Mentor®
A Siemens Business

UCDB API Reference

Software Version 2020.4

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Table of Contents

Chapter 1
Introduction to the UCDB API 9
UCDB Data Hierarchy. 11
Scopes and Coveritems. 11
Design Unit Scopes. 12
UCDB Scope Types 13
UCDB Data Models. 15
Code Coverage Roll-Ups in Design Units and Instances 15
Statement Coverage 15
Branch Coverage. 17
Expression and Condition Coverage. 23
Finite State Machine Coverage. 26
Toggle Coverage. 27
Group Data Model 33
SVA and PSL Covers 36
Assertion Data. 37
SystemVerilog Covergroup Coverage 42
Covergroup With a Cross 42
Sparse Cross Bin Representation 45
CROSSSBINIDX and CROSSUBINIDX 46
Covergroup in Package With Multiple Instances. 48
Covergroup in a Class (Embedded Covergroup) 51
Design Units 54
Test Data Records and History Nodes 55

Chapter 2UCDB Use Cases 59
UCDB Access Modes 59
Error Handling 60
Traverse a UCDB in Memory 61
Read Coverage Data 62
Find Objects in a UCDB 65
Increment Coverage. 67
Remove Data From a UCDB 69
User-Defined Attributes and Tags in the UCDB. 71
Traverse from Testplan to Coverage Data with Tags 75
File Representation in the UCDB 77
Addition of New Data to a UCDB. 82
Test Data Records 92
Create a UCDB from Scratch in Memory 93
Read-Streaming Mode 94
Write-Streaming Mode 96

Chapter 3UCDB in Questa and ModelSim. 99

UCDB in the Tool Architecture 99
Using the mti_AddUCDBSaveCB FLI Callback 100
Questa Compatibility 103

Chapter 4
UCDB API Functions 105

Source Files 105
Error Handler 111
Tests 113
Databases and Database Files 121
User-Specified Attributes 127
Scopes 130
Coverage and Statistics Summaries 157
Coveritems 167
Toggles. 177
Groups 180
Tags 183
Formal Data 187
Test Traceability 197

Appendix A
UCDB Organization 201

Test Section 201
Coverage Section 203
Scope Nodes 203
Coveritems 203
Nesting Rules 204
Attributes 208

Appendix B
UCDB Diff BNF 217

UCDB Diff BNF Syntax 217

Glossary

End-User License Agreement
with EDA Software Supplemental Terms

List of Figures

Figure 1-1. Basic Design/Coverage Hierarchy 12
Figure 1-2. Design/Coverage Hierarchy With Design Units 14
Figure 1-3. Data Model for Verilog Statements 16
Figure 1-4. Data Model for Verilog Statements in Generate Blocks. 17
Figure 1-5. Data Model for a Verilog if-else-if 19
Figure 1-6. Data Model for a VHDL if-elsif 21
Figure 1-7. Data Model for a case Statement 22
Figure 1-8. Data Model for an Expression 24
Figure 1-9. Data Model for a Finite State Machine 27
Figure 1-10. Data Model for a VHDL Integer Toggle. 29
Figure 1-11. Data Model for an Enum Toggle 30
Figure 1-12. Data Model for an Extended Register Toggle. 31
Figure 1-13. Data Model for a Connected Net Toggle 33
Figure 1-14. Data Model for a Group 35
Figure 1-15. Data Model for SVA and PSL Cover Directives 37
Figure 1-16. Data Model for Assertions (With Fail Count Only) 38
Figure 1-17. Data Model for an Assertion (With All Counts). 39
Figure 1-18. Data Model for an Immediate Assertion With Pass/Fail Counts 41
Figure 1-19. Data Model for a Cross 44
Figure 1-20. Data Model for a Covergroup (With Per-Instance Coverage) 50
Figure 1-21. Data Model for an Embedded Covergroup 53
Figure 1-22. Data Model for a Testplan With Linked Coverage 57
Figure 3-1. Questa and the UCDB Save FLI Callback 100

List of Tables

Table 1-1. Cross Bin Index Attributes 47
Table 4-1. Values for num_coveritems Dependent on Coverage Type 159
Table A-1. Fields of a Test Record 201
Table A-2. Attributes of a History Node 202
Table A-3. Nesting Rules Enforced by UCDB 204
Table A-4. UCDB Defined Attributes 208
Table A-5. UCDB Defined Objects 214

Chapter 1
Introduction to the UCDB API

UCDB API is an application programming interface for the Unified Coverage Database included in the Questa® SIM and ModelSim® SE™ products. The UCDB and its API are completely independent of Questa and ModelSim, however UCDBs are easily created with these tools. In this document, the term *Questa* refers to both the Questa and the ModelSim SE systems.

Questa software uses the UCDB API for saving, reading, reporting on and merging UCDB format databases. The Questa GUI features are based on the UCDB API as are the command-line interface features in the Coverage View mode:

```
shell prompt> vsim -viewcov ucdb_file
```

Use the coverage save command to create Questa UCDB format databases and use the vcover commands to externally process UCDBs. For simple tasks such as generating a coverage report or merging coverage data, use the corresponding Questa tool features. Use the UCDB API for more complex tasks such as the following.

- Importing data into a UCDB or Questa database from another source
- Exporting data to a database that has a format not supported by Questa (for example, an SQL database or a graphing package)
- Analyzing coverage data in a way not supported by any tool
- Loading coverage data into a UCDB from a VPI application linked with Questa (that will be saved by Questa)

Use the C-based UCDB API to do the following.

- Read from and write to UCDBs.
- Create a UCDB.
- Add data to an existing UCDB.
- Traverse and analyze a UCDB, specifically with the read API.

The UCDB API library supports both memory efficient modes (read- or write-streaming modes) and a fully-populated data model (in-memory mode).

Find the UCDB API library in these locations.

- *<install_dir>/questasim/platform/libucdb.a*. (UNIX).

- *<install_dir>/questasim/platform/ucdb.lib* (Windows).

Find the annotated header file in *<install_dir>/questasim/platform/ucdb.h*.

Find examples illustrating how to compile various UCDB API applications in *<install_dir>/ questasim/platform/ucdb/*.

UCDB Data Hierarchy 11
UCDB Data Models 15

UCDB Data Hierarchy

This section provides information about scopes and coveritems, design unit scopes, and UCDB scope types so that you have a good understanding of the structure of a UCDB file.

Scopes and Coveritems 11
Design Unit Scopes. 12
UCDB Scope Types 13

Scopes and Coveritems

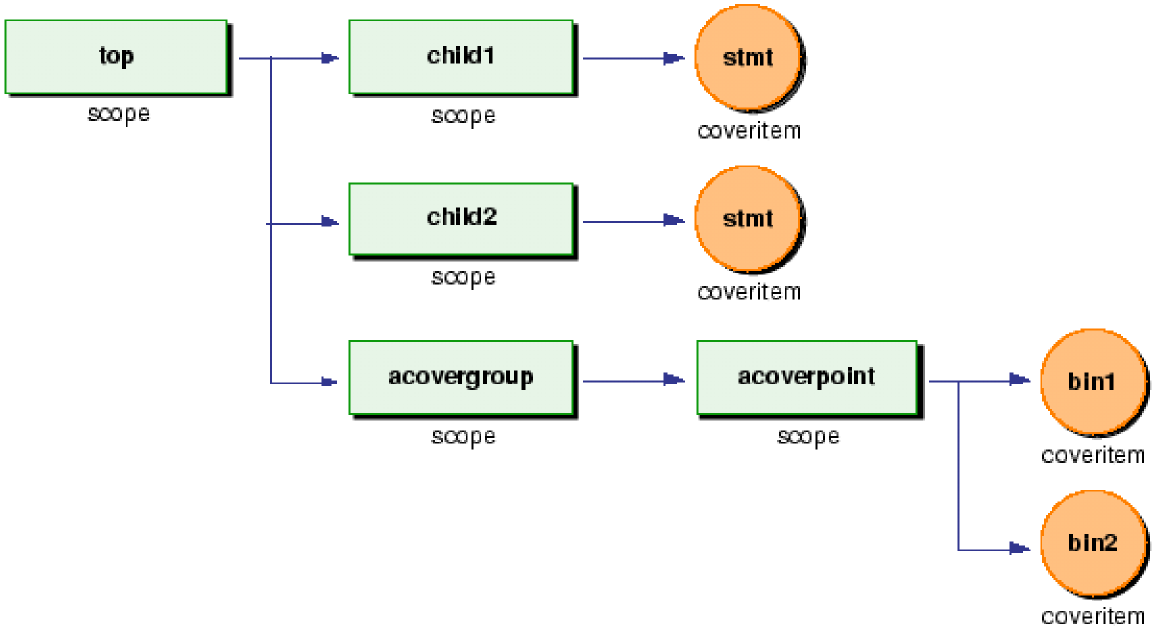
Designs and testbenches are hierarchically organized. Design units (Verilog modules or VHDL entity/architectures), testplans, and coverage data (of which the SystemVerilog covergroup is the best example) can all be hierarchical. Therefore, the UCDB needs some general way to store hierarchical structures.

The UCDB has *scopes* (also referred to as *hierarchical nodes*), which store hierarchical structures (that is, elements of a database that can have children).

The UCDB stores coverage data and assertion data as *counters*, which indicate how many times something happened in the design. For example, they 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.

Tree models of hierarchical organization are central to the UCDB. Figure 1-1 is an illustration of a simple hierarchy.

Figure 1-1. Basic Design/Coverage Hierarchy



Design Unit Scopes

For representing an HDL design, a simple hierarchy is not sufficient. The design units must also be represented.

For example, the SystemVerilog code that corresponds to the tree in Figure 1-1:

```
module top;
   int i;
   covergroup acovergroup;
      acoverpoint: coverpoint i {
         bins bin1 = { 0 };
         bins bin2 = { 1 };
      }
   endgroup
   acovergroup acovervar = new;
   submodule child1();
   submodule child2();
endmodule
module submodule;
   initial $display("hello from %m");
endmodule
```

The scopes top, child1, and child2 represent the module instances of this design hierarchy, but the design units (SystemVerilog modules) are not included.

In a UCDB created by Questa with code coverage, there will be code coverage associated with the design unit. This is the union of code coverage from the instances of the design unit. This is calculated by the kernel, and because it is available immediately from the kernel, it is stored directly in the UCDB. This requires that the UCDB store another scope to correspond to the design unit.

Questa also stores source file information with the design unit. (This is not a requirement of a UCDB, but happens to be the case when one is created from Questa.)

From each module instance scope, its corresponding design unit may be accessed; in fact, the design unit must exist prior to creating the instance.

UCDB Scope Types

Because the UCDB needs to distinguish between module instances, design units, and even other scopes like those for covergroups and coverpoints, the UCDB has a scope type associated with every scope. This scope type is the C type ucdbScopeTypeT.

Scope types are in these categories (found in *ucdb.h*):

- **HDL scope —** These are the basic building blocks of the design hierarchy, or named scopes (in the true HDL sense, rather than the UCDB sense) in the design.
- **Design unit scope —** These must be provided for those HDL scopes which have corresponding design units.
- **Cover scope —** These are used to introduce hierarchy in coverage objects, essentially to group them together.
- **Group scope —** These are used to maintain bus structures for supporting part selects and supporting a general bus data model.
- **Testplan scope —** A scope to represent part of a testplan hierarchy; this is unique because it can only have children that are other testplan scopes.

These relationships must exist between HDL scopes that are instances of a given design unit scope:

- UCDB_INSTANCE has a corresponding UCDB_DU_MODULE or UCDB_DU_ARCH scope as its design unit.
- UCDB_PROGRAM has a corresponding UCDB_DU_PROGRAM scope as its design unit.
- UCDB_PACKAGE has a corresponding UCDB_DU_PACKAGE scope as its design unit. Although VHDL and SystemVerilog do not have actual instances of packages in the language, tools like Questa do represent a package twice: the UCDB_PACKAGE corresponds to the top-level node in the instance tree, and UCDB_DU_PACKAGE corresponds to the definition of the package.

- UCDB_INTERFACE has a corresponding UCDB_DU_INTERFACE scope as its design unit.

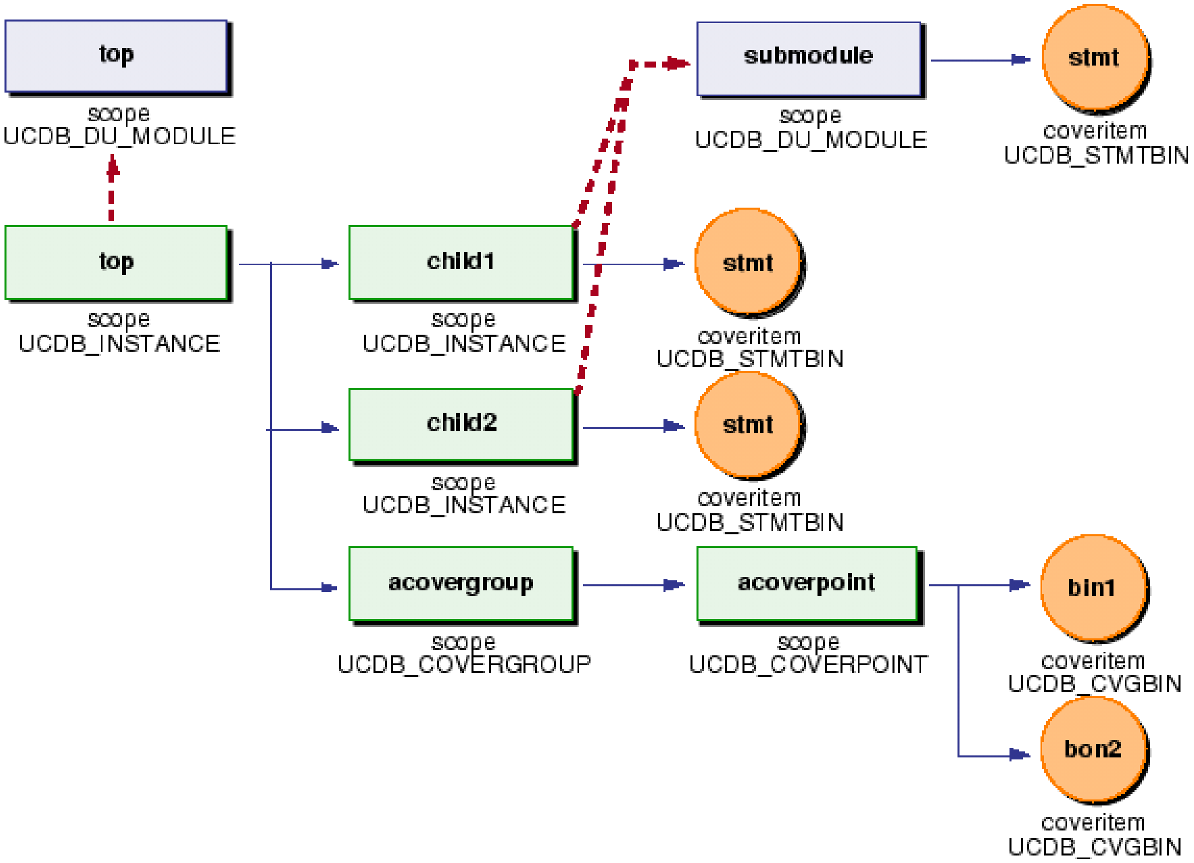
Figure 1-2 revisits the hierarchy of Figure 1-1 and shows how design unit scopes exist to represent the SystemVerilog code.

Figure 1-2 shows the ucdbScopeTypeT values for the scopes, as well as coveritem types. This figure also indicates links from the HDL scopes to the design unit scopes as red dashed lines.

The design unit scopes (UCDB_DU_MODULE in this case) have no special relationships among them; they are not really part of design hierarchy, though they represent a crucial part of the design.

In this example, the statement coverage coveritem (UCDB_STMTBIN) exists in both module instances (/top/child1 and top/child2) as well as the design unit scope (submodule). This example shows one of the uses of the design unit scope: not only does it enable you to determine that child1 and child2 are instances of the same module, but any design-unit-wide data can reside *inside* the design unit scope.

Figure 1-2. Design/Coverage Hierarchy With Design Units



UCDB Data Models

The UCDB API is a general one that creates certain objects – such as scopes, coveritems, and test data records – with certain names, types, and attributes. Use of this API allows for the creation of many different potential data models.

The data models are important because they capture assumptions about how Questa creates a UCDB data structure for a given kind of coverage. Other tools might be able to read and make sense of different data structures, but Questa will not.

The UCDB API is more general than Questa: you can create many different kinds of coverage hierarchies through the API, but only a small subset of those will be valid input to Questa.

A minimum number of assumptions must exist in order for a UCDB to be read by Questa.

Code Coverage Roll-Ups in Design Units and Instances 15
Statement Coverage 15
Branch Coverage 17
Expression and Condition Coverage 23
Finite State Machine Coverage 26
Toggle Coverage 27
Group Data Model 33
SVA and PSL Covers 36
Assertion Data 37
SystemVerilog Covergroup Coverage 42
Design Units 54
Test Data Records and History Nodes 55

Code Coverage Roll-Ups in Design Units and Instances

Questa creates code coverage roll-ups (the aggregation of design-unit-based coverage from instances of those design units) implicitly when the database is loaded into memory.

One caveat is when you access the UCDB using read-streaming mode, the nature of the storage cannot be hidden, because read-streaming mode reflects exactly what is laid out on disk. In that case, coverage never appears underneath design units.

Statement Coverage

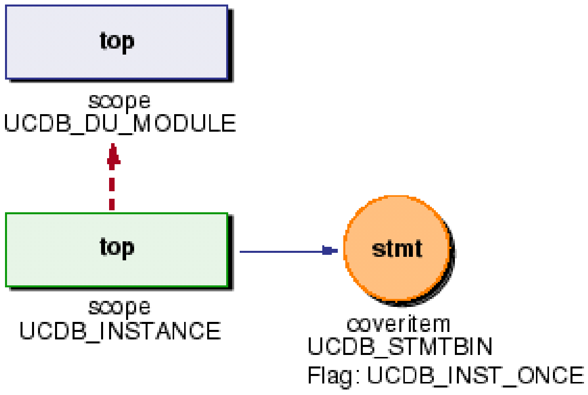
The UCDB reports statement coverage data of your design as a coveritem with no hierarchy.

Simple Statement Coverage

You can find this Verilog example in *<install_dir>/examples/ucdb/userguide/data-models/statement/*, where the file *test.v* is:

```
module top;
   initial $display("hello world");
endmodule
```

Figure 1-3. Data Model for Verilog Statements



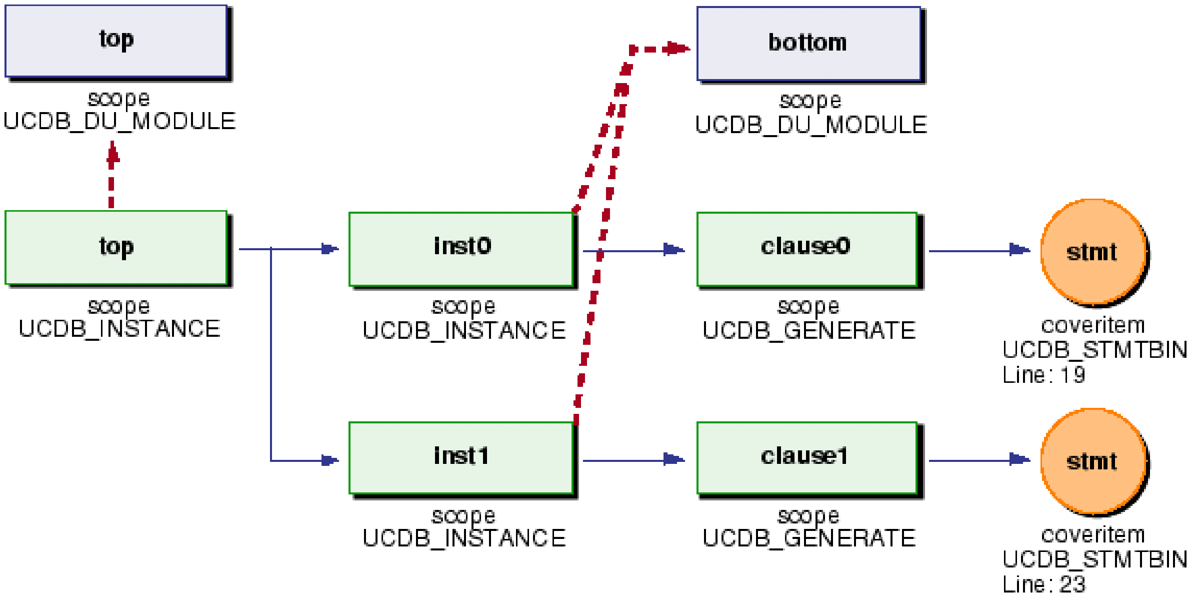
The statement bin does not appear with the design unit also; this is because of the UCDB_INST_ONCE optimization described in the section “Design Units” on page 54.

Statement Coverage With Generates

You can find this Verilog example in *<install_dir>/examples/ucdb/userguide/data-models/statement-generate/*.

```
module top;
   bottom #0 inst0();
   bottom #1 inst1();
endmodule
module bottom;
   parameter clause = 0;
   if (clause == 0)
   begin: clause0
      initial $display("hello from %m");     //line 19
   end
   else
   begin: clause1
      initial $display ("hello from %m");    // line 23
   end
endmodule
```

Figure 1-4. Data Model for Verilog Statements in Generate Blocks



The data model illustrates key characteristics about Verilog statements in generate blocks:

- The UCDB contains UCDB_GENERATE scopes for the generate blocks. These must exist even if the generate block does not have a name that you generated (the "begin: label" constructs in the code example). Having different generate blocks for different scopes would handle the case of the for-generate where different blocks correspond to the same line of source.
- The statements appear inside the generate scopes as well as the design unit scopes. In Figure 1-4, the line number associated with the statement is shown to distinguish between the two statements.
- While the code coverage from generate blocks could be merged into the instance – for example, having another set of merged statement coveritems as children of the UCDB_INSTANCE scopes in this example – that is not a requirement of the data model. Questa does this aggregation in real time, so it never stores any redundant data with the instances.

Note

This real-time aggregation is inconsistent with design units, where redundant data is stored.)

Branch Coverage

Branch coverage is based on the common coding styles of Verilog if-else, VHDL if-elsif-else, and Verilog and VHDL case statements.

- Verilog if-else — A single UCDB_BRANCH scope has two coveritems, one each for the if and else branches.
- VHDL if-elsif-else — There are as many coveritems as the if-cascade has clauses.
- Verilog and VHDL case statements — There is one coveritem per value in the case statement.

Additionally, branch coverage has extra information in the scope:

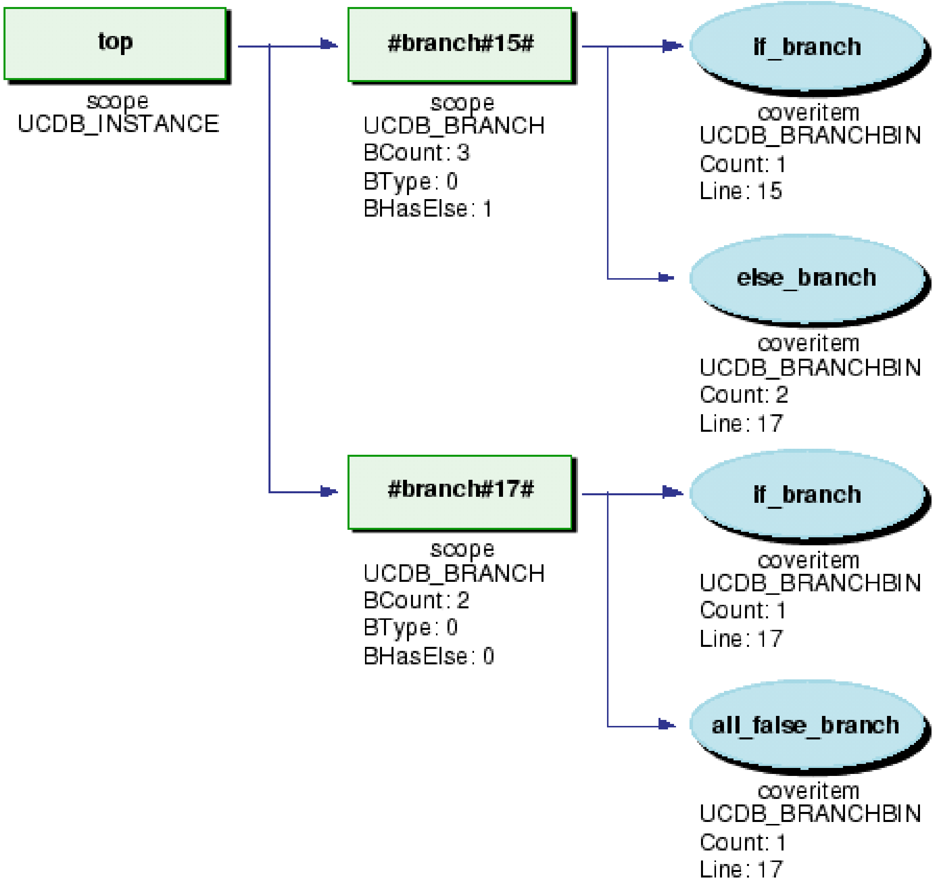
- BCOUNT attribute — Shows the total number of times the test was executed. This is useful if the branch does not have an “else” clause.
- BTYPE attribute — Distinguishes between branch and if-else
- BHASELSE attribute — Distinguishes between if-else branches having an else and those that do not.

Branch Coverage of Verilog if-else

You can find this SystemVerilog example in <*install_dir*>*/examples/ucdb/userguide/data-models/branch-vlog-if/*.

```
module top;
        bit x = 0;
        bit y = 0;
        always @(x or y) begin
                if (x)                             //line 15
                        $display("x is true");
                else if (y)                        //line 17
                        $display("y is true");
        end
        initial begin
                #1; x = 1;
                #1; x = 0;
                #1; y = 1;
        end
endmodule
```

Figure 1-5. Data Model for a Verilog if-else-if



In Figure 1-5, the design unit is omitted. This data model drawing also indicates coverage counts. The data model’s basic components are described as follows:

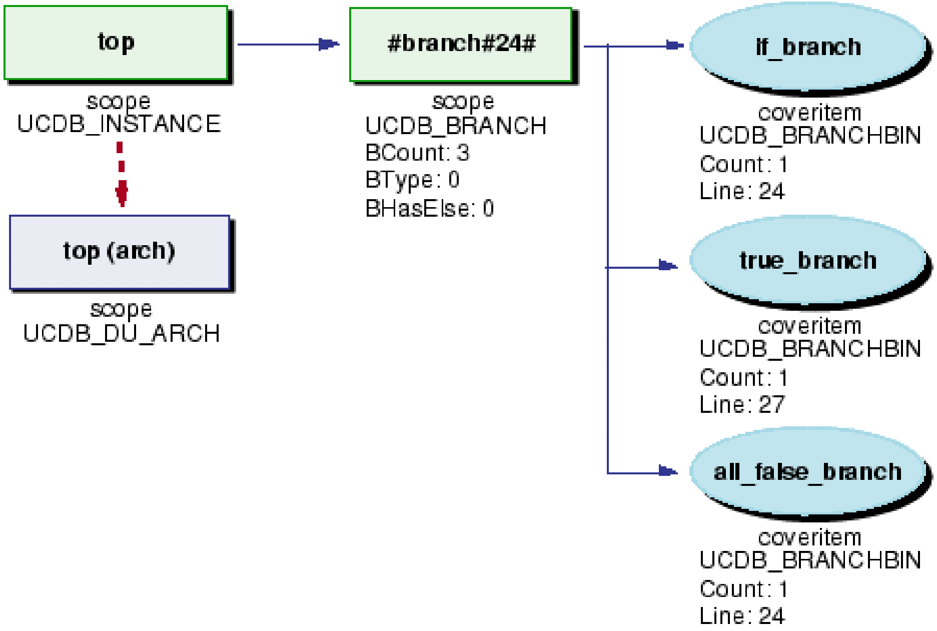
- UCDB_BRANCH scopes have names according to the type of coverage and line number.
- BCOUNT is the sum of if and else counts (even if the “else” is lacking, as in the line 7 branch.)
- BTYPE is 0 for these cases to indicate an “if” as opposed to a “case” statement.
- BHASELSE is 0 for the line 7 branch to indicate that it does not have an else clause.
- The coveritem if_branch is for the true clause of the branch.
- The coveritem else_branch is for the false clause of the branch if it has an explicit “else”.
- The coveritem all_false_branch is for the missing else.

Branch Coverage of VHDL if-elsif-else

You can find this VHDL example in *<install_dir>/examples/ucdb/userguide/data-models/branch-vhdl-if/*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use std.textio.all;
entity top is end;
architecture arch of top is
    signal x : std_logic := '0';
    signal y : std_logic := '0';
    begin
    branch: process
        variable myoutput : line;
    begin
        wait until x'event or y'event;
        if (x = '1') then                              // line 24
            write(myoutput,string'("x is true"));
            writeline(output,myoutput);
        elsif (y = '1') then                           // line 27
            write(myoutput,string'("y is true"));
            writeline(output,myoutput);
        end if;
    end process branch;
    drive: process
    begin
        wait for 10 ns;
        x <= '1';
        wait for 10 ns;
        x <= '0';
        wait for 10 ns;
        y <= '1';
        wait;
    end process drive;
end architecture;
```

Figure 1-6. Data Model for a VHDL if-elsif



This is the VHDL branch to correspond to the Verilog version (Figure 1-5). In this VHDL if-elsif example, the design unit is shown to illustrate the difference between VHDL and Verilog design units: the scope type is different, and the architecture name follows the entity name in parenthesis. These diagrams omit the work library name, which varies depending how the module or architecture was compiled. Technically, the work library name is part of the design unit name in the UCDB too.

The most obvious difference is that there is a single UCDB_BRANCH scope rather than multiple ones. This is because VHDL has the "elsif" syntax that enables a branch to have multiple paths rather than just two paths. The VHDL and Verilog branches share the following common features:

- The first branch coveritem is called if_branch.
- The last coveritem is called all_false_branch if there is no explicit else.
- If there were an explicit else, the last coveritem would be called else_branch.
- The attributes with the UCDB_BRANCH scope carry the same meanings.

Some differences between the VHDL and Verilog branches:

- Coveritems to correspond to elsif branches are called true_branch.
- The UCDB_BRANCH scope may have many coveritem children to correspond to all the "elsif" branches in the VHDL if construct.

Case Statements

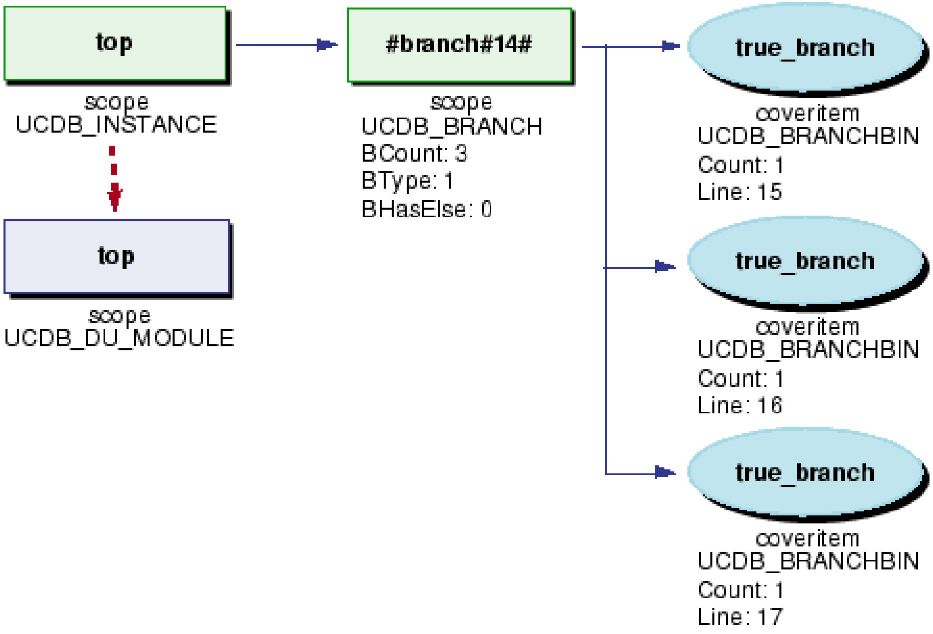
You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/branch-case/*.

