0,1})
        "D_2": `fcc_update_queue(d, {2,3})
      endcase
      if (!coming_from_cross) `fcc_do_rand(d)
    end
    "A_B_D_CROSS": begin
      //For a cross bin, loop on bins of
      //enclosed CVPs, and update queues of each
      //according to order of definition.
      string s;
      int cur_item_num, cnt;
      while((char = hole.getc(cnt)) != 0) begin
        cnt++;
        if (char != ",")//Mark cross separators
          s = {s, string'(char)};
        else begin
          case(cur_item_num)
            0: core_randomize("A", s, 1);
            1: core_randomize("B", s, 1);
            2: core_randomize("D", s, 1);
          endcase
          cur_item_num++;
          s="";
        end
      end
      `fcc_do_rand3(a,b,d)
    end
  endcase
endfunction
```