```
module top;
    int x = 0;
    always @(x)
        case (x)
            1:          $display("x is 1");                 // line 15
            2:          $display("x is 2");                 // line 16
            default:    $display("x is neither 1 nor 2");   // line 17
        endcase
    initial begin
        #1; x = 1;
        #1; x = 2;
        #1; x = 3;
    end
endmodule
```

This is very similar to the if-elsif construct. The key difference is that the BTYPE attribute has value 1, and that all the coveritems are named true_branch.

Because of similar naming, there is no way in the data model to distinguish between the explicit values in the case statement and the default value. However, there are differences in the line numbers stored with the coveritems, so you could identify the difference from the source files if available.

Figure 1-7. Data Model for a case Statement



Expression and Condition Coverage

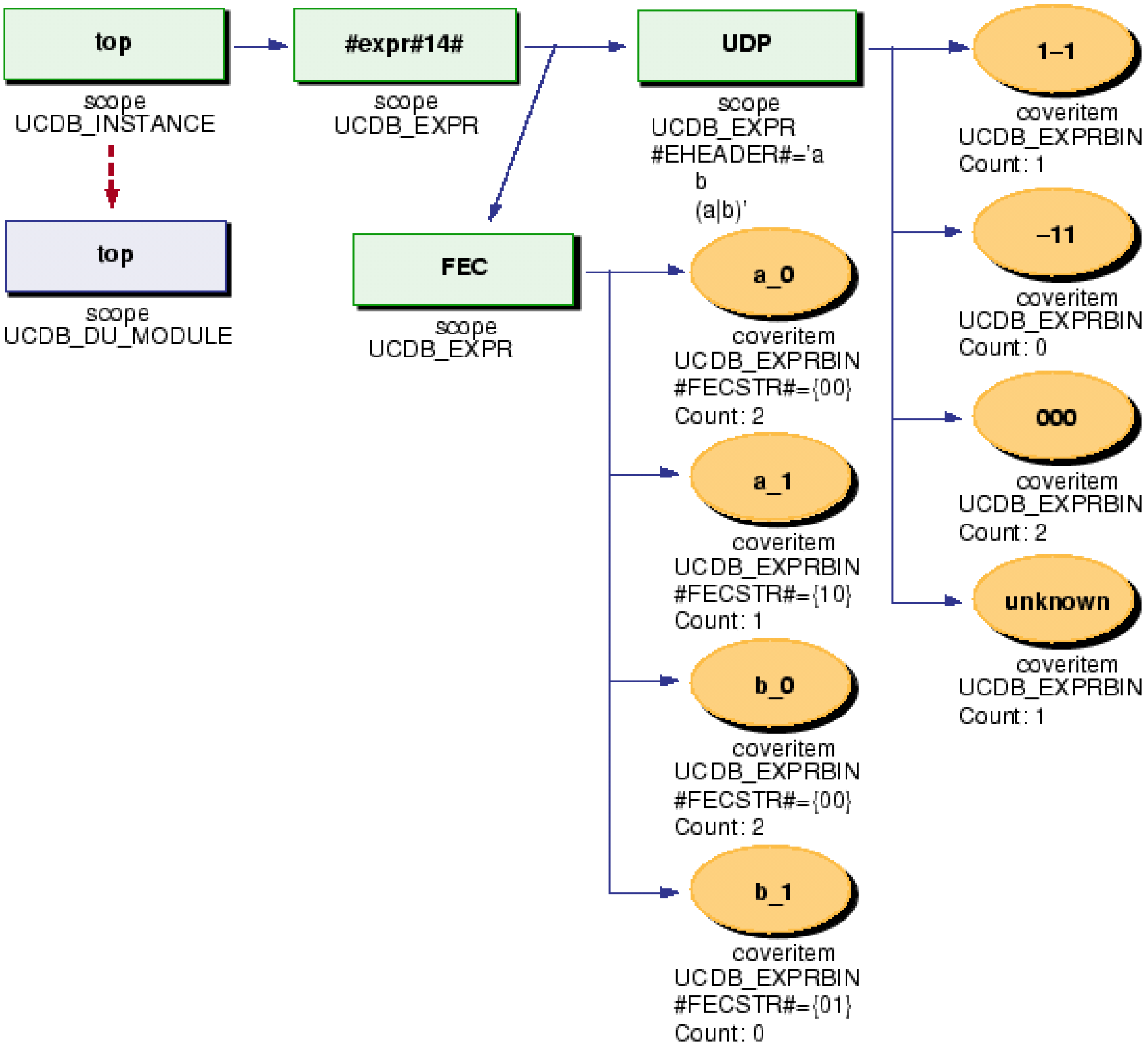
The data model represents both expression coverage, the truth table coverage for an expression used to drive a continuous assignment, and condition coverage, the truth table coverage for an expression in a branch.

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/expr-cond/*.

```
module top;
    logic a = 0;
    logic b = 0;
    assign c = (a|b);               // line 14
    always @(c)
        if (a || b)                 // line 16
            $display("a or b");
    initial begin
        #1; a = 1;
        #1; a = 0;
        #1; a = 1'bx;
    end
endmodule
```

The example is configured by default for expression coverage only. It can be configured for either expression coverage, condition coverage, or both. Expression coverage is for line 14; condition coverage is for line 16.

Figure 1-8. Data Model for an Expression



The data model for expression/condition coverage is split into two styles, each represented simultaneously by default. (There is a way to turn off FEC-style coverage, with the -nocoverfec switch to vopt, for example.)

- The UDP-style coverage is underneath the node named UDP. UDP stands for “user-defined primitive” which really means “truth table.” Verilog UDPs use a truth table syntax in their specification and the names of the UDP-style coverage bins are similar to Verilog UDP row specifications.
- FEC-style coverage is also a truth-table-based coverage, but of a different kind of truth table. FEC stands for “focused expression coverage.” While a UDP-style truth table is generated to have a minimal number of rows, the FEC-style truth table considers each input independently, where each row in the truth table corresponds to a change in a particular input, where that input affects the output. Complete coverage in FEC guarantees that each input changed. FEC is also sometimes called MCDC (modified condition decision coverage).

If this test case were configured for condition coverage instead, the following differences would exist.

- UCDB_COND scope type instead of UCDB_EXPR.
- UCDB_CONDBIN coveritem type instead of UCDB_EXPRBIN.
- EHEADER would be the same except for the last line: “(a || b)”.
- Difference in line numbers.
- Difference in the coverage enabled flags in the design unit.

UDP-Style Expression and Condition Coverage

For an illustration of how the UDP sub-tree data model corresponds to UDP-style expression coverage, view the report generated by Questa.

```
Line       4 Stmt    1     assign c = (a|b);
Expression totals: 2 hits of 3 rows = 66.7%
Truth Table:         a
                     |b
                hits ||(a | b)
  Row   1:         1 1-1
  Row   2:   ***0*** -11
  Row   3:         2 000
  unknown:         1
```

The columns of the truth table are stored with the EHEADER attribute with the expression scope. This is a newline-separated string. The coveritem names correspond literally to the rows of the truth table: “1-1”, “-11”, “000”, and “unknown”. The “unknown” coveritem does not contribute to coverage; its presence is necessary for the report only.

FEC-Style Expression Condition Coverage

The FEC sub-tree data model is explained by the Focused Expression View portion of the report:

```
Line       14 Item    1     assign c = (a|b);
Expression totals: 3 hits of 4 rows = 75.0%
     Rows:      hits          Fec Targets       Matching input patterns
  --------  --------      --------------        ----------------------
  Row   1:         2                 a_0                        { 00 }
  Row   2:         1                 a_1                        { 10 }
  Row   3:         2                 b_0                        { 00 }
  Row   4:   ***0***                 b_1                        { 01 }
```

This is very similar to the UDP style data model, except that the bin names are “Fec Targets” – meaning, the specific input transition represented by the row. With the “-fecanalysis” option, the report shows matching input patterns, which are associated as the attribute “#FECSTR#” with each bin.

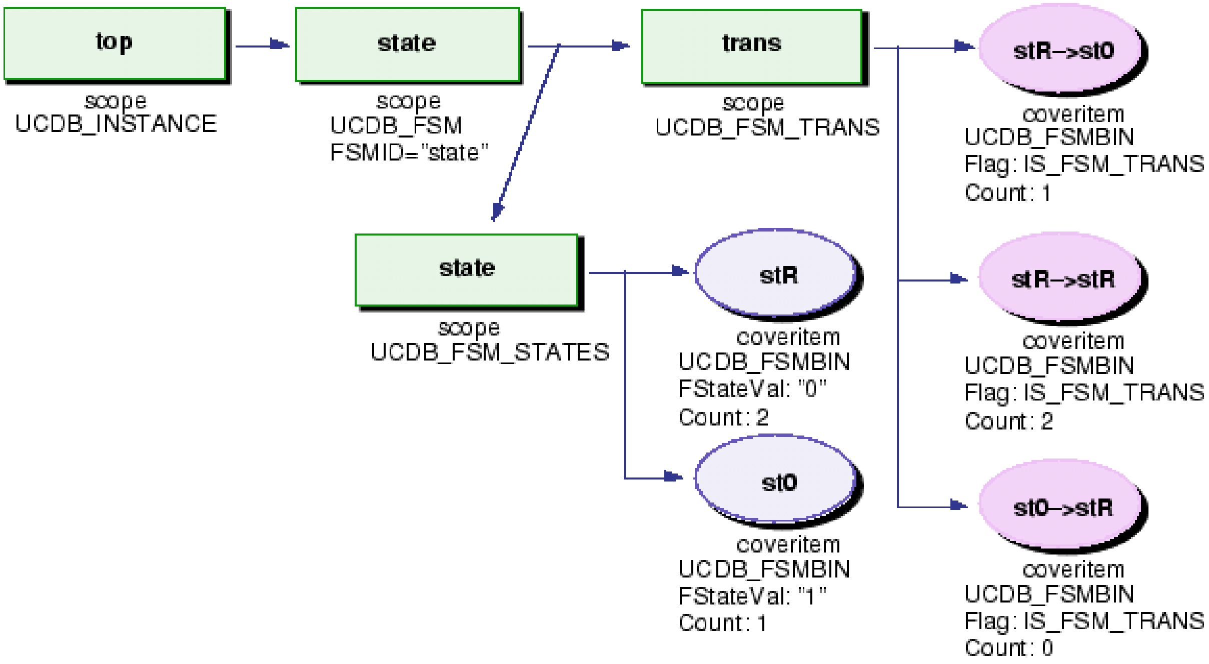
Finite State Machine Coverage

The UCDB represents a Finite State Machine (FSM) as a two-level hierarchy of coverage scopes.

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/fsm/*.

```
module top;
    bit clk = 0;
    bit i = 0;
    bit reset = 1;
    enum { stR, st0 } state;
    always @(posedge clk or posedge reset)
    begin
        if (reset)
            state = stR;
        else
            case(state)
                stR: if (i==0) state = st0;
            endcase
    end
    always #10 clk = ~clk;
    always @(state) $display(state);
    initial begin
        $display(state);
        @(negedge clk);
        @(negedge clk) reset = 0;
        @(negedge clk);
        $stop;
    end
endmodule
```

Figure 1-9. Data Model for a Finite State Machine



The two-level hierarchy is as follows:

- The topmost level is for the state machine itself (whose "FSMID" is identified as an attribute).
- Two child scopes: one for states and one for transitions. These scopes are distinguished by name and scope type. The state machine scope itself (of type UCDB_FSM) is identified by the name of the state variable if possible.

Note

The state machine can take the same name as a toggle coverage scope for exactly the same variable in HDL source.

Refer to "Toggle Coverage" on page 27 for more information.

The state coveritems are named according to the state name. An integer form of the state name is held in the attribute FSTATEVAL and used in the report.

The transition coveritems are named according to the transition. The flag IS_FSM_TRAN is used to distinguish a coveritem of type UCDB_FSMBIN when it is an FSM transition bin.

Toggle Coverage

The UCDB data model for toggle coverage represents, the six basic types of toggle coverage: integer, enum, register, net, extended register, and extended net.

- **Integer toggles —** Covered if the toggle is assigned any value. (VHDL only with Questa) Some unique number of integer values are maintained up to a configurable tool limit.

Note

The Questa simulator breaks Verilog or SystemVerilog integer types into constituent bits such that they become net or register toggles.

- **Enum toggles —** Covered if all the enum values have been assigned.
- **Two-transition register toggles —** Covered if toggled from 0->1 and 1->0.
- **Two-transition net toggles —** Covered if toggled from 0->1 and 1->0. Net (or wire) toggles must be reported without redundancy: in other words, connected nets are reported only once, by the topmost or canonical name. This checking for redundancy is sometimes called “unaliasing” because two connected nets in different levels of hierarchy are really aliases of each other. The topmost net is usually considered to have the canonical name for all connected nets.
- **Six-transition extended register toggles —** (Adds Z transitions.) Covered if it toggles from 1->0 and 0->1 without any Z transitions, otherwise it must show all transitions:

  0->1, 1->0, 0->Z, Z->0, 1->Z, and Z->1.
- **Six-transition extended net toggles —** (Adds Z transitions) Covered with similar rules to register toggles. Unaliasing or elimination of redundancy among connected nets also occurs.

VHDL Integer Toggles

You can find this VHDL example in *<install_dir>/examples/ucdb/userguide/data-models/toggle-int/*.

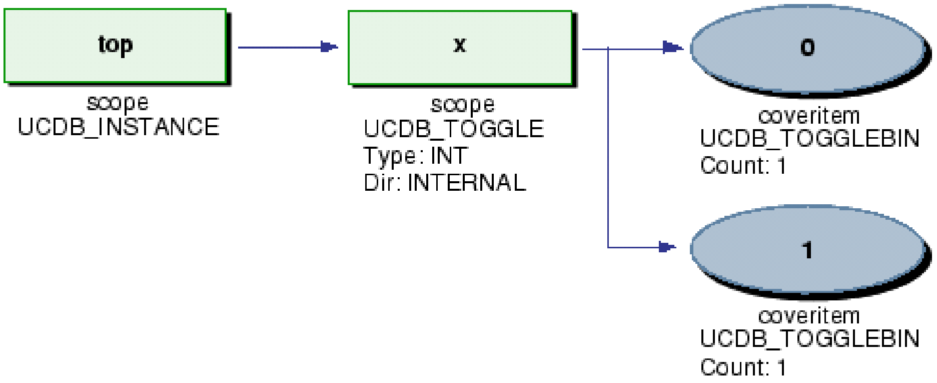
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use std.textio.all;
entity top is
end;
architecture arch of top is
    signal x : integer := 0;
    begin
    branch: process
        variable myoutput : line;
    begin
        wait until x'event;
        write(myoutput,x);
        writeline(output,myoutput);
    end process branch;
    drive: process
    begin
        wait for 10 ns;
        x <= 1;
        wait;
    end process drive;
end architecture;
```

The UCDB_TOGGLE scope is named the same as the variable or signal being covered; if it is also a finite state machine variable, the name appears twice in the database.

The scope has specific information relevant to toggles, two fields of which are visible in Figure 1-10:

- **Type —** These types are the ucdbToggleTypeT enum values that correspond to the six types of toggles.
- **Dir (Direction) —** The ucdbToggleDirT enum values: INTERNAL, IN, OUT, and INOUT. These are used by the report software to restrict the subset of toggles being reported upon.
- **Canonical Name —** The canonical name of the toggle node if it is a wire and is not the topmost node. This is not shown in the example because it is NULL.

Figure 1-10. Data Model for a VHDL Integer Toggle



The integer toggle has bins for both of the values it assumes: “0” and “1”. The bins are named according to the integer value of the signal.

Note

If there were no data changes (no events) on the integer signal, there would be no bins for the toggle scope in the UCDB. However, because the default integer value is counted as a bin value, it is not possible to have only one bin for the integer toggle; it has at least the default value plus some set of other values to which it was assigned (up to a configurable tool limit with the ToggleMaxIntValues variable in the *modelsim.ini* file.)

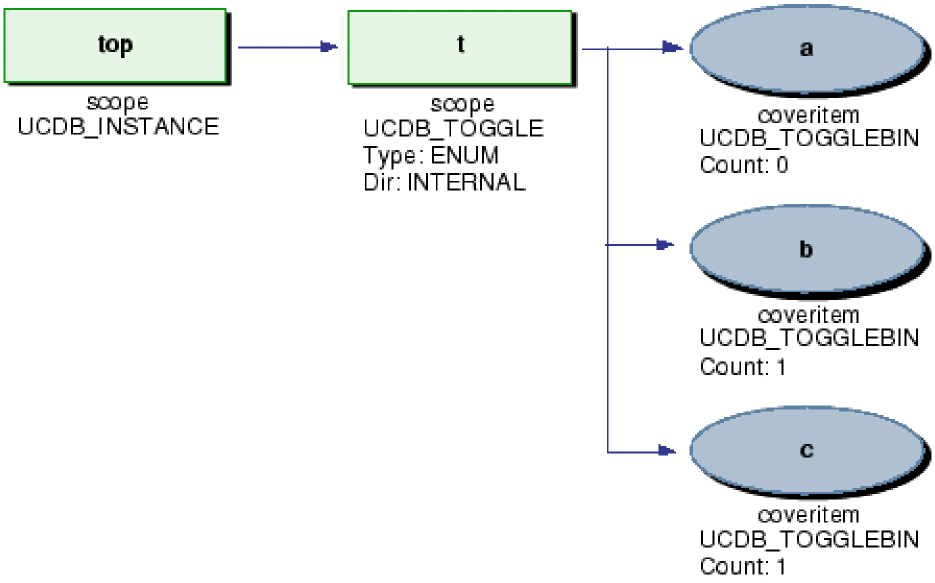
Enum Toggles

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/toggle-enum/*.

```
module top;
    enum { a, b, c } t = a;
    initial begin
        #1; t = c;
        #1; t = b;
    end
endmodule
```

This example is similar to the one showing VHDL Integer Toggles, except with the toggle type equal to ENUM. The coveritems are named according to enum values. In this case, the default value is explicitly not covered, to distinguish between an explicit and implicit assignment to that value. In this particular simulation, “b” and “c” are covered while “a” is not, so the toggle “/top/t” itself is uncovered.

Figure 1-11. Data Model for an Enum Toggle



Extended Register Toggle

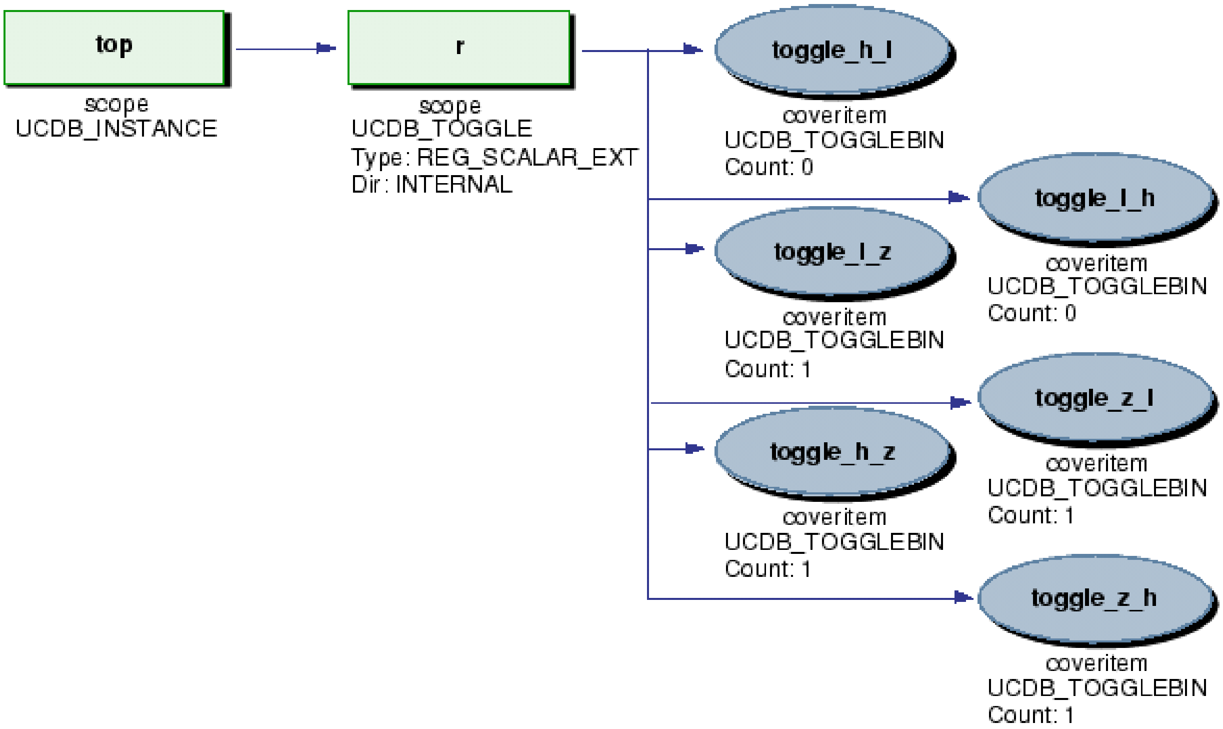
The following example shows an extended (6-transition) toggle for a register only. You can also use extended toggle coverage for nets; the only difference is the toggle type.

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/toggle-reg-ext/*.

```
module top;
    logic r = 1'bx;
    initial begin
        #1; r = 1'b0;
        #1; r = 1'b1;
        #1; r = 1'bz;
        #1; r = 1'b0;
        #1; r = 1'b1;
        #1; r = 1'b0;
    end
endmodule
```

The type of the toggle scope shows that this is a register extended toggle. The six bins are named according to the possible transitions among 0, 1, and z. If this were a 2-transition toggle, it would be covered based on “toggle_h_l” and “toggle_l_h” bins. However, because it has some z transitions with non-zero count, it would have to have all bins with non-zero count in order for the “/top/r” register toggle to be covered.

Figure 1-12. Data Model for an Extended Register Toggle



Net Toggle with Connected Net

This example requires turning off the optimizer in Questa to allow the “bottom” module to survive elaboration.

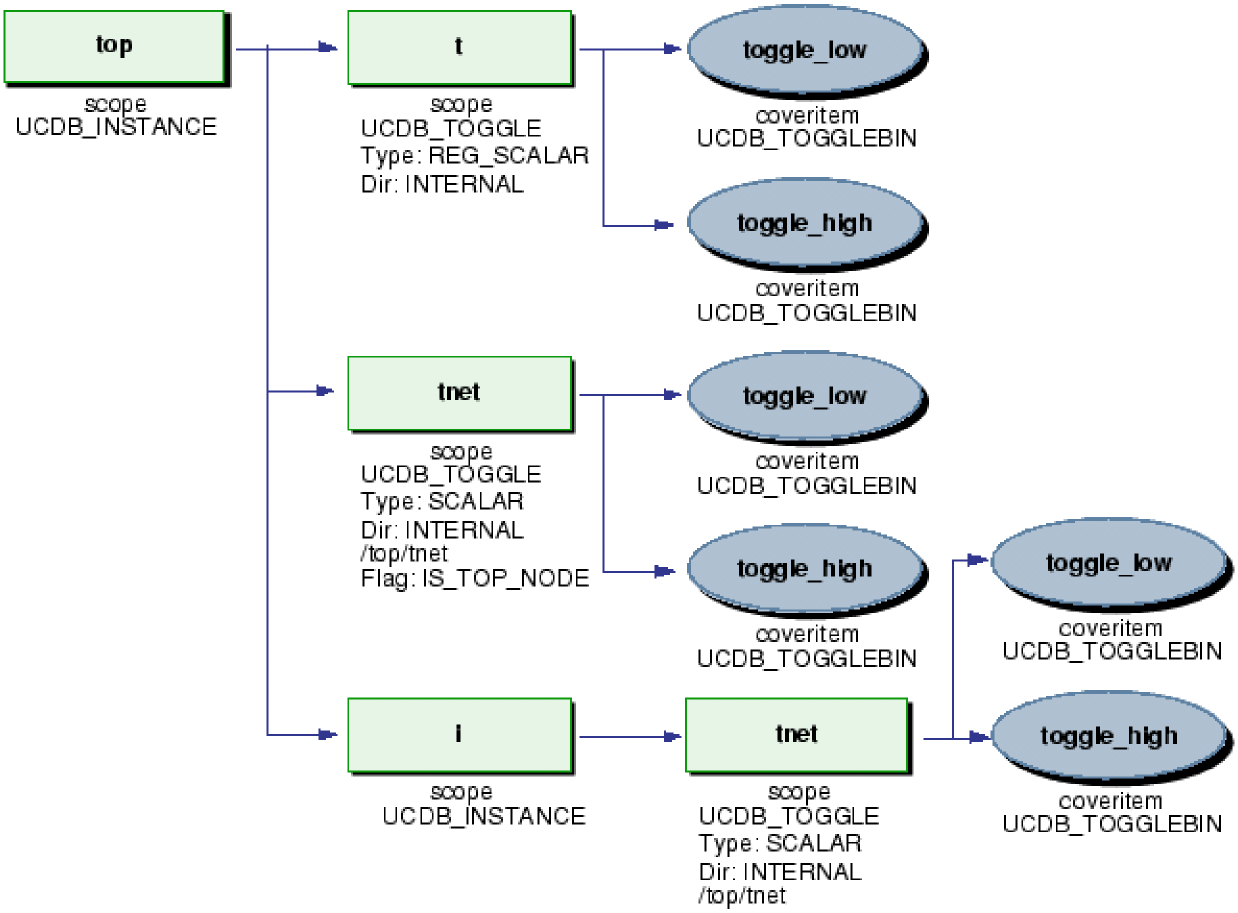
You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/toggle-net/*.

```
module top;
    bit t = 0;
    wire tnet;
    assign tnet = t;
    initial begin
        #1; t = 1;
        #1; t = 0;
    end
    bottom i(tnet);
endmodule
module bottom(input wire tnet);
    always @(tnet)
        $display(tnet);
endmodule
```

Because the UCDB does not represent connectivity, it must indicate the connectedness of two nets (“tnet” in this example) in a different way. Use the following two data attributes:

- The top node has a flag – UCDB_IS_TOP_NODE – set for the toggle scope. This is useful when traversing the entire database to restrict the report or other analysis to top-level (canonical) nodes only. However, it does not suffice for analyzing a subset of the database. Hierarchical references as well as port connections can create connected nets.
- The canonical name is stored for all net toggles. This name is accessed with the ucdb_GetToggleInfo() function, which also returns toggle type and toggle direction. In this example, /top/tnet and /top/i/tnet both have the same canonical name: “/top/tnet”.

Figure 1-13. Data Model for a Connected Net Toggle



Group Data Model

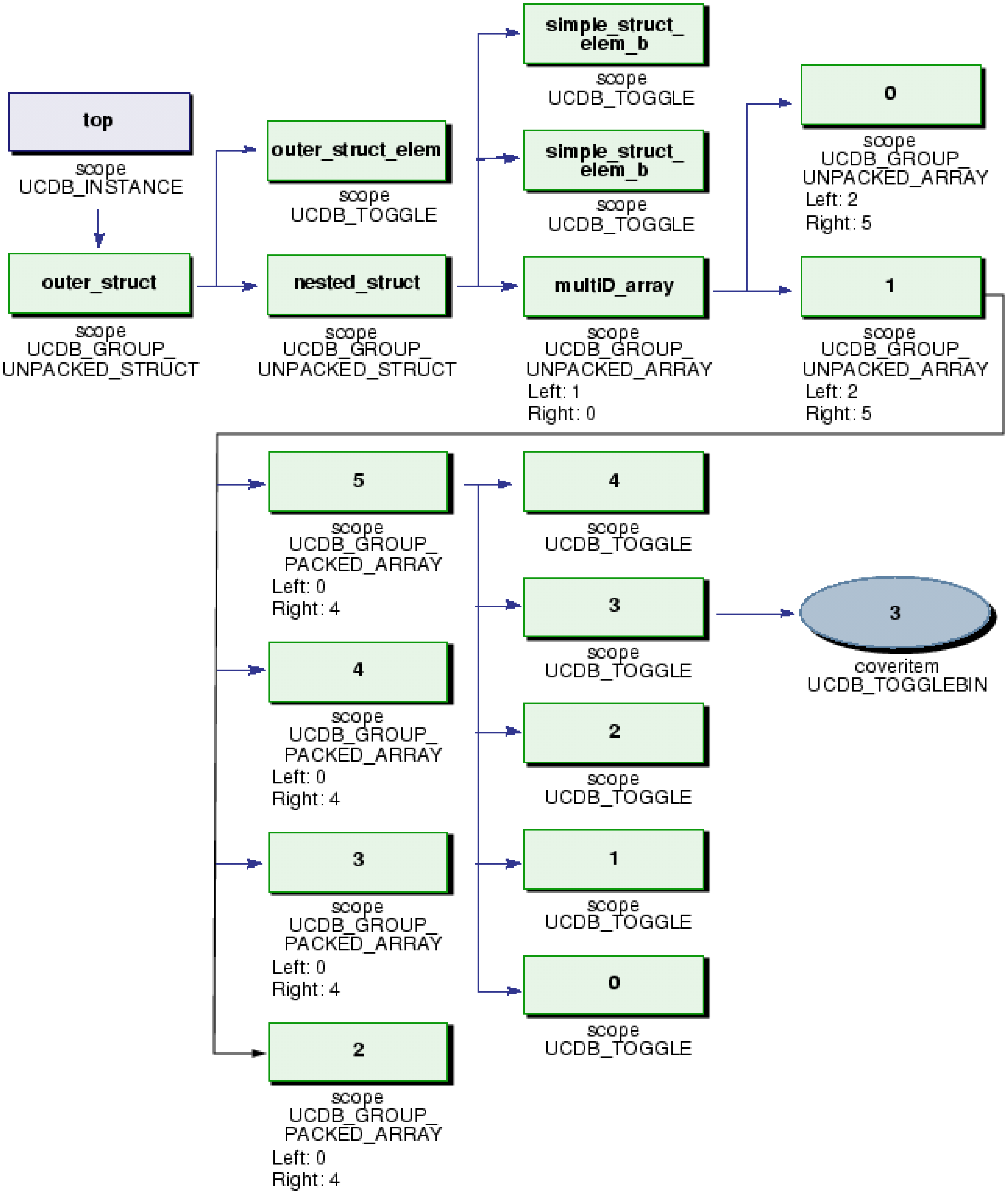
The UCDB data model provides a definition of group information.

The following is an example of the group data model.

SystemVerilog Example (“top/outer_struct.nested_struct.multiD_array[1][5][3]”):

```
module top;

   typedef struct {
      reg[0:4] multiD_array [1:0] [2:5];
      bit simple_struct_elem_b;
      bit simple_struct_elem_c;
   } ST1;

   typedef struct {
      bit outer_struct_elem;
      ST1 nested_struct;
   } ST2;

   ST2 outer_struct;

   initial begin
       outer_struct.nested_struct.multiD_array[1][5][3]= 1’b0;
   end

endmodule
```

Figure 1-14. Data Model for a Group



SVA and PSL Covers

Cover directives in PSL or cover statements in SystemVerilog Assertions language are exactly the same in Questa. (Both are referred to as “cover directives” in Questa.)

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/cover/*.

```
module top;
    bit a = 0, b = 0, clk = 0;
    always #10 clk = ~clk;
    initial begin
        @(negedge clk);        b = 1;
        @(negedge clk); a = 1; b = 0;
        @(negedge clk); a = 0;
        @(negedge clk); $stop;
    end
    // psl default clock = rose(clk);
    // psl pslcover: cover {b;a};            // line 21
    sequence a_after_b;
        @(posedge clk) b ##1 a;
    endsequence
    svacover: cover property(a_after_b);     // line 25
endmodule
```

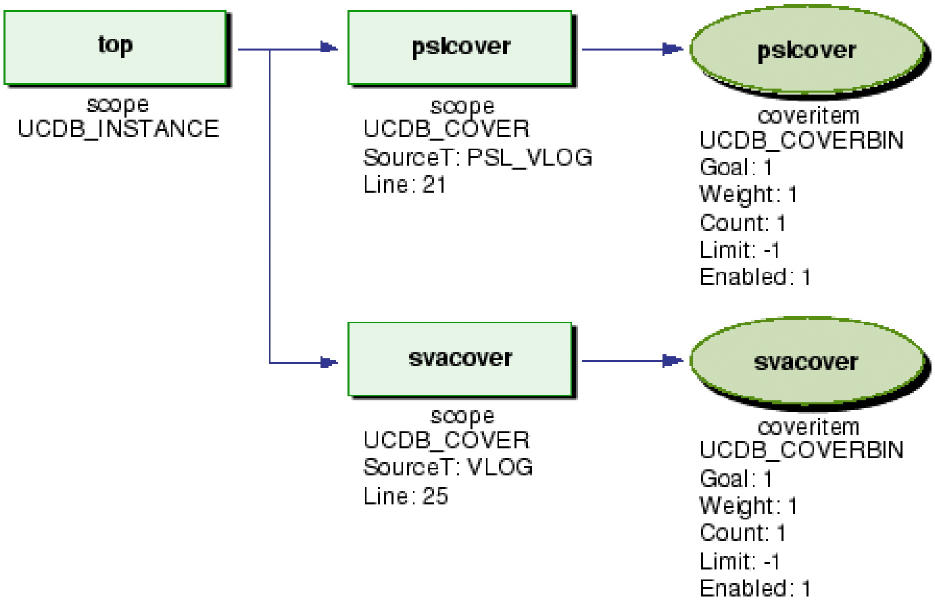
The two cover directives, lines 21 and 25, are identical except for the following two differences:

- **Line number —** Accessed with source information
- **Scope source type —** Accessed with ucdb_GetScopeSourceType()). Specifically, PSL_VLOG for the Verilog PSL, VLOG for the native SVA cover. The value PSL_VHDL is used for VHDL PSL.

There are additional data, accessed with ucdb_GetCoverData(), available for cover directives:

- **Goal —** A tool feature in Questa, the “at_least” value for a cover directive, set with the fcover configure command.
- **Weight —** An individual weight for this cover directive, another tool feature. The weight is set at the coveritem level as well as the UCDB_COVER scope level.
- **Limit —** Questa has a tool feature for disabling a cover after reaching a certain count. When you set the value to -1, there is no limit.
- **Enabled —** Questa has a tool feature for disabling a cover directive. This feature is disabled when the enabled bit is set to FALSE.
- **Count —** The pass count for the cover directive. Failure counts are implied in the SystemVerilog LRM for sequences; vacuous passes and attempts for properties. Multiple counts are not maintained in Questa.

Figure 1-15. Data Model for SVA and PSL Cover Directives



Assertion Data

The UCDB data model for assertions maintains different counts in different circumstances.

- The immediate or concurrent assertion with a fail count only
- The concurrent assertion with a full complement of seven counts (assert debug mode)
- The immediate assertion with both fail count and pass count (assert debug mode)

Assertions with Fail Counts Only

You can find this Verilog example in *<install_dir>/examples/ucdb/userguide/data-models/assert/*.

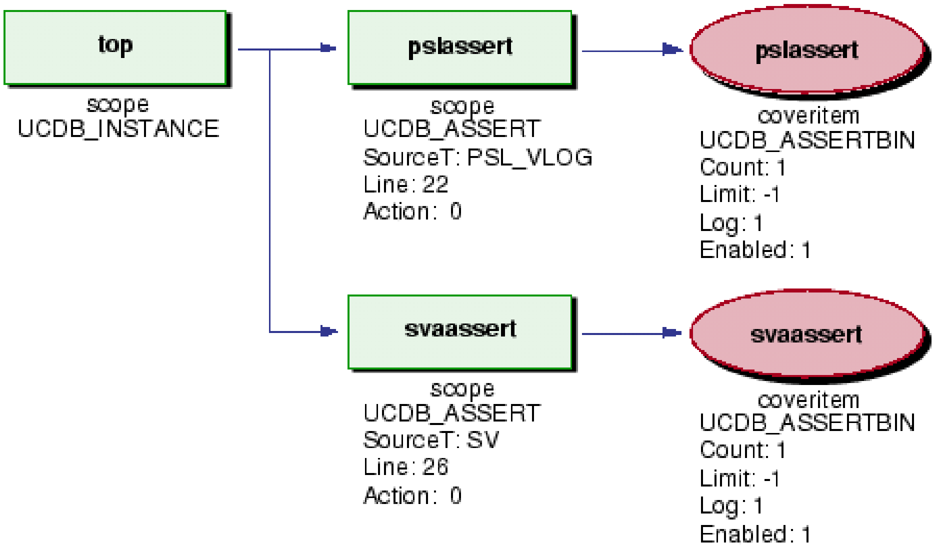
```
module top;
    bit a = 0, b = 0, clk = 0;
    always #10 clk = ~clk;
    initial begin
        @(negedge clk);        b = 1;
        @(negedge clk); a = 1; b = 0;
        @(negedge clk); a = 0; b = 1;
        @(negedge clk);        b = 0;
        @(negedge clk); $stop;
    end
    // psl default clock = rose(clk);
    // psl pslassert: assert always {b} |=> {a};
    property a_after_b;
        @(posedge clk) b |=> a;
    endproperty
    svaassert: assert property(a_after_b);
endmodule
```

The UCDB_ASSERTBIN is the fail count for the assertion. Other aspects of the data model include the following:

- The “ACTION” attribute on the UCDB_ASSERT scope. This is an integer attribute whose values indicate how the simulator should react to an assertion failure:
  - 0 — Continue after failure.
  - 1 — Break after failure.
  - 2 — Exit after failure.
- Log (the flag UCDB_LOG_ON) — this is a bit to indicate that the assertion failure messages appear in the simulator transcript.

Other aspects of the data model are similar to the cover directives.

Figure 1-16. Data Model for Assertions (With Fail Count Only)

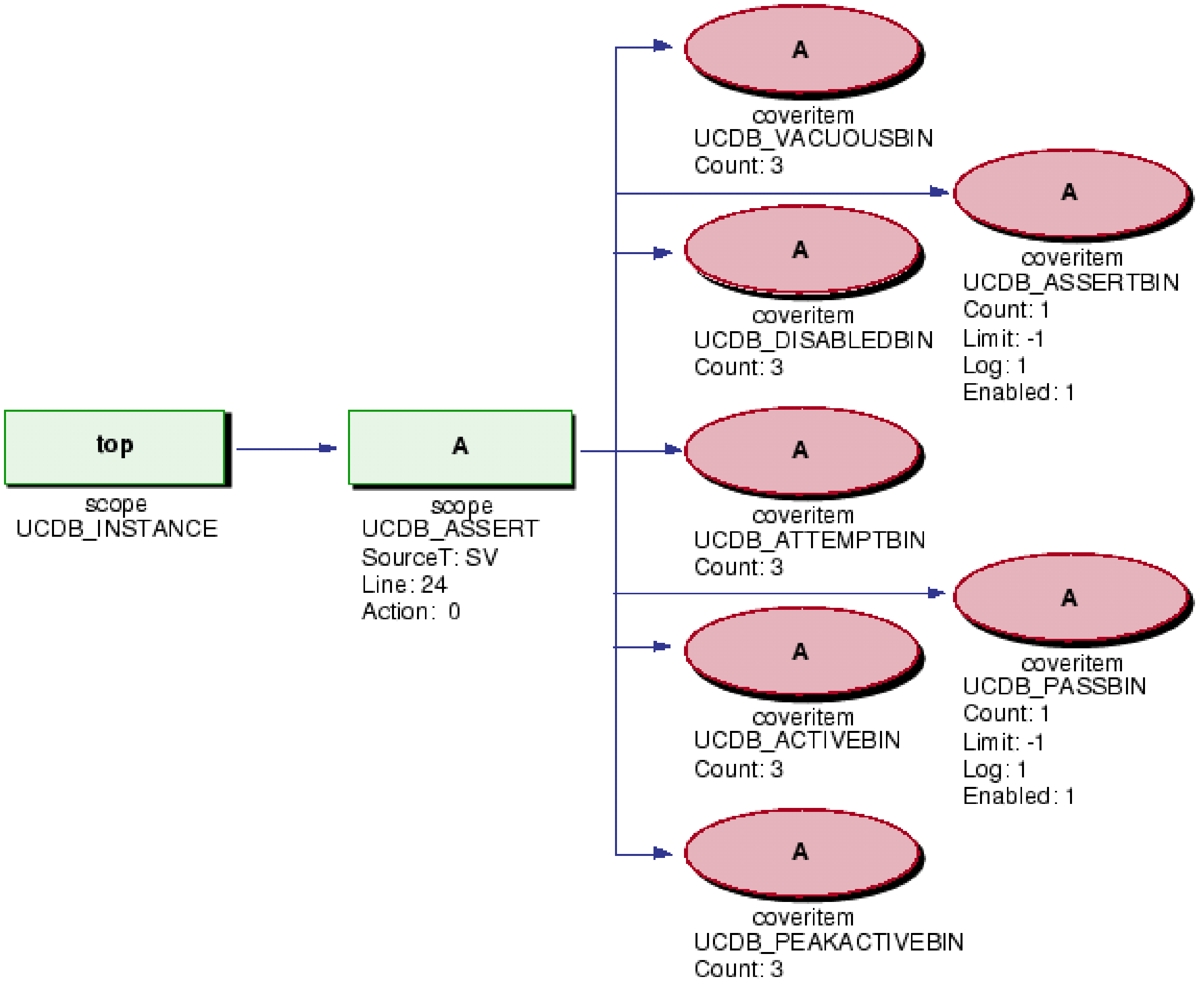


Assertion with All Counts Using -assertdebug

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/assert-debug/*.

```
module top;
    bit a = 0, b = 0, clk = 0;
    always #10 clk = ~clk;
    initial begin
        @(negedge clk);        b = 1;
        @(negedge clk); a = 1; b = 0;
        @(negedge clk); a = 0; b = 1;
        @(negedge clk);        b = 0;
        @(negedge clk); $stop;
    end
    property a_after_b;
        @(posedge clk) b |=> a;
    endproperty
    A: assert property(a_after_b);        // line 24
endmodule
```

Figure 1-17. Data Model for an Assertion (With All Counts)



This data model currently represents seven bins with the following meanings:

- **ASSERTBIN —** The assertion failure count. Has data values for limit, log, and so on.
- **PASSBIN —** The assertion non-vacuous pass (success) count. Similar to the ASSERTBIN in which flags and data fields it offers. PASSBIN is useful to determine if an assertion has been fully exercised during simulation. Coverage metrics derived from an assertion use this metric if available.
- **VACUOUSBIN —** The vacuous pass (success) count. This is for implications whose left-hand-side is false.
- **DISABLEDBIN —** Counts the number of cycles for which the assertion was explicitly disabled through the SystemVerilog “disable iff” construct. This is essentially the number of attempts missed because the assertion was disabled.
- **ATTEMPTBIN —** The number of times the assertion was attempted: the number of times its clocking expression triggered.
- **ACTIVEBIN —** The number of threads left active (in progress) at the end of simulation for this assertion.
- **PEAKACTIVEBIN —** The maximum number of threads ever created for this assertion at any given point in time.

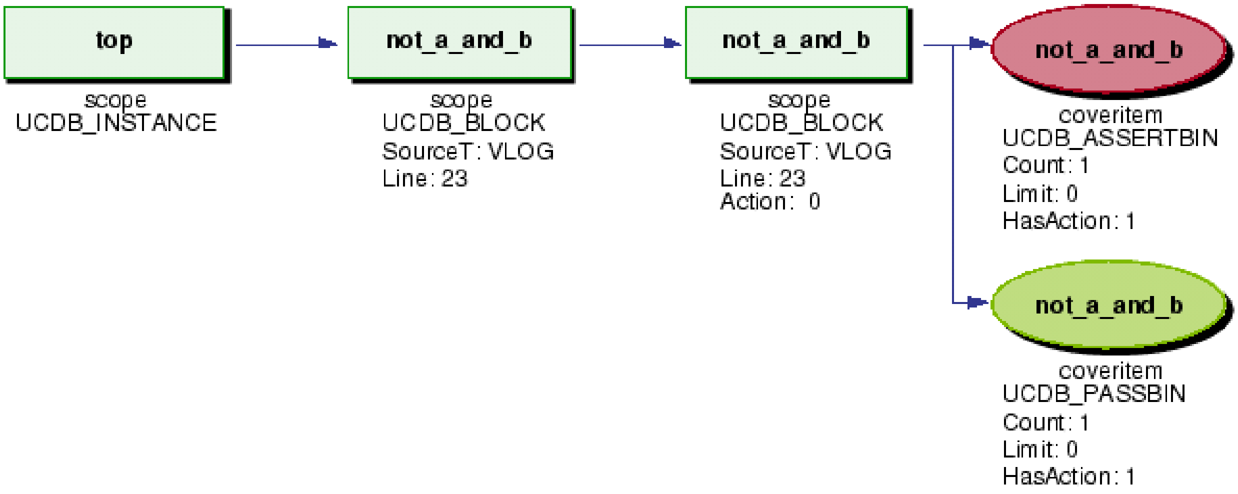
Immediate Assert with Pass/Fail

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/immed-assert/*.

Example, compiled with -assertdebug and without the optimizer (“immed-assert”):

```
module top;
    bit a = 0, b = 0, clk = 0;
    always #10 clk = ~clk;
    initial begin
        @(negedge clk);        b = 1;
        @(negedge clk); a = 1; b = 0;
        @(negedge clk); a = 1; b = 1;
        @(negedge clk);        b = 0;
        @(negedge clk); $stop;
    end
    always @(posedge clk)                    // line 22
       not_a_and_b: assert (!(a && b)) else $error("a and b both true!");
endmodule
```

Figure 1-18. Data Model for an Immediate Assertion With Pass/Fail Counts



SystemVerilog Covergroup Coverage

This section describes types of covergroup coverage, including covergroup with a cross, sparse cross bin representation, CROSSSBINIDX and CROSSUBINIDX, covergroup with per-instance coverage, and embedded covergroups.

SystemVerilog covergroups have both a type and instances of that type. The type is associated with the declaration in a particular scope; instances are created when covergroups are instantiated (or constructed) with the new keyword.

The following are changes to the SystemVerilog standard that affect the covergroup data model:

- There are two covergroup type aggregation algorithms:
    - Weighted average of instances is the default.
    - Merge of instances. Set with type_option.merge_instances. In this case, instance-specific coverage is merged into the cumulative (or type) coverage according to bin name for each coverpoint and cross.
- Setting option.per_instance specifies whether a covergroup instance is to be saved into the coverage database; when false, implementations are not required to be saved. In the case of the Questa simulator, to avoid throwing away coverage by default, option.per_instance is effectively ignored.
- Setting option.get_inst_coverage specifies whether the get_inst_coverage() method is enabled; if not, it returns the same as get_coverage(). This option applies only when option.merge_instances is 1, which enables an optimization that is not yet enabled in Questa.

Covergroup With a Cross 42
Sparse Cross Bin Representation 45
CROSSSBINIDX and CROSSUBINIDX 46
Covergroup in Package With Multiple Instances 48
Covergroup in a Class (Embedded Covergroup) 51

Covergroup With a Cross

The UCDB provides a data model specific to covergroups containing a cross.

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/covergroup/*.

```
module top;
    int a = 0, b = 0;
    covergroup cg;
        type_option.comment = "Example";
        option.at_least = 2;
        cvpa: coverpoint a { bins a = { 0 }; }
        cvpb: coverpoint b { bins b = { 1 }; }
        axb: cross cvpa, cvpb { type_option.weight = 2; }
    endgroup
    cg cv = new;
    initial begin
        #1; a = 0; b = 1; cv.sample();
        #1; a = 1; b = 1; cv.sample();
        #1; $display($get_coverage());
    end
endmodule
```

The covergroup type rollup is part of the subtree rooted at the “cg” (UCDB_COVERGROUP) node, specifically, the subtree containing the UCDB_COVERPOINT and UCDB_CROSS children. The covergroup instance is the subtree rooted at the UCDB_COVERINSTANCE node. It is a mirror of the type subtree.

Note

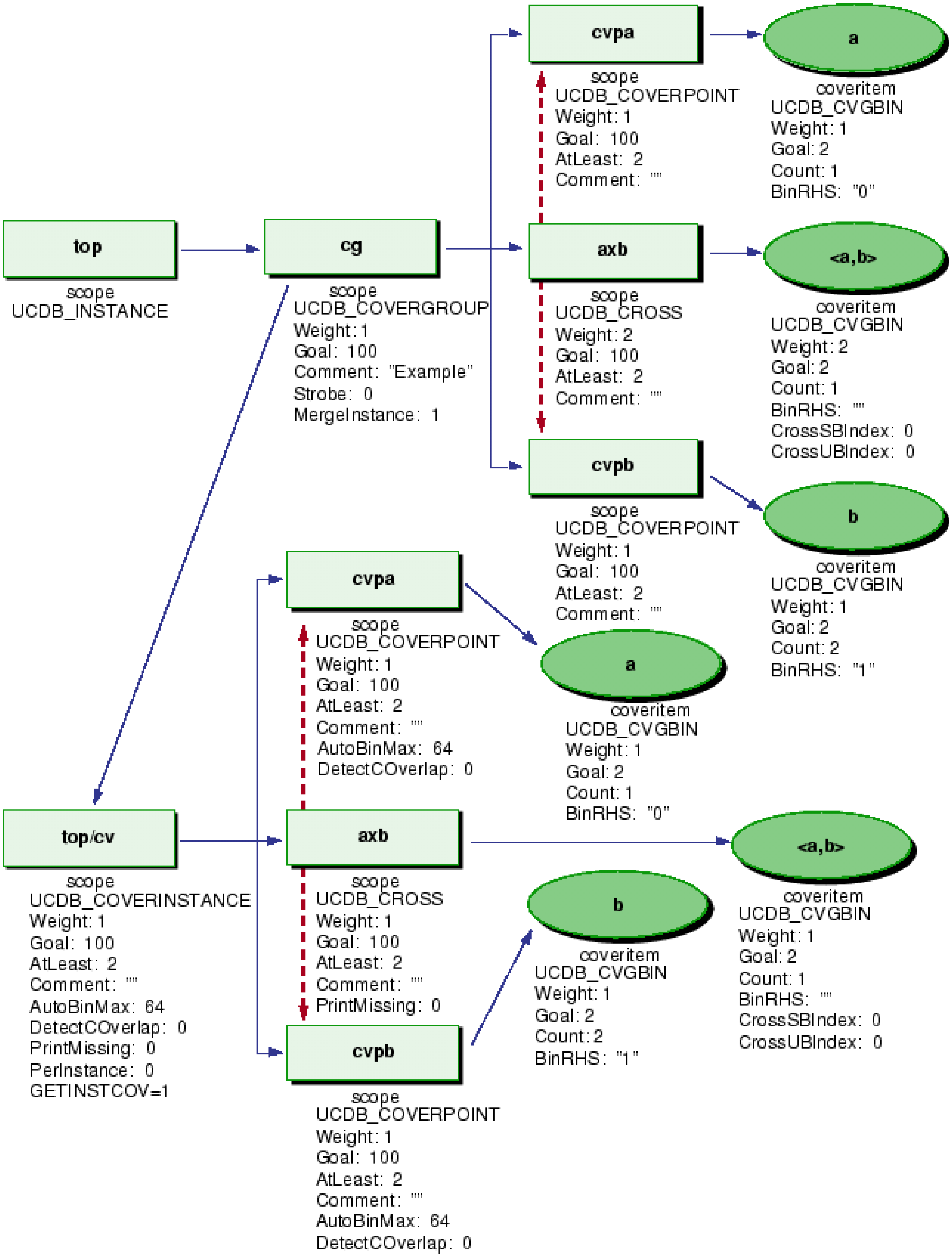
When there are multiple instances, the number of coverpoint and cross children must be the same among all instances, but the numbers of bins can be different.

In this case, it is true, the instance data is largely redundant, but because option.per_instance is effectively ignored by Questa, the instance data serves the purpose of storing instance-specific options and is also used when the data is reloaded with $load_coverage_db().

The following are a couple of important features of the instance data:

- Weight is a primary data component of a UCDB scope – This weight is accessed with ucdb_GetScopeWeight(). The cross weight is reflected in the weight for the axb cross scope. It is also reflected in the weight associated with the coveritem itself, but this meausurement is less useful.
- The goal for the scope – ucdb_GetScopeGoal() – This is the goal established by the covergroup type_option.goal or option.goal. In this example, all scopes including covergroup, coverinstance, coverpoint, and cross – have a default of 100, reported as a percentage. The attribute name is “#GOAL#” to adhere to a convention whereby attributes with “#” in the name do not appear in the command-line and graphical user interface.

Figure 1-19. Data Model for a Cross



- The goal associated with the coveritem is really the “at_least” value for the covergroup. This allows a simple algorithm for determining if a coveritem is covered: if its count is greater than or equal to its goal.
- Other attributes reflect the type_option or option values associated with the covergroup, coverpoint, or cross:
    - COMMENT — For the type_option.comment in all scopes in the type subtree, or the option.comment in all scopes in the instance subtree.
    - STROBE — For the type_option.strobe for the covergroup scope.
    - AUTOBINMAX — For the option.auto_bin_max in covergroup and coverpoint scopes.
    - DETECTOVERLAP — For the option.detect_overlap in covergroup and coverpoint scopes.
    - PRINTMISSING — For the option.cross_num_print_missing in covergroup and cross scopes.
    - GETINSTCOV — For the option.get_inst_cov.
- There are some additional defined attributes:
    - BINRHS — the set of sampled values that could potentially cause the bin to increment. These are referred to as the “bin right-hand side values” because they are derived from the right-hand-side of the “=” declaration for the bin. BINRHS is not set for the cross bin because the bin depends only on the coverpoint bins, which are referenced as part of the bin name (“<a,b>”) in this case. If the cross bin were explicitly declared (with the cross select expression syntax), then there would be a meaningful BINRHS attribute for the cross bin.
    - CROSSSBINIDX and CROSSUBINIDX — these are used to implement the SystemVerilog call $load_coverage_db().
- There is an association between the cross and its component coverpoints, indicated by the red dashed lines in Figure 1-12 on page 31. These associations are accessed with the following functions:
    - ucdb_GetNumCrossedCvps()
    - ucdb_GetIthCrossedCvp()
    - ucdb_GetIthCrossedCvpName()

Sparse Cross Bin Representation

The UCDB only counts cross bins with non-zero coverage, resulting in a more efficient database.

Unfortunately, much of this infrastructure relies on a private API. (If you are a customer and you would like to use it, please request it.) The software will still accept UCDBs with fully enumerated crosses, that is, all cross bins stored explicitly in the database, and there is a trick for allowing the API to traverse all bins whether they are stored or not. So the sparse implementation is optional.

To enable the API to traverse all bins, whether stored or not, use this function:

```
void
ucdb_SetIterateAllCrossAutoBins(
    ucdbT db,
    int yesno);
```

If this is called as ucdb_SetIterateAllCrossAutoBins(db,1), then the API will create a bin object during traversal whether it was really stored or not.

Other relevant bits of information for sparse crosses:

- The 0x10000000 bit is set in the cross scope flags value if it is sparsely implemented.
- The attribute “#CROSSNUMBINS#” shows the total number of coverage bins in the cross, useful for computing total coverage.

The mechanism for storing crosses sparsely closely follows the cross select expression syntax and semantic in SystemVerilog. There is an expression API that can be used with the cross, essentially to store and retrieve the cross select expression with which the cross was specified in the SystemVerilog source.

This approach is a mechanism whereby the dimensions of the covered cross space are specified with tuples of ranges (a pair of low-high values to indicate a space in two dimensions, for example.) Both mechanisms share the technique of storing full details only for cross bins with non-zero coverage counts.

CROSSSBINIDX and CROSSUBINIDX

The UCDB data model for cross bins relies on user-defined attributes and only apply when using the $load_coverage_db() system task.

These user-defined attributes are associated with cross bins to implement the SystemVerilog predefined system task $load_coverage_db(). For Questa, the $load_coverage_db() predefined system task cannot work unless these attributes are correctly set.

Note

If you do not care about using $load_coverage_db(), then these user-defined attribute values could be ignored. They are created automatically by Questa but might take some work to reproduce independently. $load_coverage_db() requires a corresponding SystemVerilog covergroup to have been created in simulation, otherwise the load will fail anyway. If you are porting the covergroup from third-party data, you are required to create a corresponding SystemVerilog covergroup into which the data could be reloaded in simulation with $load_coverage_db().

Consider this more complex covergroup with a cross.

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/covergroup-cross3x3/*.

```
covergroup cg;
   cvpa: coverpoint a { bins azero = { 0 }; bins anonzero[] = { [1:2] }; }
   cvpb: coverpoint b { bins bzero = { 0 }; bins bnonzero[] = { [1:2] }; }
   axb: cross cvpa, cvpb;
endgroup
```

Table 1-1. Cross Bin Index Attributes

Bin Name	CROSSUBINIDX	CROSSSBINIDX
<azero,bzero>	0	0
<anonzero[1],bzero>	1	0
<anonzero[2],bzero>	1	1
<azero,bnonzero[1]>	2	0
<azero,bnonzero[2]>	2	1
<anonzero[1],bnonzero[1]>	3	0
<anonzero[2],bnonzero[1]>	3	1
<anonzero[1],bnonzero[2]>	3	2
<anonzero[2],bnonzero[2]>	3	3

- CROSSUBINIDX — mnemonically, cross user bin index. Using an internal terminology by which a “bin declaration” is a “user bin”. This is the syntactic bin declaration with a bin name and it is terminated by a semicolon.
- CROSSSBINIDX — mnemonically, cross sub-bin index. Using an internal terminology by which a bin is a sub-bin. This is the actual bin or coveritem object with an individual count, which may map 1-to-1 with the declaration or many-to-1.

It is important to view the coverpoint bin declarations in isolation. Although there are three bins in each coverpoint and thus nine in the cross, there are two bin declarations in each coverpoint. The cross is organized into four groups of crosses of bin declarations:

- <azero,bzero>
- <anonzero[*],bzero>
- <azero,bnonzero[*]>
- <anonzero[*],bnonzero[*]>

The CROSSUBINIDX is an index value corresponding to these groups. The bin declarations in the leftmost crossed coverpoint (“a” in this case) are less significant because they change more rapidly as the cross bins are enumerated. This is implementation specific and is reflected in the order of bins in the report.

The CROSSSBINIDX is a given bin's index within one of these groups.

Covergroup in Package With Multiple Instances

The UCDB data model supports the scenario of a covergroup with per-instance coverage along with a covergroup within a package.

You can find this Verilog example in *<install_dir>/examples/ucdb/userguide/data-models/covergroup-perinstance/*.

```
package p;
    covergroup cg (ref int v);
        option.per_instance = 1;
        coverpoint v { bins val[] = { [0:1] }; }
    endgroup
endpackage
module top;
    int a, b;
    p::cg cva = new(a);
    p::cg cvb = new(b);
    initial begin
        #1; a = 0; cva.sample();
        #1; b = 1; cvb.sample();
        #1; $display("cva=%.2f cvb=%.2f cva+cvb=%.2f",
                     cva.get_inst_coverage(),
                     cvb.get_inst_coverage(),
                     p::cg::get_coverage());
    end
endmodule
```

This Verilog example illustrates two interesting cases together: the covergroup with per-instance coverage (option.per_instance assigned to 1), and the covergroup in a package. Figure 1-20 shows the different scope types for a package. The package has an instance type UCDB_PACKAGE, and a design unit type UCDB_DU_PACKAGE.

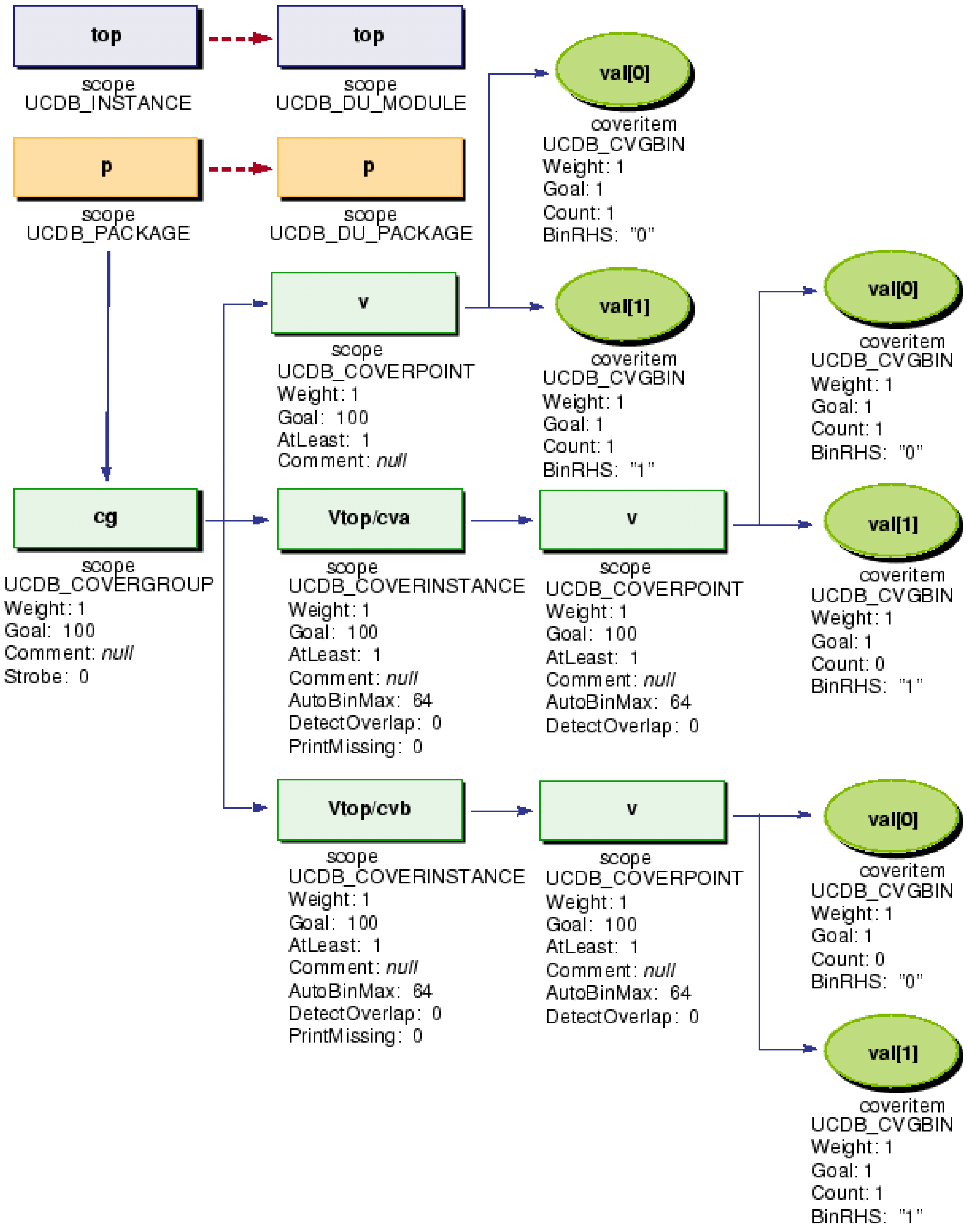
The module instance "top" has nothing in it. The covergroup variables are in the module top, but covergroup variables are nothing more than references to a previously created covergroup object. The object might exist with no reference (because coverage must persist, covergroup objects are not garbage collected); there could be more than one reference to a given object; or the same reference might refer to different objects at different points in time. So the covergroup variable is not very relevant to the covergroup objects themselves. Consequently, the covergroup instances in the UCDB are stored underneath the covergroup type (UCDB_COVERGROUP scope) as a different UCDB scope type: UCDB_COVERINSTANCE.

Covergroup instances are identified by name. You can assign the name explicitly by assigning option.name or using the set_inst_name() built-in method. If not assigned explicitly, Questa automatically assigns the covergroup instance name using the path to the variable used to construct the covergroup object. This path is quoted as an extended identifier so that references to paths within the UCDB work easily. The middle coverpoint scope in Figure 1-20 would be referenced as "/p/cg/\/top/cva /v".

Note

The space after cva terminates the extended identifier.

Figure 1-20. Data Model for a Covergroup (With Per-Instance Coverage)



The UCDB_COVERINSTANCE scopes and their child scopes have attributes that convey the option values for those scopes:

- **ATLEAST —** The option.at_least value.
- **COMMENT —** The option.comment for the corresponding scopes.
- **AUTOBINMAX —** The option.auto_bin_max setting.
- **DETECTOVERLAP —** This is option.detect_overlap.
- **PRINTMISSING —** The option.cross_num_print_missing.

Note

The option.per_instance itself is implied by the presence of the UCDB_COVERINSTANCE in the data model.

The covergroup with perinstance coverage example shows that you can calculate the get_inst_coverage() for /top/cva or /top/cvb from the UCDB_COVERINSTANCE scopes. You can also calculate the get_coverage() for covergroup cg from the UCDB_COVERPOINT scope "/p/cg/v", that is, the coverpoint that is an immediate child of the UCDB_COVERGROUP. The coverpoint (Question for the writer: is this the correct term?) represents the type coverage for the covergroup.

The covergroup coverage is the merge of the coverage from the two instances. IEEE Std 1800-2005 states "It is important to understand the cumulative....of all instances." In other words, the /top/cva instance covers bin val[0], while the /top/cvb instance covers bin val[1]. Therefore, each instance has 50% coverage, but the type is covered 100% because each bin is covered in the union contributed from all instances. This coverage is reflected in the simulation output of the $display in the example:

```
# cva=50.00 cvb=50.00 cva+cvb=100.00

# cva=50.00 cvb=50.00 cva+cvb=100.00
```

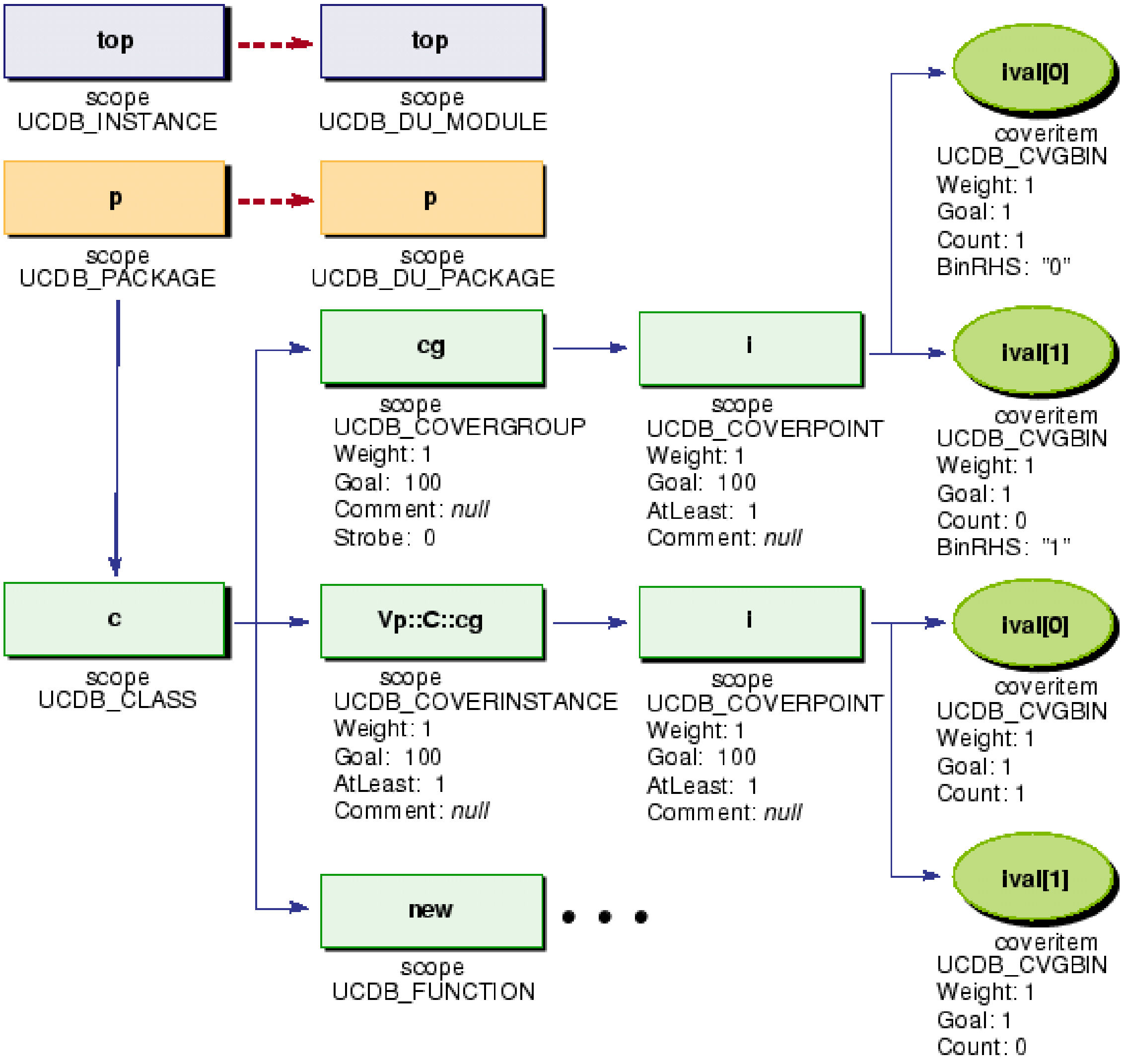
Covergroup in a Class (Embedded Covergroup)

The UCDB data model provides information for when you embed covergroups within a class.

You can find this SystemVerilog example in <*install_dir*>*/examples/ucdb/userguide/data-models/covergroup-embedded/*.

```
package p;
    class c;
        int i;
        covergroup cg;
            coverpoint i { bins ival[] = { [0:1] }; }
        endgroup
        function new();
            cg = new;
        endfunction
        function void sample(int val);
            i = val;
            cg.sample();
        endfunction
    endclass
endpackage
module top;
    p::c cv = new;
    initial begin
        cv.sample(0);
        $display($get_coverage());
    end
endmodule
```

Figure 1-21. Data Model for an Embedded Covergroup



The covergroup type name is stored as the declaration name “cg”. Technically, this is incorrect: IEEE Std 1800-2008 specifies that the embedded covergroup declaration creates a covergroup of anonymous type. In Questa this is really “#cg#” and is invisible to the user. However, because UCDB scope names must be visible during coverage analysis, Questa transforms the anonymous name to the visible covergroup variable name. This is allowable because the embedded covergroup has other restrictions that result in a 1-to-1 mapping between the covergroup type and the covergroup variable.

The data model shows how other unexpected scopes, such as, “new” and “post_randomize”, are created in the UCDB. This is because scopes are saved in the UCDB prior to determining whether or not they contain coverage.

The presence of these scopes does illustrate how the UCDB captures the complete context tree of the elaborated design.

Design Units

The output of ucdbdump shows some of the data associated with a design unit, with which you can analyze coverage information about your design units.

You can find this SystemVerilog example in *<install_dir>/examples/ucdb/userguide/data-models/fsm/*.

```
------------- DESIGN UNIT ------------------
Name            : work.top
Type            : UCDB_DU_MODULE
Source type     : VERILOG
File info       : name = test.sv line = 0
Flags           : 0x00000121
Attribute: name = DUSIGNATURE string = ogR[Jb^m9kQbO9nX]eoj;1
```

- Name — The name is composed as library.name for Verilog and library.entity(architecture) for VHDL. In Verilog, the architecture notation may be used for variants created by parameterization or optimization; however, these are merged together to create a single design unit.

  The reason is that these variants may be created arbitrarily by the optimizer is that they could be artifacts not intended by design author. This does have the consequence that the context tree in the UCDB will differ from the context tree visible in Questa in simulation. The same is true for VHDL design units which are sometimes denoted library.entity(architecture)#index. The different index versions are merged together to reflect the canonical design unit.

- Flags

  - UCDB_ENABLED_STMT (0x00000002) through UCDB_ENABLED_TOGGLEEXT (0x00000080) — A flag equired for code coverage to appear in the Questa reports. These flags are created correctly by Questa itself.

  - UCDB_SCOPE_UNDER_DU — An internal flag to mark the scopes under the design unit, if any, as well as the design unit itself.

  - UCDB_INST_ONCE — A flag indicating that there was only one instance of the design unit so there is no code coverage roll-up stored under the design unit. This optimization is less apparent when the UCDB is loaded into memory.

- DUSIGNATURE attribute — This is a crucial attribute used to determine that the code content of the design unit has not changed, so that line number mappings used in all code coverage (except FSM and toggle) are still valid.

Test Data Records and History Nodes

In earlier versions of the database and API, there were only test data records which were designed to record information about the test run which produced the UCDB. It is not possible to create a UCDB without a test data record. When test data records were later extended, they became a special case of the history node. The history node records information about any process that creates a UCDB.

In Questa, there are three ways to create a UCDB:

- By running the simulator — The simulator will create a “test data record” with various information about the simulation run.
- By XML testplan import — This creates a testplan history node.
- By merging — This creates a “merge history node”.

Because of the merging process, whereby UCDBs may be combined in various ways to create other UCDBs, history nodes are arranged in a tree. The test data records and testplan history nodes must be leaves of the tree. But a merge must have child nodes, which are the inputs to the merge. The topology of the tree enables each merge to be reproduced with its original inputs.

The motivation of the history nodes, besides recording interesting information about each test, merge, or testplan, is to enable each of these operations to be reproduced automatically by the tool.

Any of these nodes may have user-defined attributes. Presently user-defined attributes are heavily used with test data records, but they could be used in the other cases, too.

Testplan Hierarchy and Tags

The UCDB has the facility for representing a testplan hierarchy. Ordinarily a testplan is created as a spreadsheet, Word document, or other file and there is some symbolic convention in the tool to link between sections of the testplan and coverage objects in the design. This link could be through fields in the document, through the covergroup comment, through Verilog-2001 attributes, or any other mechanism.

The association in the UCDB is made through a specialized data attribute called a *tag*. A tag is nothing other than a string that is associated with a scope; there may be multiple tags per scope. Any scope or test data record can be tagged. A testplan section is represented by a UCDB scope of type UCDB_TESTPLAN. If it shares a tag with any other scope not of type UCDB_TESTPLAN, the coverage associated with that non-testplan scope is considered linked to the section represented by the testplan scope. After that it is a tool feature to calculate coverage in some way that is meaningful based on the testplan and the coverage linked to it.

Any UCDB scope could be linked with the testplan, not just coverage scopes. However, coveritems may not be linked with the testplan because the tag API does not apply to coveritems.

The testplan example shows a trivial testplan with two sections linked to two trivial coverpoints. Creating a testplan is ordinarily a tool feature, but this example shows how to create one with the API, and introduces the API rather than the data model. In this case, it is not possible to create the data model in an easy-to-understand way without using the API as well as the HDL source.

You can find this C example in *<install_dir>/examples/ucdb/userguide/data-models/testplan/*.

```
ucdbT db = ucdb_Open(ucdbfile);
ucdbScopeT testplan, section1, section2, cvpi, cvpj;
if (db==0) return;

/* Create testplan scopes: */
testplan = ucdb_CreateScope(db,NULL,"testplan",NULL,1,UCDB_NONE,
                                UCDB_TESTPLAN,0);
section1 = ucdb_CreateScope(db,testplan,"section1",NULL,1,UCDB_NONE,
                                UCDB_TESTPLAN,0);
section2 = ucdb_CreateScope(db,testplan,"section2",NULL,1,UCDB_NONE,
                                UCDB_TESTPLAN,0);

/* Look up coverpoint scopes: */
cvpi = ucdb_MatchScopeByPath(db,"/top/cg/i");
cvpj = ucdb_MatchScopeByPath(db,"/top/cg/j");

/* Tag to link testplan scopes to coverpoint scopes */
ucdb_AddObjTag(db,section1,"1");
ucdb_AddObjTag(db,cvpi,"1");
ucdb_AddObjTag(db,section2,"2");
ucdb_AddObjTag(db,cvpj,"2");

/* Write everything back to the same file */
ucdb_Write(db,ucdbfile,NULL,1,-1);
ucdb_Close(db);
```

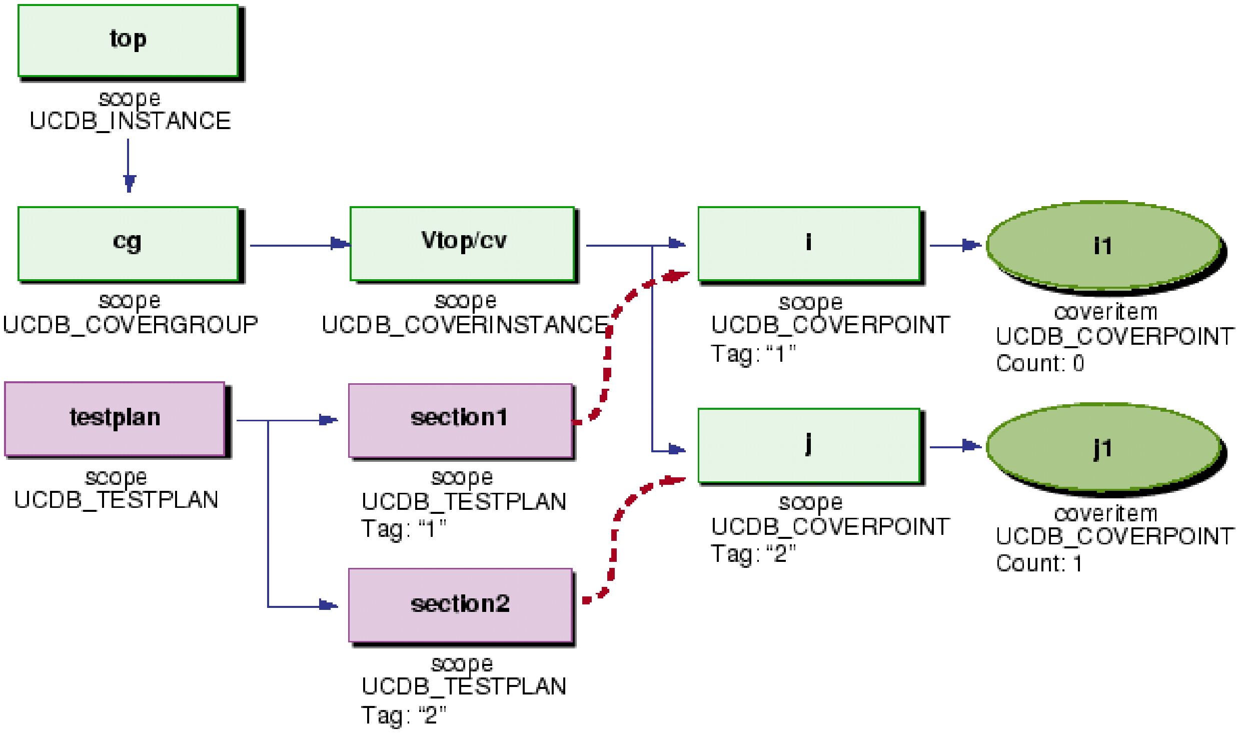
This example executes the following sections of code:

- Open the UCDB file, loading its contents into memory.
- Create the three testplan scopes. The first one, testplan, is used subsequently as the parent of section1 and section2 (the second argument to ucdb_CreateScope() is the parent node). This creates the parent-child relationship and thus the hierarchical structure of the testplan.
- Look up the coverpoint scopes by path. Paths in the UCDB use the path separator (/) by default and otherwise concatenate the names of the scope on a downward traversal through the hierarchy. In this case top is the module instance, cg the covergroup underneath the instance, and i and j the coverpoints underneath the covergroup.
- Give section1 the same tag as /top/cg/i, and section2 the same tag as /top/cg/j. This is all that is necessary to make the testplan association with coverage.

- Write the UCDB data in memory back to the same file from which it was read, and close the UCDB handle in order to de-allocate its memory.

The execution of the C code results in the following data model:

Figure 1-22. Data Model for a Testplan With Linked Coverage



In this case, the shared tag names imply the red dashed-line links from UCDB_TESTPLAN scopes to the UCDB_COVERPOINT scopes. Refer to the section "Traverse from Testplan to Coverage Data with Tags" on page 75 for more information.

In Questa there is a SECTION attribute used in the report, however, tags are composed in a more sophisticated way. Tags are automatically applied by vcover merge according to other attributes attached to the testplans. Also, testplan scopes may have user-defined attributes added from the testplan, so that a testplan data structure created by Questa is more complex than the structure illustrated inFigure 1-22. However, the basic tree relationships are the same.

Memory Statistics

There is a facility in the UCDB API for memory statistics, which are available in constant time when a UCDB is loaded. These statistics are designed for fast access.

The API calls are ucdb_GetMemoryStats() and ucdb_SetMemoryStats() that use the ucdbAttrValueT attribute value structure, but otherwise rely on two enumerators to create, in effect, a 2-dimensional array of attributes. The enumerators are ucdbMemStatsEnumT and ucdbMemStatsTypeEnumT, which are essentially a category and type, respectively.

The memory statistics API is used internally by Questa.

Chapter 2UCDB Use Cases

Understanding the UCDB data models is a prerequisite to using the API. The API is more general than the specific data models used to represent specific kinds of coverage. It is also important to know how to use the API and understand some specific use scenarios.

Note

For more information about UCDB data models, see “UCDB Data Models” on page 15.

UCDB Access Modes **59**

Error Handling **60**

Traverse a UCDB in Memory **61**

Read Coverage Data **62**

Find Objects in a UCDB **65**

Increment Coverage **67**

Remove Data From a UCDB **69**

User-Defined Attributes and Tags in the UCDB **71**

Traverse from Testplan to Coverage Data with Tags **75**

File Representation in the UCDB **77**

Addition of New Data to a UCDB **82**

Test Data Records **92**

Create a UCDB from Scratch in Memory **93**

Read-Streaming Mode **94**

Write-Streaming Mode **96**

UCDB Access Modes

You can open a UCDB file in various modes enabling different methods of analysis.

- **In-memory —** Open the UCDB file so that the entire UCDB data model lies in memory. This is the most general of the use models. All functions related to data access and modification should work in this mode.

- **Read-streaming —** Given a filename, the file is opened and closed within an API function, but you specify a callback that is called for each data record in the database. Effectively this maintains a narrow “window” of data visibility as the database is traversed, so its data access is limited. Some types of data are maintained globally, but the goal of this mode is to minimize the memory profile of the reading application.

- **Write-streaming —** Open a database handle that must be used to write data in a highly specific manner. This is the most difficult mode to perfect, because to successfully write the file, data must be created in requisite order with a precise sequence of API calls, but it has the advantage that the data is streamed to disk without maintaining substantial UCDB data structures, and so minimizes the memory profile of the writing application.
- **Summary read —** This is a constant-time read of a coverage summary stored within the file. This enables overall statistics from the database to be read without traversing the entire database. The disadvantage is that the summary coverage calculations are fixed and cannot be customized in any way.

Use subheadings, if necessary, to distinguish between the different access mode sections.

The database handle type of the UCDB is ucdbT, which is a void* pointing to a hidden structure type of implementation-specific data associated with the database. This handle must be used with nearly all API calls, for example, to open a database in-memory:

```
ucdbT db = ucdb_Open(filename);
```

If the database handle is non-NULL, you can use the open succeeded and the database handle to access all data in the database. The database is not tied in any way to the file on the file system. The database exists entirely in memory, and may be rewritten to the same file or a different file after it is changed.

Writing the database to a file is simple if the database has been previously opened in-memory. The write call can write subsets of the database, characterized by instance subsets or instance tree subsets or by coverage type subsets. The following is the basic write call without using subsets:

```
ucdb_Write(db,filename,NULL,1,-1);
```

The NULL means write the entire database, 1 is a recursive indicator that is irrelevant if NULL is given, and -1 indicates that all coverage types should be written (it is a coverage scope type mask.)

The database in memory is de-allocated with this call:

```
ucdb_Close(db);
```

Error Handling

Most API calls return status or invalid values in case of error. However, these error return cases give no extra information about error circumstances. It is recommended that all standalone applications install their own UCDB error handler. If the API application is linked with Questa, installation of an error handler will not be allowed because Questa already is linked with one.

The basic error handler is similar to the following example. All API contain a basic error handler.

```
void
error_handler(void *data,
              ucdbErrorT *errorInfo)
{
    fprintf(stderr, "%s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCDB_MSG_ERROR)
        exit(1);
}
```

The error-handler is installed as follows:

```
ucdb_RegisterErrorHandler(error_handler, NULL);
```

If there is any user-specific data to pass to the error-handler, a pointer to it is provided instead of NULL and that value is passed as the void* first argument to the callback.

Traverse a UCDB in Memory

The ucdb_CallBack() function is a versatile function that is only available in-memory: given a scope pointer (NULL in this case, meaning traverse the entire database) it traverses everything recursively.

The callback function, called “callback” in this case, is called for every scope, every test record, and every coveritem in the part of the database being traversed. Design units and test data records are only traversed when the entire database is being traversed, as in this case.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/traverse-scopes/*

```
ucdbCBReturnT
callback(
    void*           userdata,
    ucdbCBDataT*    cbdata)
{
    ucdbScopeT scope;
    switch (cbdata->reason) {
    case UCDB_REASON_DU:
    case UCDB_REASON_SCOPE:
        scope = (ucdbScopeT)(cbdata->obj);
        printf("%s\n",ucdb_GetScopeHierName(cbdata->db,scope));
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}

void
example_code(const char* ucdbfile)
{
    ucdbT db = ucdb_Open(ucdbfile);
       if (db==NULL)
        return;
    ucdb_CallBack(db,NULL,callback,NULL);
    ucdb_Close(db);
}
```

The ucdbCBDataT* argument to the callback function gives information about the database object for which the callback is executed. The “reason” element tells what kind of object it is. There are also reasons for end-of-scope (useful for maintaining stacks, so that the callback can know how many levels deep in the design or coverage tree is the current object), the test data records, and the coveritems themselves.

For the scope callbacks, REASON_DU and REASON_SCOPE, the obj element of ucdbCBDataT is identical to a ucdbScopeT, which is a handle to the current scope. In this example, for stylistic reasons, the obj is type-cast explicitly into the scope variable.

The function ucdb_GetScopeHierName() returns a hierarchically composed name for the given scope handle.

Read Coverage Data

The read coverage data example illustrates how to read coverage counts for all coveritems in all instances of a database. This is also based upon the ucdb_CallBack() function for traversing the entire database in memory.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/read-coverage/*.

```
/* Callback to report coveritem count */
ucdbCBReturnT
callback(
    void*           userdata,
    ucdbCBDataT*    cbdata)
{
    ucdbScopeT scope = (ucdbScopeT)(cbdata->obj);
    ucdbT db = cbdata->db;
    char* name;
    ucdbCoverDataT coverdata;
    ucdbSourceInfoT sourceinfo;

    switch (cbdata->reason) {
    case UCDB_REASON_DU:
        /* Do not traverse data under a DU: see read-coverage2 */
        return UCDB_SCAN_PRUNE;
    case UCDB_REASON_CVBIN:
        scope = (ucdbScopeT)(cbdata->obj);
        /* Get coveritem data from scope and coverindex passed in: */
        ucdb_GetCoverData(db,scope,cbdata->coverindex,
                          &name,&coverdata,&sourceinfo);
        if (name!=NULL && name[0]!='\0') {
            /* Coveritem has a name, use it: */
            printf("%s%c%s: ",ucdb_GetScopeHierName(db,scope),
                   ucdb_GetPathSeparator(db),name);
        } else {
            /* Coveritem has no name, use [file:line] instead: */
            printf("%s [%s:%d]: ",ucdb_GetScopeHierName(db,scope),
                   ucdb_GetFileName(db,&sourceinfo.filehandle),
                   sourceinfo.line);
        }
        print_coverage_count(&coverdata);
        printf("\n");
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}
```

If a design unit scope is encountered in the callback, the UCDB_SCAN_PRUNE return value instructs the callback generator to skip further callbacks for data structures underneath the design unit.

The callback prints something for the UCDB_REASON_CVBIN callback. The UCDB_REASON_CVBIN callback is for coveritems in the data model. The cbdata->obj value is set to the parent scope of the coveritem, and cbdata->coverindex is the index that can be used to access the cover item. Data for the coveritem is accessed with ucdb_GetCoverData(). This retrieves the name, coverage data, and source information for the coveritem. The source information is essential sometimes because some coverage objects, specifically, statement coveritems, do not have names: they can only be identified by the source file, line, and token with which they are associated.

The coverage data itself is printed in this function:

```
void
print_coverage_count(ucdbCoverDataT* coverdata)
{
    if (coverdata->flags & UCDB_IS_32BIT) {
        /* 32-bit count: */
        printf("%d", coverdata->data.int32);
    } else if (coverdata->flags & UCDB_IS_64BIT) {
        /* 64-bit count: */
        printf("%lld", coverdata->data.int64);
    } else if (coverdata->flags & UCDB_IS_VECTOR) {
        /* bit vector coveritem: */
        int bytelen = coverdata->bitlen/8 + (coverdata->bitlen%8)?1:0;
        int i;
        for ( i=0; i<bytelen; i++ ) {
            if (i) printf(" ");
            printf("%02x",coverdata->data.bytevector[i]);
        }
    }
}
```

This function comprehensively shows how the coverage count must be printed. There are not currently any source inputs or tools that create the UCDB_IS_VECTOR type of coverage data, but 32-bit and 64-bit platforms each create counts of their respective integer sizes.

read-coverage2 Example

The read-coverage2 example shows you how to handle traversing the code coverage data underneath a design unit.

The problem is the UCDB_INST_ONCE optimization where coverage data for a single-instance design unit is stored only in the instance. For a per-design-unit coverage roll-up, it is convenient to access data through the UCDB design unit scope. The UCDB API enables this data access. However, the problem comes when printing the path to those scopes that were accessed underneath the design unit. Because the data is actually stored underneath the instance, the path prints the same whether it was accessed through the design unit or not. Extra code must be written to determine how the data was accessed: via the design unit or through the instance tree.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/read-coverage2/*.

```
    struct dustate* du = (struct dustate*)userdata;

    switch (cbdata->reason) {
    /*
     *  The DU/SCOPE/ENDSCOPE logic distinguishes those objects which
occur
     *  underneath a design unit.  Because of the INST_ONCE optimization,
it is
     *  otherwise impossible to distinguish those objects by name.
     */
    case UCDB_REASON_DU:
        du->underneath = 1; du->subscope_counter = 0; break;
    case UCDB_REASON_SCOPE:
        if (du->underneath) {
            du->subscope_counter++;
        }
        break;
    case UCDB_REASON_ENDSCOPE:
        if (du->underneath) {
            if (du->subscope_counter)
                du->subscope_counter--;
            else
                du->underneath = 0;
        }
        break;
```

The “du” user data pointer has “underneath,” which is a flag that is 1 while underneath a design unit, and a “subscope_counter” for subscopes underneath the design unit. (FSM coverage, for example, will create subscopes underneath a design unit.) Then if du->underneath is true, the application can print something distinctive to indicate when a coveritem was really found through the design unit rather than the instance:

```
read_coverage ../../data-models/toggle-enum/test.ucdb
/top/t/a: 0 (FROM DU)
/top/t/b: 1 (FROM DU)
/top/t/c: 1 (FROM DU)
/top/t/a: 0
/top/t/b: 1
/top/t/c: 1
```

Find Objects in a UCDB

The easiest way in the UCDB API to find particular objects by name in the database is the ucdb_PathCallBack() function. It has the added advantage of handling wildcards for multiple characters (*) and a single character (?) in individual path component names.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/find-object/*.

```
ucdbCBReturnT
callback(
    void*           userdata,
    ucdbCBDataT*    cbdata)
{
    switch (cbdata->reason) {
    case UCDB_REASON_SCOPE:
        print_scope(cbdata->db,(ucdbScopeT)(cbdata->obj));
        break;
    case UCDB_REASON_CVBIN:
        print_coveritem(cbdata->db,(ucdbScopeT)(cbdata->obj),
                        cbdata->coverindex);
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}

void
example_code(const char* ucdbfile, const char* path)
{
    ucdbT db = ucdb_Open(ucdbfile);
    if (db==NULL)
        return;
    ucdb_PathCallBack(db,
                      0,     /* do not recurse from found object */
                      path,
                      NULL, /* design unit name does not apply */
                      UCDB_NONTESTPLAN_SCOPE,   /* tree root type */
                      -1,    /* match any scope type */
                      -1,    /* match any coveritem type */
                      callback, NULL);
    ucdb_Close(db);
}
```

The arguments to ucdb_PathCallBack(), in order, are the following:

- A database handle that must be opened with ucdb_Open().
- A recursion flag. In this case, you are looking for scopes and not what is underneath them, so the recursion is false.
- The path passed in from the command line of the example.
- The design unit name is NULL because it does not apply to the intent here. Paths could be design-unit-relative. In that case, the design unit name must be given.
- The tree root type must be given to distinguish between the two basic types of trees available in the UCDB: the testplan tree or the design instance tree.
- The scope mask restricts the search to particular scope types; -1 in this case means all scope types.

- The cover mask restricts the search to particular coveritem types; -1 in this case means all coveritem types. An alternative is to set this value to 0, in which case only scopes would be matched and not coveritems at all.
- The callback function.
- The private data for the callback function.

The print_scope() and print_coveritem() functions use scope or coveritem names, types, and line numbers to display data about the object found in the database. Statement coveritems will never be found by this API because they have no names at all. Only a linear search by filename, line number, and token number could find a particular statement coveritem.

The “sink” design supplied with *examples/ucdb/ucdbcrawl* has many different types of coverage in it. This design illustrates using the find_object example with a pattern that is known to have multiple matches:

```
./find_object ../../../ucdbcrawl/sink.ucdb '/top/mach/state/*'
Found scope '/top/mach/state/states': type=20000000 line=33
Found scope '/top/mach/state/trans': type=40000000 line=33
Found cover '/top/mach/state/st0': types=00000001/00000200 line=0
Found cover '/top/mach/state/st1': types=00000001/00000200 line=0
Found cover '/top/mach/state/st2': types=00000001/00000200 line=0
Found cover '/top/mach/state/st3': types=00000001/00000200 line=0
```

In this case, “/top/mach/state” is both an FSM scope and a toggle scope. When matching all children with “*”, this matches the transition and state child scopes of the FSM scope, and the enum toggle bins. Source information for toggles is stored at the scope level (not at the bin level). Therefore, the output for the toggle bins shows line=0.

Increment Coverage

You can apply ucdb_IncrementCover to statement coveritems if their parent scopes are identified. To increment a coveritem multiple times, it is recommended that you save a scope pointer and coverindex for later use. The ucdb_PathCallBack() approach has the disadvantage that it only recognizes named items, which excludes statement coveritems.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/increment-cover/*

```
ucdbCBReturnT
callback(
    void*           userdata,
    ucdbCBDataT*    cbdata)
{
    switch (cbdata->reason) {
    case UCDB_REASON_CVBIN:
      ucdb_IncrementCover(cbdata->db,(ucdbScopeT)(cbdata->obj),
                                cbdata->coverindex,1);
        return UCDB_SCAN_STOP;
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}

void
example_code(const char* ucdbfile, const char* path)
{
    ucdbT db = ucdb_Open(ucdbfile);
    if (db==NULL)
        return;
    ucdb_PathCallBack(db,
                      0,    /* do not recurse from found object */
                      path,
                      NULL, /* design unit name does not apply */
                      UCDB_NONTESTPLAN_SCOPE,   /* tree root type */
                      -1,   /* match any scope type */
                      -1,   /* match any coveritem type */
                      callback, NULL);
    ucdb_Write(db,ucdbfile,
               NULL,    /* save entire database */
               1,       /* recurse: not necessary with NULL */
               -1);     /* save all scope types */
    ucdb_Close(db);
}
```

The callback in this case uses the UCDB_SCAN_STOP return code to avoid iterating over the entire database: the iteration is halted after recognizing the coveritem to increment.

The example_code() function illustrates saving the UCDB back to its original file. The original file is closed by the operating system after ucdb_Open() completes, so there is no link between the open UCDB handle “db” and the original file. The UCDB can be changed and written back to the same file or any other file.

The following are the ucdb_Write() arguments in order

“db” and “ucdbfile” are obvious, the others less so:

- db
- ucdbfile

- Third argument (NULL) — a scope from which to execute the save; if NULL, save the entire database.
- Fourth argument (1) — a recursion flag, only needed if the scope handle in the previous argument is non-NULL.
- Fifth argument (-1) — a scope mask, to indicate which scopes to save to the database. You can use this argument to create a database with functional coverage only, or code coverage only.

Remove Data From a UCDB

The ucdb_RemoveScope() and ucdb_RemoveCover() functions are used to delete objects from the database.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/remove-data/*

```
ucdbCBReturnT
callback(
    void*               userdata,
    ucdbCBDataT*        cbdata)
{
    int rc;
    ucdbScopeT scope = (ucdbScopeT)(cbdata->obj);
    ucdbT db = cbdata->db;
    char* name;
    switch (cbdata->reason) {
    case UCDB_REASON_SCOPE:
        printf("Removing scope %s\n",ucdb_GetScopeHierName(db,scope));
        ucdb_RemoveScope(db,scope);
        return UCDB_SCAN_PRUNE;
    case UCDB_REASON_CVBIN:
        ucdb_GetCoverData(db,scope,cbdata->coverindex,&name,NULL,NULL);
        printf(
           Removing cover %s/%s\n",ucdb_GetScopeHierName(db,scope),name);
        rc = ucdb_RemoveCover(db,scope,cbdata->coverindex);
        if (rc!=0) {
            printf("Unable to remove cover %s/%s\n",
                   ucdb_GetScopeHierName(db,scope), name);
        }
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}
void
example_code(const char* ucdbfile, const char* path)
{
    ucdbT db = ucdb_Open(ucdbfile);
    int matches;
    if (db==NULL)
        return;
    matches = ucdb_PathCallBack(
                           db,
                           0,    /* do not recurse from found object */
                           path,
                           NULL, /* design unit name does not apply */
                           UCDB_NONTESTPLAN_SCOPE,   /* tree root type */
                           -1,   /* match any scope type */
                           -1,   /* match any coveritem type */
                           callback, NULL);
    if (matches==0)
        printf("No matches for path\n");
    else
        ucdb_Write(db,ucdbfile,
                   NULL,    /* save entire database */
                   1,       /* recurse: not necessary with NULL */
                   -1);     /* save all scope types */
    ucdb_Close(db);
}
```

Note

This example does not work with wildcards.

There is a limitation on ucdb_RemoveCover() in that it cannot delete toggle bins for the most common types: the 2-state and 3-state wires and registers. This limitation exists because toggle bins are optimized and do not really exist in isolation. The toggle scope can be deleted, but not individual bins in that case. The error handler in this example does not call exit() but allows the code to continue; otherwise there is an internal API error generated for trying to remove a toggle scope of these types. Also, the return code from ucdb_RemoveCover() is checked to provide an error message with a specific path to the object whose removal failed.

Note

When a scope is removed, all its children are removed, too.

This example also checks the return code from ucdb_PathCallBack() to indicate when no objects were matched by the given path. Otherwise, the application would remain silent.

Caution

It is possible to delete an FSM transition scope and leave a set of transitions that could be inconsistent with the state values for the same FSM.

User-Defined Attributes and Tags in the UCDB

Tags are names that are associated with scopes and test data records in the database. You can use these names could for general purpose grouping in the database.

Tags in the UCDB

In Questa, tags are used for making test traceability associations.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/print-attrtags/*

```
void
print_tags(ucdbT db, ucdbScopeT scope)
{
    int i, ntags = ucdb_GetObjNumTags(db,(ucdbObjT)scope);
    const char* tagname;
    printf("Tags for %s:\n",ucdb_GetScopeHierName(db,scope));
    if (ntags > 0) {
        for ( i=0; i<ntags; i++ ) {
            ucdb_GetObjIthTag(db,(ucdbObjT)scope,i,&tagname);
            printf("%s ",tagname);
        }
        printf("\n");
    }
}
```

This example uses an integer-based iterator. The number of tags are acquired with ucdb_GetObjNumTags, then the function ucdb_GetObjIthTag() is used to acquire the tag name for the i-th tag. Because these functions operate on both scopes (ucdbScopeT) and test data records (ucdbTestT), there is a polymorphic type ucdbObjT that can stand for both. Some functions (queries as to object type or kind, queries about tags, and queries about attributes) take these object handles rather than scope or test data record handles. However, because this is C and not C++, all these types are really void*, so they are interchangeable and type unsafe. In this example, the cast with "(ucdbObjT)" is used for readability; it is not strictly necessary.

User-Defined Attributes in the UCDB

User-defined attributes are also names that can be associated with a UCDB object, but are more powerful than tags in what they can represent.

- They can appear with any type of object in the database: test data records, scopes, and coveritems.
- There is a class of attributes, where NULL is given as the ucdbObjT handle to the API calls, that are called global or UCDB attributes. These are not associated with any particular object in the database but instead are associated with the database itself.
- User-defined attributes have values as well as names. The names are the "key" for the values. You can look up a value by name.
- Attribute values can be of five different types:
    - 32-bit integer
    - 32-bit floating point (float).
    - 64-bit floating point (double).
    - Null-terminated string.
    - A byte stream of any number of bytes with any values. This is useful for storing unprintable characters or binary values that might contain 0 (and thus cannot be stored as a null-terminated string.)

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/read-attrtags/*.

```
void
print_attrs(ucdbT db, ucdbScopeT scope, int coverindex)
{
    const char* attrname;
    ucdbAttrValueT* attrvalue;
    char* covername;
    printf("Attributes for %s",ucdb_GetScopeHierName(db,scope));
    if (coverindex>=0) {
        ucdb_GetCoverData(db,scope,coverindex,&covername,NULL,NULL);
        printf("%c%s:\n",ucdb_GetPathSeparator(db),covername);
    } else {
        printf(":\n");
    }
    attrname = NULL;
    while ((attrname = ucdb_AttrNext(db,(ucdbObjT)scope,coverindex,
                                     attrname,&attrvalue))) {
        printf("\t%s: ", attrname);
        switch (attrvalue->type)
        {
        case UCDB_ATTR_INT:
            printf("int = %d\n", attrvalue->u.ivalue);
            break;
        case UCDB_ATTR_FLOAT:
            printf("float = %f\n", attrvalue->u.fvalue);
            break;
        case UCDB_ATTR_DOUBLE:
            printf("double = %lf\n", attrvalue->u.dvalue);
            break;
        case UCDB_ATTR_STRING:
            printf("string = '%s'\n",
                   attrvalue->u.svalue ? attrvalue->u.svalue : "(null)");
            break;
        case UCDB_ATTR_MEMBLK:
            printf("binary, size = %d ", attrvalue->u.mvalue.size);
            if (attrvalue->u.mvalue.size > 0) {
                int i;
                printf("value = ");
                for ( i=0; i<attrvalue->u.mvalue.size; i++ )
                    printf("%02x ", attrvalue->u.mvalue.data[i]);
            }
            printf("\n");
            break;
        default:
            printf("ERROR! UNKNOWN ATTRIBUTE TYPE (%d)\n",
                     attrvalue->type);
        } /* end switch (attrvalue->type) */
    } /* end while (ucdb_AttrNext(...)) */
}
```

This iterator requires a loop like the following:

```
    attrname = NULL;
    while ((attrname = ucdb_AttrNext(db,(ucdbObjT)scope,coverindex,
                                     attrname,&attrvalue))) {
```

The assignment of attrname to NULL is crucial; it starts the iteration. (A common bug in this case is to leave the attrname variable uninitialized. If it happens to be 0, the loop may execute, otherwise it will behave unpredictably, either crashing or doing nothing.)

If the attribute is for a scope, coveritem==(-1). If the attribute is for a test data record, the second (ucdbObjT) argument must be a ucdbTestT handle. If the attribute is for the UCDB as a whole, the second argument must be NULL.

The same attribute name as was returned by ucdb_AttrNext() must be passed to the function for the next iteration. The ucdbAttrValueT* variable must be declared by you and is set by ucdb_AttrNext(). This variable is changed to point to memory owned by the API.

The example code for this section must switch on attrvalue->type to print something appropriate for the attribute value of the given type.

Some of the following examples for adding data to a UCDB show how to write user-defined attributes. To write them, you must create your own memory for the attribute value(s); this memory is copied for the API's purposes to store with the UCDB.

Predefined Attribute Names in the UCDB

The *ucdb.h* header uses #defines of this form:

```
#define UCDBKEY_SIMTIME         "SIMTIME"
```

Note

Any of the macros starting with UCDBKEY are predefined attribute names. You may reuse these attribute names in different scopes, but it is inadvisable to reuse these attribute names in the same scopes in which Questa itself creates them.

The built-in attributes created by Questa must be read or written with the same API as for any user-defined attribute. For test data records only, built-in attributes may also be read or written with the API functions ucdb_GetTestData() and ucdb_AddTest().

Create a Testplan in a UCDB

The data model example discussed in the section “Test Data Records and History Nodes” on page 55 shows how to create a testplan from scratch. The following are some important features of the testplans created by Questa:

- Tag names for testplan sections in Questa are a concatenation of the testplan root name and the section number. This guarantees that testplans can be merged together from different files.
- The Questa tag CLI (viewcov mode command-line interface) is actually embedded as a user-defined attribute in the testplan scope, with UCDBKEY_TAGCMD as the name.

The value is a string of semicolon-separated list of arguments to the coverage tag commands; these commands are automatically executed by vcover merge.

- Testplan sections have the UCDBKEY_SECTION attribute set to the literal section number that must appear in the report.
- The XML import for testplans can create any arbitrary user-defined attributes from the testplan source. These attributes then appear in the GUI and can be used as search criteria with the CLI or GUI.

Traverse from Testplan to Coverage Data with Tags

The ucdb_PathCallBack() function automatically performs a recursive traversal of testplan scopes, and for each testplan scope, it pursues the linked objects that share the same tag. The function considers the linked objects to be virtual children of the testplan scope.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/traverse-testplan/*.

```
void
recurse_testplan(int level, ucdbT db, ucdbScopeT scope)
{
    int t, numtags;
    const char* tagname;
    ucdbScopeT subscope;

    /* Print testplan scope name and recurse child testplan sections */
    indent(level);
    printf("%s\n",ucdb_GetScopeName(db,scope));
    subscope=NULL;
    while ((subscope=ucdb_NextSubScope(db,scope,subscope,UCDB_TESTPLAN)))
{
        recurse_testplan(level+1,db,subscope);
    }

   /* from ucdb.h: traverse non-testplan objects with the same tag name */
    numtags = ucdb_GetObjNumTags(db,(ucdbObjT)scope);
    for ( t=0; t<numtags; t++ ) {
        int found;
        ucdbObjT taggedobj;
        ucdb_GetObjIthTag(db,(ucdbObjT)scope,t,&tagname);
        for ( found=ucdb_BeginTaggedObj(db,tagname,&taggedobj);
              found; found=ucdb_NextTaggedObj(db,&taggedobj) ) {
            if (ucdb_ObjKind(db,taggedobj)==UCDB_OBJ_SCOPE
                &&
             ucdb_GetScopeType(db,(ucdbScopeT)taggedobj)==UCDB_TESTPLAN)
                continue;
            /* tagged object is not a testplan scope: */
            indent(level+1);
            if (ucdb_ObjKind(db,taggedobj)==UCDB_OBJ_SCOPE)
          printf("%s\n",ucdb_GetScopeHierName(db,(ucdbScopeT)taggedobj));
            else if (ucdb_ObjKind(db,taggedobj)==UCDB_OBJ_TESTDATA)
               printf("%s\n",ucdb_GetTestName(db,(ucdbTestT)taggedobj));
        }
    }
}

void
example_code(const char* ucdbfile)
{
    ucdbScopeT subscope;
    ucdbT db = ucdb_Open(ucdbfile);
    if (db==NULL)
        return;
    subscope=NULL;
   while ((subscope=ucdb_NextSubScope(db,NULL,subscope,UCDB_TESTPLAN))) {
        recurse_testplan(0,db,subscope);
    }
    ucdb_Close(db);
}
```

The ucdb_NextSubScope function is an iterator that must start with a NULL pointer. One common mistake with this iterator is to confuse the scope and subscope. The traversal in example_code() is a traversal of roots, because NULL is given as the scope. The subscopes are returned, but these are root scopes with no parent. The last argument to ucdb_NextSubScope, the UCDB_TESTPLAN value, is a scope mask. This is one of those cases where the scope type is used as a mask. In this case, the scope type is used as a mask, and the implementation is that each scope type occupies one and only one bit. The iterator will only return testplan scopes.

The recurse_testplan() scope prints the testplan scope name with indentation and recurses into testplan sub-scopes.

The complex second loop in recurse_testplan() is taken from an example in the *ucdb.h* header, which acquires each tag from the testplan. (Even though Questa creates testplan sections with one and only one tag, the UCDB has no such restriction in its data model.) The ucdb_BeginTaggedObj() and ucdb_NextTaggedObj() use the tag name to return the list of objects that share the tag. Tagged objects may be either scopes, testplan scopes, module instance scopes, coverage scopes, design unit scopes, and so on, or test data records. The ucdb_GetScopeType() function may only be used with scopes, so ucdb_ObjKind() is used first to guarantee that the object is a scope.

If the loop drops through the continue statement, the current object (taggedobj) shares a tag with the current testplan scope and is not itself a testplan scope. The names are printed in in the following ways: one for other scopes and one for test data records.

The end result is a simpler version of the “coverage analyze -plan / -r” command that can be used in viewcov mode in Questa. This result is essentially the logic followed by the coverage analyze command. The coverage analyze command relies on ucdb_PathCallBack(), which has the traversal logic built-in.

File Representation in the UCDB

File representation in the UCDB is designed to be efficient and capable. For efficiency, inside most objects in the database source file information is stored as: file number, line number, and token number. File numbers need to relate to a file table. The UCDB has various ways to create a file table, implicitly or explicitly.

With Questa itself, file tables are stored with design units. The file number is then the index into the file table of the design unit to which the object belongs. However, in general, filehandles may be mixed and matched among different file tables. The example for this section (filehandles) uses both the design-unit file table created by Questa and a global one created implicitly through the API.

For capability, a file is specified as two parts: the directory to which the file belongs and the relative path to the file itself. This enables a heuristic algorithm to try to find the file even if the UCDB has been moved. A “heuristic” is not guaranteed to work. The heuristic includes possible use of a Questa-specific environment variable (MGC_WD) that can be used explicitly to point to source if the original directory no longer exists. Additionally, there is the MGC_LOCATION_MAP feature that allows mapping of directory prefixes between different networks so that Questa files can be portable between different file systems. The UCDB implementation will make use of MGC_LOCATION_MAP features if present.

Creating a Filehandle From a Filename

For this example, Questa itself manages the file table. This behavior has the disadvantage that each time a filehandle is created by name, there is a string-based lookup to ensure that the file table contains only unique names.

You can find this C example in *<install_dir>/examples/ucdb/userguide/use-cases/filehandles/create_filehandles.c*.

```
void
create_statement_with_filehandle(ucdbT db,
                 ucdbScopeT parent,
                 ucdbFileHandleT filehandle,
                 int line,
                 int count)
{
    ucdbCoverDataT coverdata;
    ucdbSourceInfoT srcinfo;
    ucdbAttrValueT attrvalue;
    int coverindex;
    coverdata.type = UCDB_STMTBIN;
    coverdata.flags = UCDB_IS_32BIT;    /* data type flag */
    coverdata.data.int32 = count;       /* must be set for 32 bit flag */
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0;                  /* fake token # */
    coverindex = ucdb_CreateNextCover(db,parent,
                                   NULL, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
...
}
example_code(const char* ucdbfile)
{
...
   /* Let UCDB API create a global file table for each unique filename: */
    ucdb_CreateSrcFileHandleByName(db,&filehandle,
                                 NULL,    /* let API create file table */
                                   "test.sv",
                                   pwd);
   create_statement_with_filehandle(db,instance,filehandle,3,1);
...
```

In this example, the ucdb_CreateSrcFileHandleByName() takes these arguments:

- Database
- Filehandle to be filled in.
- Path to scope in which file table is to reside. If NULL, that means a global file table. A global file is most efficient, but Questa does not use this because it does per-design-unit compilation and much of its source information is oriented around the design unit.
- Name of file.

- Directory in which the file is found. This example relies on a “PWD” environment.

The filehandle is assigned to the ucdbSourceInfoT structure. The structure contains other information for line number and token number. This structure is passed to API functions like ucdb_CreateNextCover() and ucdb_CreateScope() and ucdb_CreateInstance(), which create new objects in the database. For details on the creation of new objects, refer to “Addition of New Data to a UCDB” on page 82.

The token number is difficult to use unless you have access to a tokenizer (lexical analyzer) for each source language of interest.

Creating a Filehandle From an Existing File Table

You can find this example in <*install_dir*>*/examples/ucdb/userguide/use-cases/filehandles/test.sv*

```
module top;
        initial begin
                // $display("hello");
                // $display("there");
        `include "test2.sv"
        end
endmodule
```

You can find this example in <*install_dir*>*/examples/ucdb/userguide/use-cases/filehandles/test2.sv*

```
        // $display "world";
```

Even though these source files have commented-out statements, the compiler did parse the code, and Questa did create a file table inside the “work.top” design unit that has two entries. The first entry is “test.sv” and the second entry is “test2.sv”. Consequently, this code can be used to create statements that use filehandles from the existing design unit file table:

You can find this example in <*install_dir*>*/examples/ucdb/userguide/use-cases/filehandles/create_filehandles.c*

```
void
create_statement_with_filenumber(ucdbT db,
                 ucdbScopeT parent,
                 ucdbScopeT filetable_scope,
                 int filenumber,
                 int line,
                 int count)
{
    ucdbCoverDataT coverdata;
    ucdbSourceInfoT srcinfo;
    ucdbFileHandleT filehandle;
    ucdbAttrValueT attrvalue;
    int coverindex;

ucdb_CreateFileHandleByNum(db,&filehandle,filetable_scope,filenumber);
    coverdata.type = UCDB_STMTBIN;
    coverdata.flags = UCDB_IS_32BIT;    /* data type flag */
    coverdata.data.int32 = count;       /* must be set for 32 bit flag */
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0;                  /* fake token # */
    coverindex = ucdb_CreateNextCover(db,parent,
                                   NULL, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
...
    /* Re-use file table from DU: */
    create_statement_with_filenumber(db,instance,du,0,4,1);
    create_statement_with_filenumber(db,instance,du,1,1,1);
...
```

This is the more efficient approach to creating a filehandle. It requires a handle to the scope containing the file table (or NULL if using a global file table). The function ucdb_CreateFileHandleByNum() is used to create a filehandle from the given file table.

This creates two statements:

- First statement from file 0 (“test.sv”) from du's file table, at line 4, with count 1.
- Second statement from file 1 (“test2.sv”) from du's file table, at line 1, with count 1.

There are other ways to create filehandles, as well. For example, the ucdb_CloneFileHandle() function can be used if you do not have access to the scope containing the file table, but only have access to a valid filehandle. You can clone the filehandle, which means to use the same file table, but with a different file number, such as a different offset into the table.

The example did not show how to create the file table because that was already completed by Questa for the design unit. To create the file table, use ucdb_SrcFileTableAppend() for each successive file.

Dumping File Tables

Access a filename from a file table using ucdb_GetFileName(). The ucdb_GetFileName() was used in the “read-coverage” example as a way to identify a statement bin, because statement bins have no names.

The dump_filehandles example shows how to dump file tables throughout a database.

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/filehandles/dump_filehandles.c*.

```
void
dump_filetable(ucdbT db, ucdbScopeT scope)
{
   int file;
   for ( file=0; file<ucdb_FileTableSize(db,scope); file++ ) {
      if (file==0) {
         if (scope)
            printf("File Table for '%s':\n",
                  ucdb_GetScopeHierName(db,scope));
         else
            printf("Global File Table:\n");
      }
      printf("\t%s\n", ucdb_FileTableName(db,scope,file));
   }
}

ucdbCBReturnT
callback(
    void*           userdata,
    ucdbCBDataT*    cbdata)
{
    switch (cbdata->reason) {
    case UCDB_REASON_DU:
    case UCDB_REASON_SCOPE:
      dump_filetable(cbdata->db,(ucdbScopeT)(cbdata->obj));
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}

void
example_code(const char* ucdbfile)
{
    ucdbT db = ucdb_Open(ucdbfile);
   printf("Dumping file tables for '%s' ...\n", ucdbfile);
   dump_filetable(db,NULL);
   ucdb_CallBack(db,NULL,callback,NULL);
    ucdb_Close(db);
}
```

The global file table is dumped, with scope==NULL. Any scope can have a file table, except for toggle scopes, which have limited capability for space efficiency (because there are potentially many toggles). There are some limitations on where a filehandle may be used for a given file table. Basically, the scope with the file table must be an ancestor in the UCDB hierarchy relative to the object that refers to it with a filehandle.

The function ucdb_FileTableName() is for dumping the filename directly from the table. The same name could be acquired indirectly by using ucdb_CreateFileHandleByNum() to get a filehandle from the table, and then ucdb_GetFileName() to get a name from the filehandle. The ucdb_FileTableName() function dumps the filename in a single step.

In the example, there are only two file tables: the one created by Questa in the design unit, and the global one created by “create_filehandles” that partially overlaps the design unit table:

```
Dumping file tables for 'test.ucdb' ...
Global File Table:
        test.sv
File Table for 'work.top':
        test.sv
        test2.sv
```

Addition of New Data to a UCDB

The single complex example “create-ucdb/create_ucdb.c” creates a hardcoded UCDB from scratch. The code that it uses could be adapted, with variations, to add objects to an existing UCDB. Even in the “create_ucdb.c” example, the database exists; it just starts out empty and is added to with each call.

The example is not exhaustive. Statements, an enum toggle, and a covergroup are created as an illustration. To create other types of objects, refer to “UCDB Data Models” on page 15. It also may help to reverse-engineer UCDB data created by Questa using the ucdbdump example from *examples/ucdb/ucdbdump*.

Add Design Unit to a UCDB

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/create-ucdb/*

```
ucdbScopeT
create_design_unit(ucdbT db,
                   const char* duname,
                   ucdbFileHandleT file,
                   int line)
{
    ucdbScopeT duscope;
    ucdbSourceInfoT srcinfo;
    ucdbAttrValueT attrvalue;
    srcinfo.filehandle = file;
    srcinfo.line = line;
    srcinfo.token = 0;                              /* fake token # */
    duscope = ucdb_CreateScope(db,
                            NULL,           /* DUs never have a parent */
                               duname,
                               &srcinfo,
                               1,                /* weight */
                               UCDB_VLOG,        /* source language */
                               UCDB_DU_MODULE,   /* scope type */
                               /* flags: */
                               UCDB_ENABLED_STMT | UCDB_ENABLED_BRANCH |
                               UCDB_ENABLED_COND | UCDB_ENABLED_EXPR |
                               UCDB_ENABLED_FSM | UCDB_ENABLED_TOGGLE |
                               UCDB_INST_ONCE | UCDB_SCOPE_UNDER_DU);
    attrvalue.type = UCDB_ATTR_STRING;
    attrvalue.u.svalue = "FAKE DU SIGNATURE";
    ucdb_AttrAdd(db,duscope,-1,UCDBKEY_DUSIGNATURE,&attrvalue);
    return duscope;
}
```

Design units must be created before their corresponding instances. Design units come in five types:

- **UCDB_DU_MODULE —** Verilog or SystemVerilog module
- **UCDB_DU_ARCH —** VHDL architecture
- **UCDB_DU_PACKAGE —** Verilog, SystemVerilog or VHDL package
- **UCDB_DU_PROGRAM —** SystemVerilog program block
- **UCDB_DU_INTERFACE —** SystemVerilog interface

One crucial fact about all these types, except packages, is that differently parameterized versions of the same design unit are merged together by Questa when saving a UCDB. This is because different parameterizations may be created arbitrarily and capriciously by the optimizer. The Structure window in Questa shows these parameterizations, but when a UCDB is loaded into the Coverage View mode GUI, the Structure window shows only the canonical module, architecture, and so forth.

Questa does not use the UCDB_SV language type except for types of objects peculiar to SystemVerilog (such as interfaces.) A module will always have the UCDB_VLOG language type.

The flags for the design unit have the requirement that in order for the Questa reports to work correctly the flags must be turned on to correspond to the different types of code coverage that have been compiled for the design unit. If these flags are not present, the report will not recognize the corresponding code coverage type.

The UCDB_INST_ONCE flag is hardcoded in this case, but you are responsible for maintaining it. If you add an instance to a design unit that already has a single instance, the flag must be cleared. In this example, it is assumed that the design unit will only ever have a single instance.

The flag UCDB_SCOPE_UNDER_DU is required for certain coverage CLI commands and summary data to work correctly: it supplies the implementation for ucdb_ScopeIsUnderDU() and has implications for ucdb_CalcCoverageSummary(). If the flag is not set, some design-unit-oriented coverage may be mistaken as being per-instance.

The UCDBKEY_DUSIGNATURE attribute is required to detect source code changes for the files associated with the design unit.

The Questa implementation of the signature is not available as a public API. If a valid signature is not computed by the API user, it has implications for the merge. If UCDBs from the same design source are merged together, there will be no problem, but the potential problem of merging files from different source would not be detected. (Merging from different source is a problem for the UCDB because most code coverage objects, with the exception of FSMs and toggles, are identified by source code only; that is, by some combination of file, line, and token number.)

The weight of a design unit has relevance to the Questa coverage analyze command and the Test Tracking GUI.

Add Module Instance to a UCDB

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/create-ucdb/*.

```
ucdb_CreateInstance(db,parent,instname,
   NULL,           /* source info: not used in Questa */
   1,              /* weight */
   UCDB_VLOG,      /* source language */
   UCDB_INSTANCE,  /* instance of module/architecture */
   duscope,        /* reference to design unit */
   UCDB_INST_ONCE);/* flags */
```

Because the UCDB is a hierarchical data structure, the parent must be given. Setting the value to NULL creates the instance at the top-level; that is, creates it as root. This implicitly adds the new instance underneath the parent.

The instance name (instname) will become part of the path to identify the instance in the UCDB hierarchy. If the name contains odd characters, it is good practice to turn it into an escaped (or extended) identifier to enable path searching in Questa to work properly. The escaped identifier syntax will be VHDL style for instances under a VHDL parent and Verilog style for instances under a Verilog parent.

Source information may be given.

The weight may be relevant to the coverage analyze command and the Test Tracking GUI.

The scope type (UCDB_INSTANCE in this case) must map correctly to the given design unit type:

- UCDB_INSTANCE for design unit type of UCDB_DU_MODULE or UCDB_DU_ARCH.
- UCDB_PACKAGE for design unit type of UCDB_DU_PACKAGE.
- UCDB_INTERFACE for design unit type of UCDB_DU_INTERFACE.
- UCDB_PROGRAM for design unit type of UCDB_DU_PROGRAM.

The UCDB_INST_ONCE flag is set only for a single instance of a given design unit. If adding an additional instance, you must clear the flag explicitly.

```
ucdb_SetScopeFlag(db,scope,UCDB_INST_ONCE,0);
```

Add Statement to a UCDB

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/create-ucdb/*.

```
void
create_statement(ucdbT db,
                 ucdbScopeT parent,
                 ucdbFileHandleT filehandle,
                 int line,
                 int count)
{
    ucdbCoverDataT coverdata;
    ucdbSourceInfoT srcinfo;
    ucdbAttrValueT attrvalue;
    int coverindex;
    coverdata.type = UCDB_STMTBIN;
    coverdata.flags = UCDB_IS_32BIT;    /* data type flag */
    coverdata.data.int32 = count;       /* must be set for 32 bit flag */
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0;                  /* fake token # */
    coverindex = ucdb_CreateNextCover(db,parent,
                                   NULL, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
    /* SINDEX attribute is used internally by Questa: */
    attrvalue.type = UCDB_ATTR_INT;
    attrvalue.u.ivalue = 1;

ucdb_AttrAdd(db,parent,coverindex,UCDBKEY_STATEMENT_INDEX,&attrvalue);
}
```

Like any object to be created in the design or test bench or testplan hierarchy, this requires a parent. The third argument to ucdb_CreateNextCover() is the name of the object. Statements do not have a name as created by Questa. (You can provide one, but Questa will ignore it.)

The &coverdata argument is a pointer to the ucdbCoverDataT structure. This structure contains all the data associated with the bin except for the name and source information. The “data” field is a union containing the coverage count: int32 for 32-bit platforms or int64 for 64-bit platforms. In this example, it is hard-coded to 32-bits, which requires setting both the appropriate field of the union and the corresponding flag. Other data fields are optionally enabled based on the flags field of ucdbCoverDataT. Statements require only the data field (the coverage count).

The SINDEX user-defined attribute is used to determine the ordering of the statement on a line. If the statement is the only one to appear on the line, SINDEX is always 1. The second statement on a line would have value 2, and so on. If this SINDEX attribute is not given, the ItemNo column of the Questa statement coverage details report (vcover report -code s -byfile -details ucdb) will not be correct.

Add Toggle to a UCDB

Toggles have special data characteristics which require they be created with a special API call.

You can find this example in <*install_dir*>*/examples/ucdb/userguide/use-cases/create-ucdb/*

```
void
create_enum_toggle(ucdbT db,ucdbScopeT parent)
{
   ucdbCoverDataT coverdata;
   ucdbScopeT toggle;
   toggle = ucdb_CreateToggle(db,parent,
      "t",                      /* toggle name */
      NULL,                     /* canonical name */
      0,                        /* exclusions flags */
      UCDB_TOGGLE_ENUM,         /* toggle type */
      UCDB_TOGGLE_INTERNAL);  /* toggle "direction" */
   coverdata.type = UCDB_TOGGLEBIN;
   coverdata.flags = UCDB_IS_32BIT;     /* data type flag */
   coverdata.data.int32 = 0;            /* must be set for 32 bit flag */
   ucdb_CreateNextCover(db,toggle,
      "a",                              /* enum name */
      &coverdata,
      NULL);                            /* source data */
   coverdata.data.int32 = 1;            /* must be set for 32 bit flag */
   ucdb_CreateNextCover(db,toggle,
      "b",                              /* enum name */
      &coverdata,
      NULL);                            /* source data */
}
```

This example corresponds to a source toggle declared as follows in SystemVerilog:

```
enum { a, b } t;
```

The toggle has only name and no source information (so NULL values are passed to ucdb_CreateNextCover()). Source info could be added later using ucdb_SetScopeSourceInfo() on toggle scopes.

The canonical name is used for wire (net) toggles, as described in the section "Toggle Coverage" on page 27. The exclusions flags may apply to the toggle, so those can be given, too.

The toggle type and directionality (input, output, inout, or internal) are given. Directionality really only applies to net toggles, but is set to internal for others.

Recall that an enum toggle has bins whose names correspond to the enum values in the source language. If creating bins for other types of toggles, use the appropriate UCDBBIN_TOGGLE_ #define value as declared in *ucdb.h*.

Add Covergroup to a UCDB

The covergroup is created in various stages. The covergroup for the "create-ucdb" example looks like this:

```
   enum { a, b } t;
   covergroup cg;
      coverpoint t;
   endgroup
```

This requires creating a hierarchy as follows:

1. cg
   a. t
      i. a
      ii. b

The top level code is shown in the following example.

C Example (“create-ucdb”):

```
cvg = create_covergroup(db,instance,"cg",filehandle,3);
cvp = create_coverpoint(db,cvg,"t",filehandle,4);
create_coverpoint_bin(db,cvp,"auto[a]",filehandle,4,1,0,"a");
create_coverpoint_bin(db,cvp,"auto[b]",filehandle,4,1,1,"b");
```

The hierarchy is implied by the use of the parent pointers, which is the second argument to each of these functions. The parent of “cg” is the instance whose scope handle is “instance”; this is loaded into the “cvg” handle. The “cvg” handle is used as the parent to create the “cvp” handle for the coverpoint named “t”. The “cvp” handle is then used as the parent of the bins.

The creation of the covergroup is this example:

C Example (“create-ucdb”):

```
ucdbScopeT
create_covergroup(ucdbT db,
                  ucdbScopeT parent,
                  const char* name,
                  ucdbFileHandleT filehandle,
                  int line)
{
    ucdbScopeT cvg;
    ucdbSourceInfoT srcinfo;
    ucdbAttrValueT attrvalue;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0;                  /* fake token # */
    cvg = ucdb_CreateScope(db,parent,name,
                           &srcinfo,
                           1,           /* from type_option.weight */
                           UCDB_VLOG,   /* source language type */
                           UCDB_COVERGROUP,
                           0);          /* flags */
    /* Hardcoding attribute values to defaults for type_options: */
    attrvalue.type = UCDB_ATTR_INT;
    attrvalue.u.ivalue = 100;
    ucdb_AttrAdd(db,cvg,-1,UCDBKEY_GOAL,&attrvalue);
    attrvalue.u.ivalue = 0;
    ucdb_AttrAdd(db,cvg,-1,UCDBKEY_STROBE,&attrvalue);
    attrvalue.type = UCDB_ATTR_STRING;
    attrvalue.u.svalue = "";
    ucdb_AttrAdd(db,cvg,-1,UCDBKEY_COMMENT,&attrvalue);
    return cvg;
}
```

The scope type is UCDB_COVERGROUP and the source type is UCDB_VLOG. The source type could also be UCDB_SV, but that is not how Questa creates it.

The attributes must have full report capability for the covergroup. Because this covergroup has option.per_instance the default of 0, the example creates type_option values only. type_option.weight is provided directly as an argument to ucdb_CreateScope(). The option.per_instance influences the topology of the covergroup tree itself; if there are no covergroup objects with option.per_instance==1, then there will be no UCDB_COVERINSTANCE scopes in the covergroup subtree.

Following is the creation of the coverpoint.

C Example (“create-ucdb”):

```
ucdbScopeT
create_coverpoint(ucdbT db,
                  ucdbScopeT parent,
                  const char* name,
                  ucdbFileHandleT filehandle,
                  int line)
{
    ucdbScopeT cvp;
    ucdbSourceInfoT srcinfo;
    ucdbAttrValueT attrvalue;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0;                  /* fake token # */
    cvp = ucdb_CreateScope(db,parent,name,
                           &srcinfo,
                           1,           /* from type_option.weight */
                           UCDB_VLOG,   /* source language type */
                           UCDB_COVERPOINT,
                           0);          /* flags */
    /* Hardcoding attribute values to defaults for type_options: */
    attrvalue.type = UCDB_ATTR_INT;
    attrvalue.u.ivalue = 100;
    ucdb_AttrAdd(db,cvp,-1,UCDBKEY_GOAL,&attrvalue);
    attrvalue.u.ivalue = 1;
    ucdb_AttrAdd(db,cvp,-1,UCDBKEY_ATLEAST,&attrvalue);
    attrvalue.type = UCDB_ATTR_STRING;
    attrvalue.u.svalue = "";
    ucdb_AttrAdd(db,cvp,-1,UCDBKEY_COMMENT,&attrvalue);
    return cvp;
}
```

This is very similar to the covergroup creation, except for the scope type, the parent (which is the previously created covergroup), and the options (including the weight given to ucdbCreateScope()) which derive from the default values for the type_option structure in the coverpoint scope.

The bins are created as children of the coverpoint.

C Example (“create-ucdb”):

```
void
create_coverpoint_bin(ucdbT db,
                      ucdbScopeT parent,
                      const char* name,
                      ucdbFileHandleT filehandle,
                      int line,
                      int at_least,
                      int count,
                      const char* binrhs)   /* right-hand-side value */
{
    ucdbSourceInfoT srcinfo;
    ucdbCoverDataT coverdata;
    ucdbAttrValueT attrvalue;
    int coverindex;
    coverdata.type = UCDB_CVGBIN;
    coverdata.flags = UCDB_IS_32BIT | UCDB_HAS_GOAL | UCDB_HAS_WEIGHT;
    coverdata.goal = at_least;
    coverdata.weight = 1;
    coverdata.data.int32 = count;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0;                  /* fake token # */
    coverindex = ucdb_CreateNextCover(db,parent,name,
                                      &coverdata,&srcinfo);
    attrvalue.type = UCDB_ATTR_STRING;
    attrvalue.u.svalue = binrhs;
    ucdb_AttrAdd(db,parent,coverindex,UCDBKEY_BINRHSVALUE,&attrvalue);
}
```

The following data are unique for the create-ucdb example:

- UCDB_HAS_GOAL indicates that the goal field of ucdbCoverDataT should be used. This corresponds to the at_least value for the coverpoint: the threshold at which the bin is considered to be 100% covered.
- UCDB_HAS_WEIGHT indicates that the weight field of the ucdbCoverDataT is valid. This weight is identical to the weight for the parent coverpoint, but is also set here in case coverage is computed on a bin basis rather than for the coverpoint as a whole. The field is useful for coveritems with no explicit parent (for example, statement bins.)
- The BINRHSVALUE attribute is one added by Questa that depends on knowledge of how the coverpoint is declared. This should be reverse-engineered from covergroup bin declarations and using ucdbdump. The bin rhs value is the sampled value(s), on the right-hand side of the equal sign (=) in the bin declaration, that potentially cause(s) a bin to increment. In the LRM these are described as associated values or transitions. These values vary depending on whether the bin has a single value or multiple and whether it is a transition bin or not. The bin can be an enum value, it can be another type of integral value, or it transitions among those values.

Currently in Questa, the BINRHSVALUE is accessible only through the UCDB API.

Related Topics

Scopes

Test Data Records

This is an example of creating test data that is nearly identical to that created automatically by Questa for the “create-ucdb” example. The differences are in the date and userid, which cannot be reproduced because those will vary according to who runs the example and when.

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/create-ucdb/*

```
void
create_testdata(ucdbT db,
                const char* ucdbfile)
{
    ucdb_AddTest(db,
        ucdbfile,
        "test",                 /* test name */
        UCDB_TESTSTATUS_OK,     /* test status */
        0.0,                    /* simulation time */
        "ns",                   /* simulation time units */
        0.0,                    /* CPU time */
        "0",                    /* random seed */
        NULL,                   /* test script: not used by Questa */
                                /* simulator arguments: */
       "-coverage -do 'run -all; coverage save test.ucdb; quit' -c top ",
        NULL,                   /* comment */
        0,                      /* compulsory */
        "20070824143300",       /* fake date */
        "userid"                /* fake userid */
        );
}
```

All of the test data attributes (arguments to the create_testdata() function above) correspond to attributes names that can be accessed using the UCDB attribute API. One of the chief uses of the attribute data is to add user-defined attributes that can be added for any reason. In Questa, these attributes will appear in the UCDB Browser or the Test Tracking GUI if the test data record is linked as a directed test in a testplan.

You can create or access any of these test data attributes in Questa with the coverage attribute or vcover attribute commands.

The format of the date is strict, you can create it from a POSIX-compliant C library call, strftime(). You can sort the dates alphabetically.

The “test script” argument to ucdb_AddTest() is not used, though it could be. The simulator arguments are created automatically and can be used to rerun the test. The simulator arguments should be quoted such that the arguments could be passed to a shell for running with the simulator (vsim in this case.)

The comment is typically not used, but it can be set within the tool. This is a general-purpose comment that can be used for anything.

Create a UCDB from Scratch in Memory

This is the top-level code that calls all of the functions.

Note

For more information about functions, see "Addition of New Data to a UCDB" on page 82.

You can find this example in <*install_dir*>*/examples/ucdb/userguide/use-cases/create-ucdb/*.

```
void
example_code(const char* ucdbfile)
{
    ucdbFileHandleT filehandle;
    ucdbScopeT instance, du, cvg, cvp;
    ucdbT db = ucdb_Open(NULL);
    create_testdata(db,ucdbfile);
    filehandle = create_filehandle(db,"test.sv");
    du = create_design_unit(db,"work.top",filehandle,0);
    instance = create_instance(db,"top",NULL,du);
    create_statement(db,instance,filehandle,6,1);
    create_statement(db,instance,filehandle,8,1);
    create_statement(db,instance,filehandle,9,1);
    create_enum_toggle(db,instance);
    cvg = create_covergroup(db,instance,"cg",filehandle,3);
    cvp = create_coverpoint(db,cvg,"t",filehandle,4);
    create_coverpoint_bin(db,cvp,"auto[a]",filehandle,4,1,0,"a");
    create_coverpoint_bin(db,cvp,"auto[b]",filehandle,4,1,1,"b");
    printf("Writing UCDB file '%s'\n", ucdbfile);
    ucdb_Write(db,ucdbfile,NULL,1,-1);
    ucdb_Close(db);
}
```

This reproduces – with a few exceptions described in the header comment of create_ucdb.c – the UCDB created by Questa from this source:

SystemVerilog Example ("create-ucdb"):

```
module top;
   enum { a, b } t;
   covergroup cg;
      coverpoint t;
   endgroup
   cg cv = new;
   initial begin
      t = b;
      cv.sample();
   end
endmodule
```

Note

The call to ucdb_Open() with a NULL argument creates a completely empty UCDB in memory, to which any data can be added.

Because of tool requirements, it is not permissible to create a UCDB without a test data record; the ucdb_Write() will not succeed if there is no test data record.

The final ucdb_Close(db) is not strictly necessary because the memory used by the database handle will be freed when the process finishes, but it is good practice to explicitly free the memory associated with the database handle.

Read-Streaming Mode

Read-streaming mode is a callback-based traversal of a UCDB as laid out on disk. It has the advantage of reducing memory overhead, as the UCDB is never fully loaded into memory.

The read-streaming disk layout on disk is as follows:

- Header with database version and other header information.
- Global UCDB attributes can appear at any time at the top-level, but are ordinarily written as early as possible.
- Test data records.
- Design units are written before instances of them.
- Scopes (design units, instances, or any coverage scope) are written in a nested fashion: meaning that the start of the scope is distinct from the end of the scope. Scopes that start and end within another's start and end are children scopes. This is how the parent-child relationships are recorded. The start of the parent is always written before the children. The termination of the parent scope “pops” the current scope back to its parent.
- Coveritems are written immediately after the parent scope.
- Attributes and tags are written after the initial header for the scope or coveritem.
- Tail with summary data.

The tail is loaded at the same time as the header, which enables ucdb_GetCoverageSummary() to work.

The rules for read-streaming mode are relatively simple. In general, available data follows the order in which data is laid out on disk. The attributes, flags, and so on, are complete with the read object. There is no access to child scopes or coveritems at the time a scope is read. The implementation maintains the following data at all times:

- All ancestors of a given scope or coveritem

- All design units
- All global UCDB attributes and other data global to the UCDB
- All test data records
- The summary data used by ucdb_GetCoverage(), ucdb_GetStatistics(), and various other functions described in the API reference as pertaining to global coverage statistics

However, the inaccessibility of children means that any descendant nodes, or any descendants of ancestors (what you might informally call “cousin nodes” or “uncle nodes”) are not available.

Read-streaming mode maintains a relatively small window into the data, that progresses through the file, with some global data available generally.

There are some other limitations, all of which relate to the fact that children are not available except exactly when they are encountered within the streaming window:

- Because the test plan tree is implemented with tags, there is no way to know when reading a test plan node what are the other nodes sharing the same tag. Test plan trees are essentially unusable in read-streaming mode, although you can also build the associations yourself, you could.
- The functions like ucdb_PathCallBack() that require searching the database cannot work.
- The functions like ucdb_CalcCoverageSummary() that require traversing some subset of the database cannot work.

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/read-streaming/*.

```
ucdbCBReturnT
callback(
    void*           userdata,
    ucdbCBDataT*    cbdata)
{
    ucdbScopeT scope;
    switch (cbdata->reason) {
    case UCDB_REASON_DU:
    case UCDB_REASON_SCOPE:
        scope = (ucdbScopeT)(cbdata->obj);
        printf("%s\n",ucdb_GetScopeHierName(cbdata->db,scope));
        break;
    default: break;
    }
    return UCDB_SCAN_CONTINUE;
}

void
example_code(const char* ucdbfile)
{
   ucdb_OpenReadStream(ucdbfile,callback,NULL);
}
```

The read-streaming mode is based on the same callback type functions as ucdb_CallBack(). This example is the traverse-scopes example, but the example_code function is different. The database handle is only available through the callback. The path to the UCDB file is given to the open call, and this calls the callback for each object in the database.

The *examples/ucdb/ucdbdump* example is a read-streaming mode application that shows how to use the mode.

Write-Streaming Mode

Write-streaming mode is a way of writing a UCDB with optimally low memory overhead.

Caution

Avoid the write-streaming use case unless you are a professional tool developer who is concerned with memory overhead, or you are linked with the Questa kernel (through PLI, VPI, or FLI) and want to contribute your own data in real time to a UCDB being saved with the coverage save command executed from vsim. This is discussed in the section "Using the mti_AddUCDBSaveCB FLI Callback" on page 100

An alternative to using write-streaming mode is to create a UCDB from scratch in memory. For information on how to do this, refer to "Create a UCDB from Scratch in Memory" on page 93.

The write-streaming example shows the create-ucdb example adapted to write-streaming mode. There is also the *examples/ucdb/writestream* example that contains extensive comments on using the mode.

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/create-ucdb/*.

```
void
example_code(const char* ucdbfile)
{
    ucdbFileHandleT filehandle;
    ucdbT db = ucdb_OpenWriteStream(ucdbfile);
    create_testdata(db,ucdbfile);
    filehandle = create_filehandle(db,"test.sv");
    create_design_unit(db,"work.top",filehandle,0);
    create_instance(db,"top","work.top");
    create_statement(db,filehandle,6,1);
    create_statement(db,filehandle,8,1);
    create_statement(db,filehandle,9,1);
    create_enum_toggle(db);
    create_covergroup(db,"cg",filehandle,3);
    create_coverpoint(db,"t",filehandle,4);
    create_coverpoint_bin(db,"auto[a]",filehandle,4,1,0,"a");
    create_coverpoint_bin(db,"auto[b]",filehandle,4,1,1,"b");
    ucdb_WriteStreamScope(db);      /* terminate coverpoint */
    ucdb_WriteStreamScope(db);      /* terminate covergroup */
    ucdb_WriteStreamScope(db);      /* terminate instance */
    printf("Writing UCDB file '%s'\n", ucdbfile);
    ucdb_Close(db);
}
```

The differences required to convert the in-memory creation of data to a write-streaming creation of data are as follows:

- The open call is ucdb_OpenWriteStream(), which gives the name of the output file. The concept of write-streaming is that it writes to the file as it runs. So you have to create objects in the same order as you do for read-streaming mode. The API is designed to emit errors in case functions are used in the wrong order, but this has not yet been exposed to third-party developers for beta testing.
- The parent pointers for all creation API calls must be NULL. This emphasizes that the level of hierarchy for creating the current object relies on the current context. This will be explained more deeply below. Because no parent pointers are used, the functions in the example are all of type void, except for the create_filehandle() routine, because filehandles must be used when needed. In this case, because the filehandle is global, it can be used with any object.
- The ucdb_WriteStream(db) call is used to terminate the creation of the current object. For scopes, this call terminates the creation of the beginning of the scope. The ucdb_WriteStream(db) call creates the scope as a context and writes the name of the scope and other information to the file, so that subsequent objects are created as children of that scope. The API is actually relatively forgiving about the use of ucdb_WriteStream(db). It is really like a “flush” to disk. You can remove ucdb_WriteStream(db) from this example entirely and by placing this line after the include of *ucdb.h*:

  ```
  #define ucdb_WriteStream(db) ;
  ```

This definition works the same because the API will flush the current object before writing the next one if you call any ucdb_Create... API function; it calls ucbd_WriteStream() implicitly. The utility of having the explicit "flush" capability of ucdb_WriteStream() is for cases where you are reusing string storage (as in creating objects from a loop). If you need to set up string storage in advance of calling ucdb_CreateNextCover(), for example, then you must flush the current object before calling ucdb_CreateNextCover(). Because the API is designed for efficiency, it does not always copy string storage; it makes use of the string value when you call ucdb_Writestream(), and after that you may change the value.

- The ucdb_CreateInstanceByName() function must be used to create the instance. This is name-based for the design unit rather than using a ucdbScopeT handle.
- The ucdb_WriteStreamScope(db) call must be used to terminate the scope.
- The ucdb_Close(db) function terminates the write to the file as well as frees the database handle. This function writes the summary information, which has been calculated as you were writing the contents of the file.

In write-streaming mode, the nesting of calls creates the design and test bench hierarchy, which means that ucdb_WriteStreamScope(db) is not optional because it terminates a scope. Write-streaming mode maintains a "current scope." When you create a new scope, it is added under the current scope, then it itself becomes the current scope in turn. When a coveritem is added, it is added to the current scope. When the current scope is terminated, the current scope becomes the parent of that scope (or none if that scope was itself at the top-level.) The three calls to ucdb_WriteStreamScope(db) in the example are commented with the type of the scope they terminate. If you wanted to write another coverpoint to the covergroup in the write-streaming example, create it after the line commented with "// terminate coverpoint" but before the line commented with "// terminate covergroup."

Because write-streaming mode has critical dependencies on order of creation, it is a difficult mode to use. But it is necessary to use when you want the most seamless mode of integration with Questa and when you have code linked into Questa through an interface like VPI.

Chapter 3UCDB in Questa and ModelSim

If you have a model linked with the simulation kernel through PLI, VPI, or FLI, you can use the Questa coverage save command for a transparent integration of coverage data.

There is a facility for installing a callback through FLI, which is the Questa/ModelSim-proprietary simulator interface. Whenever Questa executes coverage save, it calls your callback, whereupon you may use write-streaming mode to contribute your own data to the UCDB being saved. Before contributing your own data to the UCDB, it helps to understand the role the UCDB plays in the Questa architecture.

UCDB in the Tool Architecture 99

Using the mti_AddUCDBSaveCB FLI Callback 100

Questa Compatibility. 103

UCDB in the Tool Architecture

the UCDB does not exist as a memory image in simulation. Coverage data in simulation is intricately linked into the simulation context tree (hierarchical name-based data structure), and is only extracted on demand and written, using a wrapper around the UCDB API write-streaming mode, to disk.

The UCDB only exists in memory in “viewcov” mode, where there is no current facility for linking in third-party C or C++ code. If you want to participate in the UCDB in simulation, you must install the FLI callback and write your data in write-streaming mode.

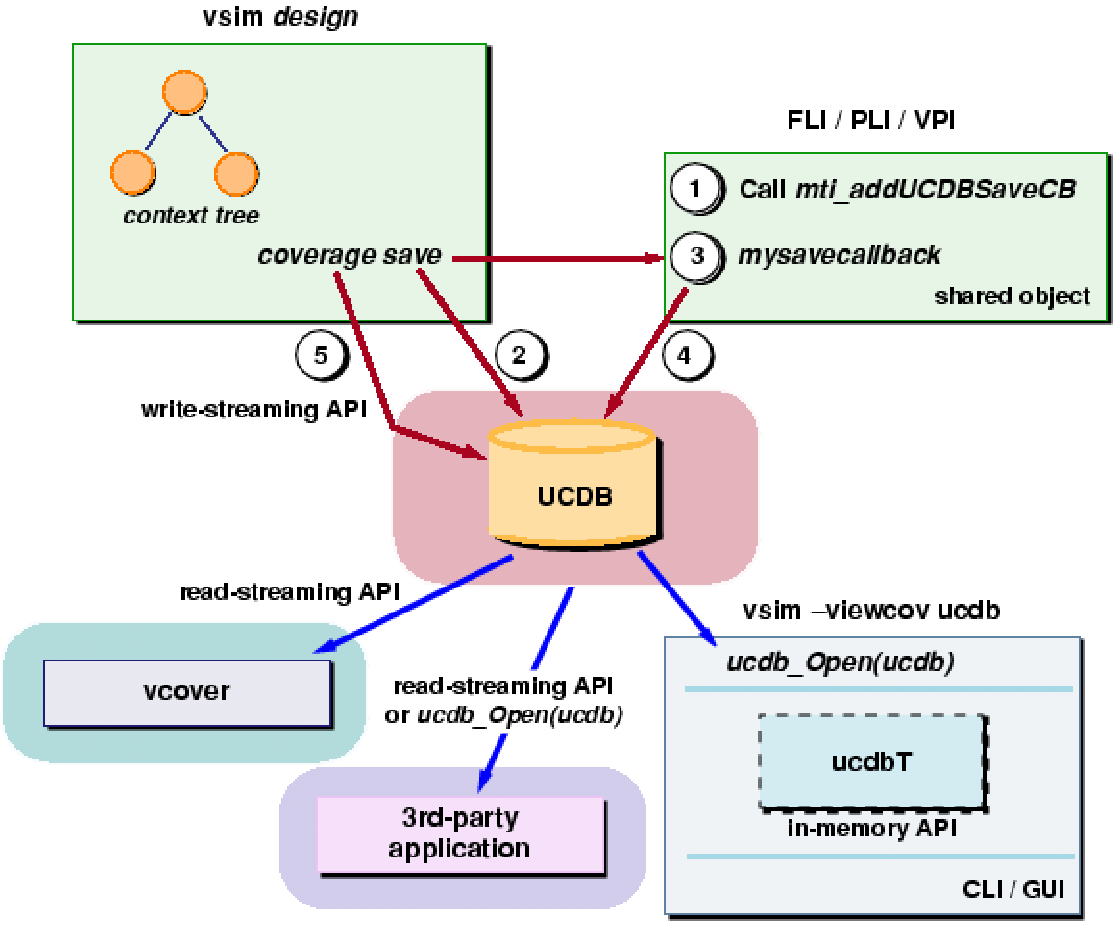
Figure 3-1 illustrates the tool architecture and the FLI callback. In the upper left is vsim in simulation mode, that is, invoked on a design. When coverage save is invoked, the data from the context tree is written to the UCDB using the UCDB write-streaming API.

If a shared object is attached to the simulator, the UCDB save FLI callback operates in this order:

1. The callback (mysavecallback in this case) is installed.
2. Vsim code underlying coverage save initiates the save of the UCDB.
3. In contexts for which the callback is installed, vsim calls the callback function you specified.
4. Your callback makes write-streaming API calls to write data into the same UCDB.
5. The vsim code underlying coverage save continues to save to the UCDB file.

Steps 3 through 5 may be repeated multiple times.

Figure 3-1. Questa and the UCDB Save FLI Callback



The diagram also illustrates the following:

- In general, the only Questa tool where a UCDB image exists in memory (illustrated here at the dashed box around “ucdbT” or a UCDB handle) is when you invoke vsim in “viewcov mode” on a UCDB file. The file is opened in memory, and the CLI and GUI have full in-memory facilities upon which to operate on the data.
- In general, the vcover utility processes all its inputs using the read-streaming API. There are some exceptions to this, but the most commonly used applications, report and merge, are exclusively read-streaming. Merge maintains its output in memory, but its inputs are always read-streaming.
- A third-party application may be either read-streaming or in-memory.

Using the mti_AddUCDBSaveCB FLI Callback

The mti_AddUCDBSaveCB callback is designed to work on only one “region” (or scope: a module instance in the example.) This means that if you have data in multiple scopes, you must install the callback multiple times. However, you only need to install the callback for as many scopes as you have coverage data to contribute.

The mti_AddUCDBSaveCB Foreign Language Interface (FLI) callback, is executed when the UCDB save is, either automatically at the end of simulation or when the CLI is used. The example uses the CLI. (If the Verilog code uses $finish, which returns control to the operating system, the save must be set up in advance.)

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/save-callback/*. This example demonstrates use of the callback to create the 2-bin covergroup from the write-streaming example in the same directory. The save-callback example uses the callback and the interoperation of the VPI and FLI.

```
/*
 *  Register mymodel with simulator
 */
void register_mymodel()
{
    s_vpi_systf_data systf_data;
    systf_data.type = vpiSysFunc;
    systf_data.sysfunctype = vpiSysFuncSized;
    systf_data.tfname = "$mymodel";
    systf_data.calltf = mymodel;
    systf_data.compiletf = mymodel_setup;
    systf_data.sizetf = NULL;
    vpi_register_systf(&systf_data);
}
...
/*
 *  UCDB Save Callback
 */
void
mymodel_ucdb_save(ucdbT db,
                  mtiRegionIdT region,
                  void* unused)
{
    vpi_printf("Saving UCDB data from VPI model ...\n");
    write_ucdb_data(db);
}

/*
 *  Register UCDB Save Callback
 */
int mymodel_setup(char* unused)
{
    vpiHandle systf_handle, scope_handle;
    char* scope_name;
    mtiRegionIdT FLI_scope_handle;

    /* Get name of enclosing scope through VPI */
    systf_handle = vpi_handle(vpiSysTfCall,NULL);
    scope_handle = vpi_handle(vpiScope,systf_handle);
    scope_name = vpi_get_str(vpiFullName,scope_handle);

    /* Convert to FLI region id type */
    FLI_scope_handle = mti_FindRegion(scope_name);
    scope_name = mti_GetRegionFullName(FLI_scope_handle);

    /* Install UCDB save callback */
    vpi_printf("Installing UCDB Save Callback for %s ...\n",scope_name);
    mti_AddUCDBSaveCB(FLI_scope_handle,mymodel_ucdb_save,NULL);
    return 0;
}
```

The callback uses FLI, while the model uses VPI. A FLI scope handle (region ID) must be derived from a VPI handle. The only way to do this is by name, specifically, full name which is a full path to the scope. In the example, “scope_name” is “top.inst” as returned from VPI but it is “/top/inst” as returned from FLI. Fortunately, the two different conventions for regarding the full name are interchangeable, and the FLI scope handle is directly acquired.

The FLI scope handle (region ID) is passed to the callback “mymodel_ucdb_save”, but it is unused in this example. You can also use private data, but no private data is illustrated in this example.

You can find this example in *<install_dir>/examples/ucdb/userguide/use-cases/save-callback/*.

```
void
write_ucdb_data(ucdbT db)
{
    ucdbFileHandleT filehandle;
    filehandle = create_filehandle(db,"test.sv");
    create_covergroup(db,"cg",filehandle,3);
    create_coverpoint(db,"t",filehandle,4);
    create_coverpoint_bin(db,"auto[a]",filehandle,4,1,0,"a");
    create_coverpoint_bin(db,"auto[b]",filehandle,4,1,1,"b");
    ucdb_WriteStreamScope(db);      /* terminate coverpoint */
    ucdb_WriteStreamScope(db);      /* terminate covergroup */
}
```

This example code is part of the top-level code from the write-streaming example. The code and the functions called are exactly the same. The difference is that the enclosing scope (for the module instance “/top/inst” in this case) has been started or initialized by the UCDB save code from Questa and will also be terminated by Questa, too. Any write-streaming mode UCDB API code may be used in the callback. Questa should emit errors for misuse of the API, and those should appear in the transcript. You can not install your own UCDB API error handler with VPI, FLI, or DPI code, because the Questa kernel has already installed its own.

Questa Compatibility

These compatibility commitments are made by Questa and its implementation of the UCDB API.

- Questa release 6.2b is the base release for the UCDB and API.
- The UCDB API will load any UCDB created newer than the 6.2b release.
- The header maintains strict backward compatibility from the 6.2b release onward. Applications compiled against a release newer than 6.2b will continue to compile and continue to link.
- From Questa 6.3 onward, the UCDB API is forward link compatible. An application can be compiled with an earlier version of the *ucdb.h* and still link with a later version of the library archive or shared object (or DLL on Windows.) This behavior enables some flexibility in dynamically linking to the UCDB API by a third-party tool whose releases may not be predictably synchronized with the Questa releases.
- Questa does not commit to backward compatibility with respect to data models. Some applications may require changes when significant portions of the data model change.

Complete backward compatibility of the API is not the same as complete backward compatibility of the data model. API compatibility means that an earlier application will continue to compile and link. However, if it makes critical assumptions about the data model that are no longer met, the application will not continue to work as expected.

This enables some flexibility to change data models in the tool. It also shows that it is difficult to know what assumptions an application might make. Some applications may be sufficiently general that they always continue to work; others may not. Until data models are standardized and can be verified to conform to the standard, the UCDB API developer should be prepared to make occasional changes to an application when data models change.

Chapter 4
UCDB API Functions

This section defines the UCDB API functions by function group.

Source Files 105
Error Handler 111
Tests 113
Databases and Database Files 121
User-Specified Attributes 127
Scopes 130
Coverage and Statistics Summaries 157
Coveritems 167
Toggles 177
Groups 180
Tags 183
Formal Data 187
Test Traceability 197

Source Files

Every UCDB object can have a source filename stored with it. Different applications have different requirements for how these are stored. Consequently, the UCDB contains an object called a “filehandle,” which provides a way of storing indirect references to filenames.

Simple Use Models

You can create objects with NULL for the ucdbSourceInfoT argument, for example:

```
mycover = ucdb_CreateNextCover(db,parent,name,&coverdata,NULL);
```

Alternatively, you can create a file and store it with the object.

```
ucdbSourceInfoT sourceinfo;
status = ucdb_CreateSrcFileHandleByName(
            db,
            &source_info.filehandle,
            NULL,
            filename,
            fileworkdir);
source_info.line = myline;
source_info.token = mytoken;
(void) ucdb_CreateNextCover(db,parent,name,&coverdata,&sourceinfo);
```

This method creates a single global look-up table for filenames within the UCDB. Filenames are stored efficiently for each object within the UCDB, and each unique filename string is stored only once. The way you store filenames does not affect access; you can always access the filename, for example:

```
ucdbSourceInfoT sourceinfo;
ucdb_GetCoverData(db,parent,i,&name,&coverdata,&sourceinfo);
if (sourceinfo.filehandle != NULL) {
   printf("filename is %s\n",
      ucdb_GetFileName(db,&sourceinfo.filehandle));
```

Scope Handle	`typedef void* ucdbScopeT;`	Scope handle
Object Handle	`typedef void* ucdbObjT;`	Either ucdbScopeT or ucdbTestT
File Handle	`typedef void* ucdbFileHandleT;`	Filehandle

Source Information Type	`typedef struct {`
	`ucdbFileHandleT filehandle;`
	`int line;`
	`int token;`
	`} ucdbSourceInfoT;`
		Source information for database objects

ucdb_CreateSrcFileHandleByName

```
int ucdb_CreateSrcFileHandleByName(
               ucdbT                db,
               ucdbFileHandleT*     filehandle,
               ucdbScopeT           scope,
               const char*          filename,
               const char*          fileworkdir)
```

db	Database
filehandle	Filehandle returned
scope	File table scope, or NULL for the global table
filename	Absolute or relative filename to look up in the table
fileworkdir	Work directory for the file when filename is a path relative to fileworkdir. Ignored if filename is an absolute path

Creates a filehandle for the specified file, from the file table associated with the given scope. If the filename is not found, it is added to the file table for the given scope. Returns 0 if successful, or -1 if error and ucdb_IsValidFileHandle(returnvalue) == 0 if error.

ucdb_CreateFileHandleByNum

```
int ucdb_CreateFileHandleByNum(
               ucdbT              db,
               ucdbFileHandleT*   filehandle,
               ucdbScopeT         scope,
               int                filenum)
```

db	Database
filehandle	Filehandle returned
scope	File table scope, or NULL for the global table
filenum	Offset of the file in the file table

Creates a filehandle for the specified offset into the file table of the specified scope. Returns 0 if successful, or -1 if error (for example, if filenum is out of bounds or no file table exists for the scope) and ucdb_IsValidFileHandle(returnvalue) == 0 if error.

ucdb_CloneFileHandle

```
int ucdb_CloneFileHandle(
               ucdbT              db,
               ucdbFileHandleT*   filehandle,
               ucdbFileHandleT*   origfilehandle,
               int                filenum);
```

db	Database
filehandle	Filehandle returned
origfilehandle	Filehandle to clone
filenum	Offset to the new file in the file table

Creates a filehandle cloned from the specified filehandle, at the specified offset, in the same table as the cloned file. The file number (offset) must be in bounds for the file table. Returns 0 if successful, or -1 if error.

ucdb_CreateNullFileHandle

```
int ucdb_CreateNullFileHandle(
               ucdbFileHandleT*   filehandle);
```

filehandle	Null filehandle returned

Creates a new filehandle. Returns 0 if successful, or -1 if error and ucdb_IsValidFileHandle(filehandle) == 0.

ucdb_IsValidFileHandle

```
int ucdb_IsValidFileHandle(
    ucdbT               db,
```

db	Database
filehandle	Filehandle to test

Checks whether or not the specified filehandle returned by a UCDB function is valid. Use this function for non-callback-based error-checking. Returns 1 if filehandle is valid, or 0 if invalid.

ucdb_GetFileName

```
const char* ucdb_GetFileName(
    ucdbT               db,
```

db	Database
filehandle	Filehandle

Returns the filename of the file specified by filehandle, or NULL if error. This function tries to reconstruct a valid filepath from the filehandle, the directory stored with it, and the UCDB. In the following algorithm, *filename* and *fileworkdir* refer to the corresponding arguments of ucdb_CreateSrcFileHandleByName() or ucdb_SrcFileTableAppend():

```
if (filename is an absolute path) return the path name
:else (filename is a relative path)
:  if (filename exists at the relative path)
:     return filename
:  else if (filename exists relative to fileworkdir)
:     return workdir/fileworkdir
:
:else if (filename exists relative to the the value of the environment
:        variable MGC_WD)
:     return $MGC_WD/filename
:
:else if (filename exists relative to the directory from which the
:        UCDB file was opened -- that is, the directory extracted from the
:        file given to ucdb_Open() or equivalent)
:     return that dir/filename
:  else if (filename exists relative to the directory extracted from the
:        ORIGFILENAME attribute of the first test record -- i.e.,
:        representing the file into which the UCDB was originally saved)
:     return that dir/filename
   else return filename.
```

If the filename was created as an absolute path, it must be correct. Otherwise only the last case indicates that the file was not found, and the original filename is returned.

ucdb_GetFileNum

```
int ucdb_GetFileNum(
    ucdbT               db,
```

db	Database
filehandle	Filehandle

Returns the file number of the file specified by filehandle, or -1 if error.

ucdb_GetFileTableScope

```
ucdbScopeT ucdb_GetFileTableScope(
    ucdbT               db,
```

db	Database
filehandle	Filehandle

Returns the scope of the table of the file specified by filehandle. Returns NULL if the specified filehandle is not valid or if the table is the global file table. Also calls an error handler (if installed) when the filehandle is not valid.

ucdb_SrcFileTableAppend

```
int ucdb_SrcFileTableAppend(
    ucdbT               db,
    ucdbFileHandleT*    filehandle,
    ucdbScopeT          scope,
    const char*         filename,
```

db	Database
filehandle	Filehandle returned
scope	File table scope, or NULL for the global table
filename	Absolute or relative filename to look up in the table
fileworkdir	This is the work directory for the file when filename is a path relative to fileworkdir. Ignored if filename is an absolute path.

Creates a filehandle for the specified file, from the file table associated with the given scope. The filename is added to the file table for the given scope, so the filename is assumed to be unique. To check for duplicate filenames, use ucdb_CreateSrcFileHandleByName. Returns 0 if successful, or -1 if error and ucdb_IsValidFileHandle(returnvalue) == 0 if error.

ucdb_FileTableSize

```
int ucdb_FileTableSize(
    ucdbT                   db,
```

db	Database
scope	File table scope, or NULL for the global table

Returns the number of files in the file table associated with the specified scope, or -1 if error.

ucdb_FileTableName

```
const char* ucdb_FileTableName(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	File table scope, or NULL for the global table
index	File table index of the file

Returns the name of the file with the specified index in the file table for the specified scope, or NULL if error.

ucdb_FileTableUpdateName

```
int ucdb_FileTableUpdateName(
    ucdbT                   db,
    ucdbScopeT              scope,
    int                     index,
    const char*             newname,
```

db	Database
scope	File table scope, or NULL for the global table
index	File table index of the file
newname	New filename
newworkdir	New work directory

Changes the filename and work directory for the specified scope’s file table (or the global file table for a NULL scope). Returns 0 if successful, or -1 if error.

ucdb_FileTableRemove

```
int ucdb_FileTableRemove(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	File table scope, or NULL for the global table
filename	File to remove from the table, or NULL for the whole table

No effect in streaming modes. Removes the specified file from the file table for the specified scope (or the entire table if filename is NULL). Returns 0 if successful, or -1 if error.

ucdb_FileInfoToString

```
const char* ucdb_FileInfoToString(
    ucdbT              db,
```

db	Database
file_info	Source file information handle

Returns a string representation of the filehandle in the specified ucdbSourceInfoT item, or NULL if error. This is equivalent to calling:

```
ucdb_GetFileName(db, &source_info->filehandle)
```

The returned string only remains valid until the next call of this routine. You must copy the returned string before the next call to this function.

Error Handler

The most convenient error-handling mode is to use ucdb_RegisterErrorHandler() before any UCDB calls. The user's error callback, a function pointer of type ucdb_ErrorHandler, is called for any error produced by the system.

Alternatively, function return values can be checked. In general, functions that return a handle return NULL (or invalid handle) on error (they return the handle otherwise). Functions that return an int return non-zero on error (0 otherwise).

Message Severity Type	`typedef ucisMsgSeverityT ucdbMsgSeverityT;`
	`#define UCDB_MSG_INFO    UCIS_MSG_INFO`
	`#define UCDB_MSG_WARNING UCIS_MSG_WARNING`
	`#define UCDB_MSG_ERROR   UCIS_MSG_ERROR`

Error Type	`typedef ucisErrorT ucdbErrorT;`
	`typedef ucis_ErrorHandler ucdb_ErrorHandler;`

Error Handler	`typedef ucis_ErrorHandler ucdb_ErrorHandler;`

ucdb_RegisterErrorHandler

```
void ucdb_RegisterErrorHandler(
    ucdb_ErrorHandler errHandle,
    void*             userdata);
```

errHandle	Error handler handle
userdata	User-specified data for the error handler

Registers the specified error handler that is called whenever an API error occurs.

ucdb_IsModified

```
int ucdb_IsModified(
```

db	Database

Returns 1 if the database was modified after it was loaded into memory, or 0 if error.

ucdb_ModifiedSinceSim

```
int ucdb_ModifiedSinceSim(
```

db	Database

Returns 1 if the database was modified after it was saved from the simulation, or 0 if error. For merged databases, if all the input databases are unmodified, the merged output is unmodified. Otherwise if any file is modified, the output database is modified.

ucdb_SuppressModified

```
int ucdb_SuppressModified(
    ucdbT             db
    int               yes);
```

db	Database
yes	Argument; 0 or 1

If yes is 1, additional changes to the specified database do not modify the database. If yes is 0, changes to the specified database do modify the database. The ucdb_SuppressModified() function suppresses both the in-memory-modified flag and the modified-since-simulation flag, so both the functions ucdb_IsModified() and ucdb_ModifiedSinceSim() return 0 if a change is made while the modify flags are suppressed.

Tests

If a UC database was created as a result of a single test run, the database has a single test data record associated with it. If it was created as a result of a test merge operation, the UC database should have multiple sets of test data. The functions defined in this section can be used to create sets of test data. Each test data record should be associated with the name of the UC database file in which the database was first stored.

For efficiency, history nodes (ucdbHistoryNodeT) and associated functions use different test records for different situations (like merging) rather than creating the same or similar test record for each database operation. Test data record nodes (ucdbTestStatusT) are a subset of history nodes.

Test Type

```
typedef ucdbHistoryNodeT ucdbTestT;
```

Test Status Type

```
typedef enum {
    UCDB_TESTSTATUS_OK,
    UCDB_TESTSTATUS_WARNING,      /* test warning ($warning called)  */
    UCDB_TESTSTATUS_ERROR,        /* test error ($error called)      */
    UCDB_TESTSTATUS_FATAL,        /* fatal test error ($fatal called)*/
    UCDB_TESTSTATUS_MISSING,      /* test not run yet                */
    UCDB_TESTSTATUS_MERGE_ERROR   /* testdata record was merged with
                                     inconsistent data values       */
} ucdbTestStatusT;
```

History Node Types

```
typedef void* ucdbHistoryNodeT;
```

History Node Kind Types

```
typedef enum {
    UCDB_HISTORYNODE_NONE,          /* no node or error             */
    UCDB_HISTORYNODE_MERGE,         /* interior merge node          */
    UCDB_HISTORYNODE_TEST,          /* test leaf node               */
    UCDB_HISTORYNODE_TESTPLAN,      /* testplan leaf node          */
} ucdbHistoryNodeKindEnumT;
```

ucdb_AddTest

```
ucdbTestT ucdb_AddTest(
    ucdbT           db,
    const char*     filename,        /* ORIGFILENAME    */
    const char*     testname,        /* TESTNAME        */
    ucdbTestStatusT test_status,     /* TESTSTATUS      */
    double          simtime,         /* SIMTIME         */
    const char*     simtime_units,   /* TIMEUNIT        */
    double          realtime,        /* CPUTIME         */
    const char*     seed,            /* SEED            */
    const char*     command,         /* TESTCMD         */
    const char*     simargs,         /* VSIMARGS        */
    const char*     comment,         /* TESTCOMMENT     */
    int             compulsory,      /* COMPULSORY      */
    const char*     date,            /* DATE            */
```

db	Database to hold the test
filename	Name of UCDB file to which the database was saved
testname	Test name. Must be unique for each test run
test_status	Test status
simtime	Simulation run time of test (in simtime_units)
simtime_units	Simulation time units
realtime	CPU run time of test
seed	Randomization seed used for the test
command	Test script arguments
simargs	Simulator arguments
comment	User-specified comment
compulsory	1 if a required test, or 0 if not
date	Time of start of simulation, specified as a string. Output of strftime with format "%Y%m%d%H%M%S", for example `4:00:30 PM January 5, 2008` is coded as “20080105160030”
userid	ID of the user who created the file

Adds the specified test data to the database. Used to capture a single set of data from a test’s coverage results saved to a UCDB from simulation. The filename must be the name of the file that later will be saved. The filename is given explicitly to aid in copying test data records. Returns a new test handle, or NULL if error.

ucdb_AddPotentialTest

```
ucdbTestT ucdb_AddPotentialTest(
    ucdbT           db,
```

db	Database to hold the test
testname	Test name. Must be unique for each test run

Adds a test data record with the specified test name and test_status of UCDB_TESTSTATUS_MISSING. All other fields have invalid values. Used to tag a test data record for tests not yet run. Returns a new test handle, or NULL if error.

ucdb_GetTestData

```
int ucdb_GetTestData(
    ucdbT            db,
    ucdbTestT        test,
    const char**     filename,       /* ORIGFILENAME   */
    const char**     testname,       /* TESTNAME       */
    ucdbTestStatusT* test_status,    /* TESTSTATUS     */
    double*          simtime,        /* SIMTIME        */
    const char**     simtime_units,  /* TIMEUNIT       */
    double*          cputime,        /* CPUTIME        */
    const char**     seed,           /* SEED           */
    const char**     command,        /* TESTCMD        */
    const char**     simargs,        /* VSIMARGS       */
    const char**     comment,        /* TESTCOMMENT    */
    int*             compulsory,     /* COMPULSORY     */
    const char**     date,           /* DATE           */
```

db	Database
test	Test
filename	Name of UCDB file first associated with the test
testname	Test name
test_status	Test status
simtime	Simulation run time of test (in simtime_units)
simtime_units	Simulation time units
realtime	CPU run time of test
seed	Randomization seed used for the test
command	Test script arguments
simargs	Simulator arguments
comment	User-specified comment
compulsory	1 if a required test, or 0 if not

date	Time of start of simulation, specified as a string. Output of strftime with format "%Y%m%d%H%M%S", for example: `4:00:30 PM January 5, 2008` is coded as “20080105160030”
userid	ID of the user who created the file

Gets the data for the specified test in the specified database. Allocated values (strings, date, and attributes) must be copied if you want them to persist.Returns 0 if successful, or non-zero if error.

ucdb_GetTestName

```
const char* ucdb_GetTestName(
    ucdbT            db,
```

db	Database
test	Test

Returns the test name for the specified test handle from the specified opened database, or NULL if error.

ucdb_NextTest

```
ucdbTestT ucdb_NextTest(
    ucdbT            db,
```

db	Database
test	Test or NULL for first test handle

Returns the next (or first) test handle from the specified opened database, or NULL if error.

ucdb_CloneTest

```
ucdbTestT ucdb_CloneTest(
    ucdbT            targetdb,
    ucdbTestT        test,
```

targetdb	Target database for the cloned test
test	Source test
cloneflags	UCDB_CLONE_ATTRS (to clone attributes) or 0 (to omit attributes)

No effect if targetdb is in streaming mode. Creates an exact copy of the specified test record. Returns handle to the cloned test, or NULL if error.

ucdb_RemoveTest

```
int ucdb_RemoveTest(
    ucdbT           db,
```

db	Database
test	Test

No effect if db is in streaming mode. Removes the specified test from the database. Returns 0 if successful, or -1 if error.

ucdb_NumTests

```
int ucdb_NumTests(
```

db	Database

Reliable with in-memory mode but only works in streaming mode after all test records are read or written. Returns the number of tests associated with the specified database, or -1 if error (for example, if the value cannot be calculated yet in streaming mode).

ucdb_GetTestIndex

```
ucdb_GetTestIndex(
     ucdbT         db,
     ucdbTestT     test
```

db	Database
test	Test

Return the index value for the test record. Return -1 if an error occurs.

ucdb_GetIthTest

```
ucdb_GetIthTest(
     ucdbT         db,
     int     index
```

db	Database
index	Integer

Return the test record having the indicated index value. Return NULL if an error occurs.

ucdb_CreateHistoryNode

```
ucdbHistoryNodeT ucdb_CreateHistoryNode(
    ucdbT                       db,
    char*                       path,
```

db	Database
path	Testplan path. Must be a valid pathname (cannot be NULL). Set to merge filepath if kind is UCDB_HISTORYNODE_-MERGE, otherwise, set to filepath.
kind	History node kind

Creates a history node of the specified kind in the specified database. History node has default values of path for FILENAME and the current execution directory for RUNCWD. Returns handle to the created history node, or NULL if error or if node already exists. Returned node is owned by the routine and should not be freed by the caller.

ucdb_AddHistoryNodeChild

```
int ucdb_AddHistoryNodeChild(
    ucdbT                       db,
    ucdbHistoryNodeT            parent,
```

db	Database
parent	Parent history node
child	Child history node

Sets the specified node to be a child node of the specified parent node. Each history node appears exactly once in the history trees. In particular, every child can have at most one parent; after ucdb_AddHistoryNodeChild assigns a parent to a child, the child cannot be reassigned to a different parent; and a child node cannot be (directly or indirectly) reassigned to its own parent. Returns non-zero if successful, or 0 if error.

ucdb_NextHistoryNode

```
ucdbHistoryNodeT ucdb_NextHistoryNode(
    ucdbT                       db,
    ucdbHistoryNodeT            historynode,
```

db	Database
historynode	History node or NULL
kind	History node kind

Returns the next history node of the same kind as the specified history node, or if historynode is NULL, returns the first history node of the specified kind. Returns NULL if error or if node does not exist. History node "order" is vendor specific. Returned node is owned by the routine and should not be freed by the caller.

ucdb_HistoryRoot

```
ucdbHistoryNodeT ucdb_HistoryRoot(
```

db	Database

Returns the unique history node that has no parent, or NULL if error or if multiple roots exist. Returned node is owned by the routine and should not be freed by the caller. This routine assumes that only one history node is defined.

ucdb_NextHistoryRoot

```
ucdbHistoryNodeT ucdb_NextHistoryRoot(
    ucdbT                        db,
    ucdbHistoryNodeT             historynode,
```

db	Database
historynode	History node or NULL
kind	History node kind

Returns the next orphan history node of the same kind as the specified history node, or if historynode is NULL, returns the first orphan history node of the specified kind. Returns NULL if node does not exist. History node order is vendor specific. The returned node is owned by the routine and should not be freed by the caller. This routine assumes multiple history roots are possible (that is, a collection of subtree orphans).

ucdb_NextHistoryLookup

```
ucdbHistoryNodeT ucdb_NextHistoryLookup(
    ucdbT                        db,
    ucdbHistoryNodeT             historynode,
    const char*                  attributekey,
    const char*                  attributevalue,
```

db	Database
historynode	History node or NULL
attributekey	UCDB_ATTR_STRING attribute key
attributevalue	Attribute value
kind	History node kind

Returns the next history node of the same kind as the specified history node that has an attribute matching the specified key/value pair, or if historynode is NULL, returns the first history node of the specified kind that has an attribute matching the specified key/value pair. Returns NULL if error or if node does not exist. History node “order” is vendor specific. Returned node is owned by the routine and should not be freed by the caller.

ucdb_GetHistoryNodeParent

```
ucdbHistoryNodeT ucdb_GetHistoryNodeParent(
    ucdbT                     db,
```

db	Database
child	History node

Returns the parent of the specified history node, or NULL if error or if specified node is a root node. Returned node is owned by the routine and should not be freed by the caller.

ucdb_GetNextHistoryNodeChild

```
ucdbHistoryNodeT ucdb_GetNextHistoryNodeChild(
    ucdbT                     db,
    ucdbHistoryNodeT          parent,
```

db	Database
parent	Parent history node
child	Child history node or NULL

Returns the next history node after the specified child history node, or if child is NULL, returns the first history node of the specified parent history node. Returns NULL if error or if next node does not exist. History node “order” is vendor specific. Returned node is owned by the routine and should not be freed by the caller.

ucdb_CloneHistoryNode

```
ucdbHistoryNodeT ucdb_CloneHistoryNode(
    ucdbT                     targetdb,
    ucdbT                     sourcedb,
```

targetdb	Target database for the copied node
sourcedb	Source database containing the node to copy
historynode	History node to copy

Creates an exact copy (including attributes) of the specified history node. Returns the history node for the copy, or NULL if error or if the target history node exists.

ucdb_GetHistoryKind

```
    ucdbScopeTypeT ucdb_GetHistoryKind(
        ucdbT                   db,
```

db	Database
object	Object

Polymorphic function (aliased to ucdb_GetObjType) for acquiring an object type. Returns UCDB_HISTORYNODE_TEST (object is a test data record), UCDB_HISTORYNODE_TESTPLAN (object is a testplan record), UCDB_HISTORYNODE_MERGE (object is a merge record), scope type ucdbScopeTypeT (object is none of the other types), or UCDB_SCOPE_ERROR if error. This function can return a value with multiple bits set (for history data objects). Return value *must not be* used as a mask.

ucdb_CalculateHistorySignature

```
    char* ucdb_CalculateHistorySignature(
        ucdbT                   db,
```

db	Database
file	File

Returns a history signature of the specified file, or NULL if error. The returned string is owned by the routine and must not be freed by the caller. If a file's contents remain unmodified, recalculating the file's history signature produces the same results. Conversely, when the file is modified, the resulting signature will also be changed. Use this mechanism to check whether or not a file has become corrupted.

Databases and Database Files

A UCDB database exists in two forms: an in-memory image accessible with a database handle, and a persistent form on the file system. There are read-streaming and write-streaming modes that minimize the memory usage in the current process. These streaming modes keep only a small window of data in memory; and after you have moved onward in reading or writing, you cannot revisit earlier parts of the database. Random access is not possible.

You use the functions defined in this section to run the following operations:

- Opening a file and creating an in-memory image.

  Reading from a persistent database and creating an in-memory image are combined in the same function: ucdb_Open(), which always creates a valid database handle. If a filename is given to ucdb_Open(), the in-memory image is populated from the persistent database in the named file.

Some parts of the data model can be accessed without fully populating the in-memory data image, only if no other calls have been made since ucdb_Open() that require accessing the in-memory image. In particular, the following data can be accessed in constant time regardless of the size of the UCDB:

- ucdb_CalcCoverageSummary (scope==NULL and test_mask==NULL)
- ucdb_GetCoverage
- ucdb_GetStatistics
- ucdb_GetMemoryStats

- Writing to a file from an in-memory image.

  This operation can be performed at any time with the ucdb_Write() function. This function transfers all of (or a subset of) the in-memory image to the named persistent database file, overwriting the file if it previously existed.

- Deleting the in-memory image.

  This operation is done with the ucdb_Close() function. After this call, the database handle is no longer valid.

- Using write-streaming mode.

  To create a UCDB with minimal memory overhead, use ucdb_OpenWriteStream() to create a UCDB handle whose use is restricted. In particular, objects must be created in the following specific order:

  a. Create UCDB attributes. Creating UCDB attributes at the beginning of the file is not enforced so UCDB attributes can be created at the end of the output (which might be necessary for attributes whose values must be computed as a result of traversing the data during write).

  b. Create TestData.

  c. Create scopes. Create DU scopes before corresponding instance scopes. If a scope contains coverage items, create those first. If a scope contains child scopes, create those after coveritems.

     There are other restrictions as well; see comments for individual functions. For example, accessing immediate ancestors is okay, but accessing siblings is not (nor is it okay to access an ancestor’s siblings).

     The function ucdb_WriteStream() must be used in write-streaming mode to finish writing a particular object. The function ucdb_WriteStreamScope() must be used to finish writing a scope and to resume writing the parent scope. In write-streaming mode, the ucdb_Close() function must be used to finish the file being written to and to free any temporary memory used for the database handle.

- Using read-streaming mode

The read-streaming mode operates with callbacks. The persistent database is opened with a ucdb_OpenReadStream() call that passes control to the UCDB system that then initiates callbacks to the given callback function. Each callback function returns a reason that identifies the data valid for the callback and enough information to access the data. Read-streaming mode callback order includes the following characteristics:

a. INITDB is always the first callback.

b. UCDB attributes created first in write-streaming mode are available, as are UCDB attributes created with in-memory mode.

c. All TEST callbacks follow; after the next non-TEST callback there will be no more TEST callbacks.

d. DU callbacks must precede their *first associated instance* SCOPE callbacks, but they do not need to immediately precede the SCOPE callbacks.

e. SCOPE, DU and CVBIN callbacks can occur in any order, except for the DU before first instance rule, although nesting level is implied by the order of callbacks.

f. ENDSCOPE callbacks correspond to SCOPE and DU callbacks and imply a “pop” in the nesting of scopes and design units.

g. ENDDB callbacks can be used to access UCDB attributes written at the end of the file, if created in write-streaming modes.

- Opening UCDB in streaming mode to read data through callbacks without creating an in-memory database.

  Use the ucdb_OpenReadStream() read API to open a UCDB in stream mode with a callback function of type ucdb_CBFuncT along with user data (which can be NULL). The callback function is called for all UCDB objects present in the database, with an object of type ucdbCBDataT with the user data.

Typedefs

Callback Reason Type

```
typedef enum {
    UCDB_REASON_INITDB,      /* Start of the database,
                                apply initial settings           */
    UCDB_REASON_DU,          /* Start of a design unit scope     */
    UCDB_REASON_TEST,        /* Testplan object                  */
    UCDB_REASON_SCOPE,       /* Start of a scope object          */
    UCDB_REASON_CVBIN,       /* Cover item                       */
    UCDB_REASON_ENDSCOPE     /* End of a scope,
                                including design units           */
    UCDB_REASON_ENDDB,       /* End of database (database handle
                                still valid)                     */
    UCDB_REASON_PLANHISTORY, /* Testplan history object */
    UCDB_REASON_MERGEHISTORY /* Merge history object */
} ucdbCBReasonT;
```

Callback Return Type

```
typedef ucisCBReturnT ucdbCBReturnT;
#define UCDB_SCAN_CONTINUE UCIS_SCAN_CONTINUE
#define UCDB_SCAN_STOP     UCIS_SCAN_STOP
#define UCDB_SCAN_PRUNE    UCIS_SCAN_PRUNE
```

Read Callback Data Type

```
typedef struct ucdbCBDataS {
    ucdbCBReasonT reason;       /* Reason for this callback        */
    ucdbT         db;           /* Database handle, to use in APIs */
    ucdbObjT      obj;          /* ucdbScopeT or ucdbTestT         */
    int           coverindex;   /* If UCDB_REASON_CVBIN, index of
                                   coveritem                       */
} ucdbCBDataT;
```

Function Type for Use with ucdb_OpenReadStream()

```
typedef ucdbCBReturnT (*ucdb_CBFuncT) \
   (void* userdata, ucdbCBDataT* cbdata);
```

ucdb_Open

```
ucdbT ucdb_Open(
```

name	File system path.

Creates an in-memory database, optionally populating it from the specified file. Returns a database handle if successful, or NULL if error.

ucdb_OpenReadStream

```
int ucdb_OpenReadStream(
    const char*  name,
    ucdb_CBFuncT cbfunc,
```

name	File system path
cbfunc	User-supplied callback function
userdata	User-supplied function data

Opens a database for streaming read mode from the specified file. Returns 0 if successful, or -1 if error.

ucdb_OpenWriteStream

```
ucdbT ucdb_OpenWriteStream(
```

name	File system path (write permission must exist for the file)

Opens data in write-streaming mode, overwriting the specified file. Returns a restricted database handle if successful, or NULL if error.

ucdb_WriteStream

```
int ucdb_WriteStream(
```

db	Database

Finishes a write of the current object to the persistent database file in write-streaming mode. This operation is like a flush, which completes the write of what was most recently created in write-streaming mode. Multiple ucdb_WriteStream() calls cause no harm because if the current object has already been written, it is not written again. The specified database handle must have been previously opened with ucdb_OpenWriteStream(). Returns 0 if successful, or -1 if error.

ucdb_WriteStreamScope

```
int ucdb_WriteStreamScope(
```

db	Database

Finishes a write of the current scope (similar to the flush operation of ucdb_WriteStream) and pops (that is, terminates the current scope and reverts to its parent) the stream to the parent scope. . Objects created after this belong to the parent scope of the previously ended scope. Unlike ucdb_WriteStream, this function cannot be called benignly multiple times because it always causes a reversion to the parent scope. This write-streaming process resembles the UCDB_REASON_ENDSCOPE callback in read-streaming mode. The specified database handle must have been previously opened with ucdb_OpenWriteStream(). Returns 0 if successful, or -1 if error.

ucdb_Write

```
int ucdb_Write(
    ucdbT        db,
    const char*  file,
    ucdbScopeT   scope,
    int          recurse,
```

db	Database. The database handle "db" cannot have been opened for one of the streaming modes.
file	filename (write permission must exist for the file)
scope	Scope or NULL if all objects
recurse	Non-recursive if 0. If non-zero, recurse from specified scope or ignored if scope==NULL.
covertype	Cover types (see Cover Types) to save or -1 for everything

Copies the entire in-memory database or the specified subset of the in-memory database to a persistent form stored in the specified file, overwriting the specified file. Returns 0 if successful, or -1 if error.

ucdb_Close

```
int ucdb_Close(
```

db	Database

Invalidates the specified database handle and frees all memory associated with the handle, including the in-memory image of the database, if not in one of the streaming modes. If db was opened with ucdb_OpenWriteStream(), this functional call has the side effect of closing the output file. Returns 0 if successful, or non-zero if error.

ucdb_DBVersion

```
int ucdb_DBVersion(
```

db	Database

Returns integer version of the API library, or a negative value if error. If the database handle was created from a file (that is, ucdb_Open with non-NULL filename or ucdb_OpenReadStream) this call returns the version of the database file itself. That is, it returns the version of the API that originally created the file. Otherwise, (that is, ucdb_Open with NULL filename or ucdb_OpenWriteStream), this function is the same as ucdb_APIVersion().

ucdb_APIVersion

```
int ucdb_APIVersion(void)
```

Returns the current integer version of the API library. For a file to be readable, use the following:

```
ucdb_APIVersion() ©>= ucdb_DBVersion(db)
```

ucdb_SetPathSeparator

```
int ucdb_SetPathSeparator(
    ucdbT           db,
```

db	Database
separator	Path separator

Sets the path separator for the specified database. See Scopes. The path separator is stored with the persistent form of the database. Returns 0 if successful, or -1 if error.

ucdb_GetPathSeparator

```
char ucdb_GetPathSeparator(
```

db	Database

Returns the path separator for the specified database, or 0 if error.

ucdb_Filename

```
const char* ucdb_Filename(
```

db	Database

Returns the filename from which the specified database was read or the most recent filename written, or NULL if none.

User-Specified Attributes

User-defined attributes are associated with objects in the database (scopes, coveritems, or tests) or with the database itself (global attributes). They are key-value pairs that can be traversed or looked up by key.

Key-value string storage is maintained by the API. With *set* routines (which add key-value pairs), passed-in strings are copied to storage maintained by the API. You must not de-allocate individual strings returned by the API. On reading from or writing to memory, values returned are always owned by the API. They are good until the next call. The memory for keys is always good.

For attributes of coveritems, the coveritems are identified by a combination of the parent scope handle (pointer) and an integer index for the coveritem. To use the attribute functions for a scope only, the integer index must be set to -1. For history node objects, the index must always be -1. If a function is given an attribute handle, if that handle is of type UCDB_ATTR_ARRAY, then the index must be a value from 0 to *array size – 1*. The array size may be queried using the ucdb_AttrArraySize() function. If the attribute handle is of type UCDB_ATTR_HANDLE, then the index must be –1.

Attribute Type

```
typedef enum {
    UCDB_ATTR_INT,
    UCDB_ATTR_FLOAT,
    UCDB_ATTR_DOUBLE,
    UCDB_ATTR_STRING,
    UCDB_ATTR_MEMBLK,
    UCDB_ATTR_INT64,
    UCDB_ATTR_HANDLE,  /* Refers to other attributes: for nesting */
    UCDB_ATTR_ARRAY    /* Handle used to refer to an attribute array */
} ucdbAttrTypeT;
```

Attribute Value Type

```
typedef struct {
    ucdbAttrTypeT type;     /* Value type            */
    union {
        int64_t i64value    /* 64-bit integer value  */
        int ivalue;         /* Integer value         */
        float fvalue;       /* Float value           */
        double dvalue;      /* Double value          */
        const char* svalue; /* String value          */
        struct {
           int size;           /* Size of memory block, number of bytes */
            unsigned char* data; /* Starting address of memory block */
        } mvalue;
        ucdbAttrHandleT attrhandle; /* for HANDLE and ARRAY */
    } u;
} ucdbAttrValueT;
```

ucdb_AttrGetNext

```
const char* ucdb_AttrGetNext(
    ucdbT            db,
    ucdbObjT         obj,
    int              coverindex,
    const char*      key,
```

db Database.

obj Object type: ucdbScopeT, ucdbHistoryNodeT, or NULL (for global attribute).

coverindex Index of coveritem. If obj is ucdbScopeT, specify -1 for scope.

key Previous key or NULL to get the first attribute.

value Attribute value returned.

Returns the next attribute key and gets the corresponding attribute value from the specified database object, or returns NULL when done traversing attributes. Do not use free or strdup on keys. Memory for the returned key is owned by the API. To preserve the old key, just use another char* variable for it. For example, to traverse the list of attributes for a scope:

```
const char* key = NULL;
ucdbAttrValueT* value;
while (key = ucdb_AttrGetNext(db,obj,-1,key,&value)) {
   printf("Attribute ’%s’ is ", key);
   print_attrvalue(value);
}
```

ucdb_AttrAdd

```
int ucdb_AttrAdd(
    ucdbT            db,
    ucdbObjT         obj,
    int              coverindex,
    const char*      key,
```

db	Database
obj	Object type: ucdbScopeT, ucdbTestT, or NULL (for global attribute)
coverindex	Index of coveritem. If obj is ucdbScopeT, specify -1 for scope
key	Attribute key
value	Attribute value

Adds the specified attribute (key/value) to the specified database object or global attribute list. The attribute value is copied to the system. Returns 0 if successful, or -1 if error.

ucdb_AttrRemove

```
int ucdb_AttrRemove(
    ucdbT            db,
    ucdbObjT         obj,
    int              coverindex,
```

db	Database
obj	Object type: ucdbScopeT, ucdbTestT, or NULL (for global attribute)
coverindex	Index of coveritem. If obj is ucdbScopeT, specify -1 for scope
key	Key or NULL to remove the first attribute

Removes the attribute that has the specified key from the specified database object or global attribute list. Returns 0 if successful, or -1 if error.

ucdb_AttrGet

```
int ucdb_AttrGet(
    ucdbT           db,
    ucdbObjT        obj,
    int             coverindex,
    const char*     key,
```

db	Database
obj	Object type: ucdbScopeT, ucdbHistoryNodeT, or NULL (for global attribute)
coverindex	Index. If obj is ucdbScopeT, specify -1 for scope. Valid index for coveritem is ucdbAttrHandleT: • array index (if type is UCDB_ATTR_ARRAY) • -1 (if type is UCDB_ATTR_HANDLE)
key	Not necessary if obj is ucdbAttrHandleT and its type is UCDB_ATTR_ARRAY.
value	Attribute value returned

Gets the attribute value for the specified object/key or global attribute value if obj is NULL. Returns 1 if a match is found, or 0 if error.

ucdb_AttrArraySize

```
int ucdb_AttrArraySize(
    ucdbT           db,
```

db	Database
arrayhandle	Attribute array handle

Returns the size (max index + 1) of the attribute array, or -1 if error (that is, type is not UCDB_ATTR_ARRAY).

Scopes

Scopes functions manage the design hierarchy and coverage scopes. The UCDB database is organized hierarchically in parallel with the design database, which consists of a tree of module instances, each of a given module type.

Scopes functions contain the following components:

- Hierarchical identifiers

  - If a scope type is Verilog or SystemVerilog, Verilog-escaped identifiers syntax is assumed for a path within that scope.
  - If a scope type is VHDL, VHDL-extended identifiers are assumed. The escaped identifier syntax is sensitive to the scope type so that escaped identifiers can appear in the your preferred syntax. If a scope type is VHDL, the entity, architecture and library can be encoded in the name.

- Attributes
  - char* attributes can be omitted with a NULL value.
  - int attributes can be omitted with a negative value.

Scope Type

```
typedef unsigned int ucdbScopeTypeT;
#define UCDB_TOGGLE            UCIS_TOGGLE
                              /* cover scope: toggle coverage scope */
#define UCDB_BRANCH            UCIS_BRANCH
                              /* cover scope: branch coverage scope */
#define UCDB_EXPR              UCIS_EXPR
                              /* cover scope: expression coverage scope */
#define UCDB_COND              UCIS_COND
                              /* cover scope: condition coverage scope */
#define UCDB_INSTANCE          UCIS_INSTANCE
                              /* HDL scope: Design hierarchy instance */
#define UCDB_PROCESS           UCIS_PROCESS
                              /* HDL scope: process */
#define UCDB_BLOCK             UCIS_BLOCK
                              /* HDL scope: vhdl block, vlog begin-end */
#define UCDB_FUNCTION          UCIS_FUNCTION
                              /* HDL scope: function */
#define UCDB_FORKJOIN          UCIS_FORKJOIN
                              /* HDL scope: Verilog fork-join block */
#define UCDB_GENERATE          UCIS_GENERATE
                              /* HDL scope: generate block */
#define UCDB_GENERIC           UCIS_GENERIC
                              /* cover scope: generic scope type */
#define UCDB_CLASS             UCIS_CLASS
                              /* HDL scope: class type scope */
#define UCDB_COVERGROUP        UCIS_COVERGROUP
                              /* cover scope: covergroup type scope */
#define UCDB_COVERINSTANCE     UCIS_COVERINSTANCE
                              /* cover scope: covergroup instance scope */
#define UCDB_COVERPOINT        UCIS_COVERPOINT
                              /* cover scope: coverpoint scope */
#define UCDB_CROSS             UCIS_CROSS
                              /* cover scope: cross scope */
#define UCDB_COVER             UCIS_COVER
                              /* cover scope: directive (SVA/PSL) cover */
#define UCDB_ASSERT            UCIS_ASSERT
                              /* cover scope: directive (SVA/PSL) assert */
#define UCDB_PROGRAM           UCIS_PROGRAM
                              /* HDL scope: SV program instance */
#define UCDB_PACKAGE           UCIS_PACKAGE
                              /* HDL scope: package instance */
#define UCDB_TASK              UCIS_TASK
                              /* HDL scope: task */
#define UCDB_INTERFACE         UCIS_INTERFACE
                              /* HDL scope: SV interface instance */
#define UCDB_FSM               UCIS_FSM
                              /* cover scope: FSM coverage scope */
#define UCDB_TESTPLAN          UCIS_TESTPLAN
                              /* test scope: for testplan item */
#define UCDB_DU_MODULE         UCIS_DU_MODULE
                              /* design unit: for instance type */
#define UCDB_DU_ARCH           UCIS_DU_ARCH
                              /* design unit: for instance type */
#define UCDB_DU_PACKAGE        UCIS_DU_PACKAGE
                              /* design unit: for instance type */
#define UCDB_DU_PROGRAM        UCIS_DU_PROGRAM
```

```
                                /* design unit: for instance type */
    #define UCDB_DU_INTERFACE    UCIS_DU_INTERFACE
                                /* design unit: for instance type */
    #define UCDB_FSM_STATES      UCIS_FSM_STATES
                                /* cover scope: FSM states coverage scope */
    #define UCDB_FSM_TRANS       UCIS_FSM_TRANS
                                /* cover scope: FSM transitions
                                coverage scope*/
    #define UCDB_GROUP           INT64_LITERAL(0x0000001000000000)
                                /* group scope */
    #define UCDB_TRANSITION      INT64_LITERAL(0x0000002000000000)
                                /* cover scope: covergroup transition scope */
    #define UCDB_RESERVED_SCOPE    INT64_LITERAL(0xFF00000000000000)
                                  /* RESERVED scope type */
    #define UCDB_SCOPE_ERRORUCDB_SCOPE_ERROR
          INT64_LITERAL(0x0000000000000000)   /* error return code */
    #define UCDB_FSM_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_FSM | UCDB_FSM_STATES | UCDB_FSM_TRANS))
    #define UCDB_CODE_COV_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_BRANCH | UCDB_EXPR | UCDB_COND | UCDB_TOGGLE | UCDB_FSM_SCOPE | \
       UCDB_BLOCK))
    #define UCDB_DU_ANY ((ucdbScopeMaskTypeT) \
       (UCDB_DU_MODULE | UCDB_DU_ARCH | UCDB_DU_PACKAGE | \
       UCDB_DU_PROGRAM | UCDB_DU_INTERFACE))
    #define UCDB_CVG_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_COVERGROUP | UCDB_COVERINSTANCE | UCDB_COVERPOINT | UCDB_CROSS))
    #define UCDB_FUNC_COV_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_CVG_SCOPE | UCDB_COVER))
    #define UCDB_COV_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_CODE_COV_SCOPE | UCDB_FUNC_COV_SCOPE)\
    #define UCDB_VERIF_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_COV_SCOPE | UCDB_ASSERT | UCDB_GENERIC))
    #define UCDB_HDL_SUBSCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_PROCESS | UCDB_BLOCK | UCDB_FUNCTION | UCDB_FORKJOIN | \
       UCDB_GENERATE | UCDB_CLASS | UCDB_TASK))
    #define UCDB_HDL_INST_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_INSTANCE | UCDB_PROGRAM | UCDB_PACKAGE | UCDB_INTERFACE))
    #define UCDB_HDL_DU_SCOPE ((ucdbScopeMaskTypeT) (UCDB_DU_ANY))
    #define UCDB_HDL_SCOPE ((ucdbScopeMaskTypeT) \
       (UCDB_HDL_SUBSCOPE | UCDB_HDL_INST_SCOPE | UCDB_HDL_DU_SCOPE))
    #define UCDB_NONTESTPLAN_SCOPE ((ucdbScopeMaskTypeT) (~UCDB_TESTPLAN))
    #define UCDB_NO_SCOPES  ((ucdbScopeMaskTypeT)INT64_ZERO)
    #define UCDB_ALL_SCOPES ((ucdbScopeMaskTypeT)INT64_NEG1)
```

Source Type

Enumerated type to encode the source type of a scope, if needed. Scope type can have an effect on how the system regards escaped identifiers within the design hierarchy.

```
typedef enum {
   UCDB_VHDL,
   UCDB_VLOG,          /* Verilog                           */
   UCDB_SV,            /* SystemVerilog                     */
   UCDB_SYSTEMC,
   UCDB_PSL_VHDL,      /* assert/cover in PSL VHDL          */
   UCDB_PSL_VLOG,      /* assert/cover in PSL Verilog       */
   UCDB_PSL_SV,        /* assert/cover in PSL SystemVerilog */
   UCDB_PSL_SYSTEMC,   /* assert/cover in PSL SystemC       */
   UCDB_E,
   UCDB_VERA,
   UCDB_NONE,          /* not important                     */
   UCDB_OTHER,         /* user-defined attribute            */
   UCDB_VLOG_AMS,      /* Verilog Analog Mixed Signal */
   UCDB_VHDL_AMS,      /* VHDL Analog Mixed Signal */
   UCDB_SPICE,
   UCDB_MATLAB,
   UCDB_C,
   UCDB_CPP,
   UCDB_SOURCE_ERROR = -1 /* for error cases */
} ucdbSourceT;
```

Flags Type

```
typedef ucdbFlagsT int ucdbFlagsT;
/* Flags for scope data */
#define UCDB_INST_ONCE  UCIS_INST_ONCE /* Instance is instantiated only
                                       once; code coverage is stored only
                                       in the instance. */
/* Flags that indicate whether the scope was compiled with the         */
/* corresponding type of code coverage enabled.                        */
#define UCDB_ENABLED_STMT     UCIS_ENABLED_STMT  /* statement coverage */
#define UCDB_ENABLED_BRANCH   UCIS_ENABLED_BRANCH/* branch coverage    */
#define UCDB_ENABLED_COND     UCIS_ENABLED_COND  /* condition coverage */
#define UCDB_ENABLED_EXPR     UCIS_ENABLED_EXPR  /* expression coverage*/
#define UCDB_ENABLED_FSM      UCIS_ENABLED_FSM   /* FSM coverage       */
#define UCDB_ENABLED_TOGGLE   UCIS_ENABLED_TOGGLE/* toggle coverage    */
#define UCDB_ENABLED_TOGGLEEXT 0x00000080 /* 3-state toggle;          */
                                         /* not used in ucis.h         */
#define UCDB_SCOPE_UNDER_DU   UCIS_SCOPE_UNDER_DU/* whether or not     */
                                        /* scope is under a design unit*/
#define UCDB_SCOPE_EXCLUDED        UCIS_SCOPE_EXCLUDED
#define UCDB_SCOPE_PRAGMA_EXCLUDED UCIS_SCOPE_PRAGMA_EXCLUDED
#define UCDB_SCOPE_PRAGMA_CLEARED  UCIS_SCOPE_PRAGMA_CLEARED
#define UCDB_SCOPE_GOAL_SPECIFIED  0x00400000
#define UCDB_SCOPE_AUTO_EXCLUDED   0x00008000
#define UCDB_IS_TOP_NODE          UCIS_IS_TOP_NODE /* for top-level    */
                                                   /* toggle node      */
#define UCDB_IS_IMMEDIATE_ASSERT   UCIS_IS_IMMEDIATE_ASSERT /*for SV   */
                                         /* immediate asserts          */
/* Reuse these two flag values for covergroup scopes */
#define UCDB_IS_E_PER_INST        0x00020000  /* for covergroup */
#define UCDB_IS_E_PER_TYPE         0x00010000 /* instance scopes */
/* For Zero Information in "flags" */
#define UCDB_SCOPE_IFF_EXISTS      UCIS_SCOPE_IFF_EXISTS
#define UCDB_SCOPE_SAMPLE_TRUE    UCIS_SCOPE_SAMPLE_TRUE /* No bin     */
                                          /* under the scope is sampled*/
/* Two-bit Expression/Condition short circuit information flags applicable
   to UCDB_EXPR and UCDB_COND scopes only. Two bits are overloaded by
   re-using UCDB_SCOPE_IFF_EXISTS and UCDB_SCOPE_SAMPLE_TRUE flags which
   are applicable to the covergroup scopes only.The two bits carry
   meaningful information only when used together:
      00: Short circuit enabled
      01: Short circuit partially enabled
      10: Short circuit disallowed
      11: Short circuit disabled (Same as flag UCDB_SCOPE_SAMPLE_TRUE)
/* Flags that specify whether the short circuit is enabled or disabled at
   the Design Unit level. */
#define UCDB_SCOPE_SCKT_PART_ENABLED 0x00100000
#define UCDB_SCOPE_SCKT_DISALLOWED   0x00200000
#define UCDB_SCOPE_SCKT_DISABLED     0x00300000
/* Flag for checking if DU had short circuiting disabled for coverage */
#define UCDB_DISABLED_SHORTCKT     0x00400000
/* Flag for checking if a DU had UDP coverage enabled for expr/cond
   coverage */
#define UCDB_EXPRCOND_UDP          0x00800000
/*Flag for checking if it is a PA coverage scope */
#define UCDB_PACOVERAGE            0x02000000
/* Flag used only on bimodal expressions to trigger Extended FEC
   Analysis */
```

```
#define UCDB_EXPRCOND_EXT_FEC      0x01000000
/* Flag used to trigger matching input patterns reporting in reports */
#define UCDB_EXPRCOND_NOREC        0x20000000
/* Flag set on last row of Extended FEC table */
#define UCDB_EXPRCOND_LAST_FEC_ROW 0x00080000
#define UCDB_IS_ASSERT_DEBUG       0x10000000 /* for assert directives */
                                             /* if true, has 4 counts */
#define UCDB_SCOPEFLAG_MARK UCIS_SCOPEFLAG_MARK /* flag for     */
                                              /* temporary mark        */
#define UCDB_SCOPE_INTERNAL UCIS_SCOPE_INTERNAL /* flags for    */
                                             /* internal use          */
```

ucdb_MatchScopeByName

```
ucdbScopeT ucdb_MatchScopeByName(

    ucdbT               db,

    ucdbScopeT          parent,
```

db	Database
parent	Parent scope
name	Scope name to match

(Deprecated) Returns a handle to the scope with the specified name in the parent scope, or NULL if error.

Note

This function is deprecated because it is not designed to handle scopes with the same name and the same parent. The best approach is a linear traversal of subscopes and name comparison with strcmp. However, this function works if there is only one match for the given name in the parent.

ucdb_MatchScopeByPath

```
ucdbScopeT ucdb_MatchScopeByPath(

    ucdbT               db,
```

db	Database
pathname	Path name for scope The path separator is the one currently in use by the database (see ucdb_SetPathSeparator and ucdb_GetPathSeparator).

(Deprecated) Returns a handle to the scope with the specified path name, or NULL if error.

Note

This function is deprecated because it is not designed to handle scopes with the same name and the same parent. Instead, use ucdb_PathCallBack(), which has the added advantage of wildcard matching. However, this function works well if there is only one object named as specified for every component of the path.

ucdb_CreateScope

```
ucdbScopeT ucdb_CreateScope(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    ucdbSourceInfoT*        srcinfo,
    int                     weight,
    ucdbSourceT             source,
    ucdbScopeTypeT          type,
```

db	Database
parent	Parent scope If NULL, creates the root scope.
name	Name to assign to scope
srcinfo	Associated source information. Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight.
source	Source of scope
type	Type of scope to create
flags	Flags for the scope

Creates the specified scope beneath the parent scope. Returns the scope handle if successful, or NULL if error. In write-streaming mode, "name" is not copied, so it should be kept unchanged until the next ucdb_WriteStream* call or the next ucdb_Create* call.

Use ucdb_CreateInstance for UCDB_INSTANCE or UCDB_COVERINSTANCE scopes.

ucdb_ComposeDUName

```
const char*
ucdb_ComposeDUName(
    const char*             library_name,
    const char*             primary_name,
```

library_name	Library name
primary_name	Primary name

secondary_name	Secondary name

Composes as design unit scope name for a specified design unit. Returns the handle to the parsed design unit scope name for the specified component names, or -1 if error. The ucdb_ComposeDUName and ucdb_ParseDUName utilities use a static dynamic string (one for the “Compose” function, one for the “Parse” function), so values are only valid until the next call to the respective function. To hold a name across separate calls, you must copy it.

ucdb_ParseDUName

```
void ucdb_ParseDUName(
    const char*         du_name,
    const char**        library_name,
    const char**        primary_name,
```

du_name	Design unit name to parse
library_name	Library name returned by the call
primary_name	Primary name returned by the call
secondary_name	Secondary name returned by the call

Gets the library name, primary name, and secondary name for the design unit specified by du_name. Design unit scope name has the form:

```
library_name.primary_name(secondary_name)
```

The ucdb_ComposeDUName and ucdb_ParseDUName utilities use a static dynamic string (one for the “Compose” function, one for the “Parse” function), so values are only valid until the next call to the respective function. To hold a name across separate calls, you must copy it.

ucdb_CreateInstance

```
ucdbScopeT ucdb_CreateInstance(
    ucdbT               db,
    ucdbScopeT          parent,
    const char*         name,
    ucdbSourceInfoT*    fileinfo,
    int                 weight,
    ucdbSourceT         source,
    ucdbScopeTypeT      type,
    ucdbScopeT          du_scope,
```

db	Database
parent	Parent of instance scope If NULL, creates a new root scope.
name	Name to assign to scope

fileinfo	Associated source information Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight.
source	Source of instance
type	Type of scope to create: UCDB_INSTANCE or UCDB_COVERINSTANCE
du_scope	Previously-created scope that is usually the design unit If type is UCDB_INSTANCE, then du_scope has type UCDB_DU_*. If type is UCDB_COVERINSTANCE, then du_scope has type UCDB_COVERGROUP to capture the instance type of the instance relationship for the covergroup instance.
flags	Flags for the scope

Creates an instance scope of the specified design unit type under the specified parent. Not supported in streaming modes; use ucdb_CreateInstanceByName() in write-streaming mode. Returns a scope handle, or NULL if error.

ucdb_CreateInstanceByName

```
ucdbScopeT ucdb_CreateInstanceByName(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    ucdbSourceInfoT*        fileinfo,
    int                     weight,
    ucdbSourceT             source,
    ucdbScopeTypeT          type,
    char*                   du_name,
```

db	Database.
parent	Parent of instance scope In write-streaming mode, should be NULL. For other modes, NULL creates a root scope.
name	Name to assign to scope
fileinfo	Associated source information Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight.
source	Source of instance

type	Type of scope to create: UCDB_INSTANCE or UCDB_COVERINSTANCE
du_name	Name of previously-created scope of the instance’s design unit or the coverinstance’s covergroup type
flags	Flags for the scope

Creates an instance of the specified named design unit under the specified parent scope. Returns a scope handle, or NULL if error.

ucdb_CreateCross

```
ucdbScopeT ucdb_CreateCross(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    ucdbSourceInfoT*        fileinfo,
    int                     weight,
    ucdbSourceT             source,
    int                     num_points,
```

db	Database
parent	Parent scope: UCDB_COVERGROUP or UCDB_COVERINSTANCE.
name	Name to assign to cross scope
fileinfo	Associated source information Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight.
source	Source of cross
num_points	Number of crossed coverpoints
points	Array of scopes of the coverpoints that comprise the cross scope These coverpoints must already exist in the parent.

Creates the specified cross scope under the specified parent (covergroup or cover instance) scope. Returns a scope handle for the cross, or NULL if error.

ucdb_CreateCrossByName

```
ucdbScopeT ucdb_CreateCrossByName(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    ucdbSourceInfoT*        fileinfo,
    int                     weight,
    ucdbSourceT             source,
    int                     num_points,
```

db	Database
parent	Parent scope: UCDB_COVERGROUP or UCDB_COVERINSTANCE.
name	Name to assign to cross scope
fileinfo	Associated source information Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight.
source	Source of cross
num_points	Number of crossed coverpoints
point_names	Array of names of the coverpoints that comprise the cross scope These coverpoints must already exist in the parent.

Creates the specified cross scope under the specified parent (covergroup or cover instance) scope. Returns a scope handle for the cross, or NULL if error.

ucdb_CreateTransition

```
ucdbScopeT ucdb_CreateTransition(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    ucdbSourceInfoT*        fileinfo,
    int                     weight,
    ucdbSourceT             source,
```

db	Database
parent	Parent scope: UCDB_COVERGROUP or UCDB_COVERINSTANCE
name	Name of coveritem Can be NULL.

fileinfo	Associated source information Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight.
source	Source of the transition
item	Array of coverpoint scopes Must exist in the parent.

Creates a transition scope under the given parent. In write-streaming mode, *name* is not copied; it should be preserved unchanged until the next ucdb_WriteStream* call or the next ucdb_Create* call. Returns the scope pointer, or NULL if error.

ucdb_CreateTransitionByName

```
ucdbScopeT ucdb_CreateTransitionbyName(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    ucdbSourceInfoT*        fileinfo,
    int                     weight,
    ucdbSourceT             source,
```

db	Database
parent	Parent scope: UCDB_COVERGROUP or UCDB_COVERINSTANCE
name	Name of coveritem Can be NULL.
fileinfo	Associated source information Can be NULL.
weight	Weight to assign to the scope Negative indicates no weight. Not applicable to toggles
source	Source of the transition
item_name	Transition item Must exist in the parent

Creates a transition scope under the given parent. In write-streaming mode, name is not copied; it should be preserved unchanged until the next ucdb_WriteStream* call or the next ucdb_Create* call. Returns the scope pointer, or NULL if error.

ucdb_InstanceSetDU

```
int ucdb_InstanceSetDU(
    ucdbT                   db,
    ucdbScopeT              instance,
```

db	Database (must contain instance and du_scope)
instance	Scope of the instance
du_scope	Previously-created scope that is usually the design unit If type is UCDB_INSTANCE, then du_scope has type UCDB_DU_*. If type is UCDB_COVERINSTANCE, then du_scope has type UCDB_COVERGROUP to capture the instance type of the instance relationship for the covergroup instance.

Sets the specified design unit scope handle in the specified instance. Returns 0 if successful, or -1 if error.

ucdb_CloneScope

```
ucdbScopeT ucdb_CloneScope(
    ucdbT                   targetdb,
    ucdbScopeT              targetparent,
    ucdbT                   sourcedb,
    ucdbScopeT              scope,
    ucdbSelectFlagsT        cloneflags,
```

targetdb	Database context for clone
targetparent	Parent scope of clone
sourcedb	Source database
scope	Source scope to clone
cloneflags	Flags specifying what to copy
is_recursive	If non-zero, recursively clones subscopes. If 0, only clones the specified scope.

Has no effect when targetdb is in streaming mode. Creates a copy of the specified scope under the specified destination scope (targetparent). Predefined attributes are created by default. Returns the scope handle of the cloned scope, or -1 if error.

ucdb_RemoveScope

```
int ucdb_RemoveScope(
    ucdbT                   db,
```

db	Database

scope	Scope to remove

Has no effect when db is in streaming mode. Removes the specified scope from its parent scope, along with all its subscopes and coveritems. When a scope is removed, that scope handle immediately becomes invalid along with all of its subscope handles. Those handles cannot be used in any API routines. Returns 0 if successful, or -1 if error.

ucdb_ScopeParent

```
ucdbScopeT ucdb_ScopeParent(
    ucdbT                    db,
```

db	Database
scope	Scope

Returns the parent scope handle of the specified scope, or NULL if none or error.

ucdb_ScopeGetTop

```
ucdbScopeT ucdb_ScopeGetTop(
    ucdbT                    db,
```

db	Database
scope	Scope

Returns the top-level scope (that is, the scope with no parent) above the specified scope, or NULL if error.

ucdb_GetScopeName

```
const char* ucdb_GetScopeName(
    ucdbT                    db,
```

db	Database
scope	Scope

Returns the non-hierarchical string name of the specified scope, or NULL if error.

ucdb_SetScopeName

```
int ucdb_SetScopeName(
    ucdbT                    db,
    ucdbScopeT               scope,
```

db	Database

scope	Scope
name	Name to assign to scope

Sets the name of the specified scope. Returns -1 if error.

ucdb_GetScopeType

```
ucdbScopeTypeT ucdb_GetScopeType(
    ucdbT                   db,
```

db	Database
scope	Scope

Returns the scope type of the specified scope, or UCDB_SCOPE_ERROR if error.

ucdb_GetScopeSourceType

```
ucdbSourceT ucdb_GetScopeSourceType(
    ucdbT                   db,
```

db	Database
scope	Scope

Returns the source of the specified scope, or UCDB_SOURCE_ERROR if error.

ucdb_GetScopeFlags

```
ucdbFlagsT ucdb_GetScopeFlags(
    ucdbT                   db,
```

db	Database
scope	Scope

Returns the scope flags of the specified scope, or -1 if error.

ucdb_SetScopeFlags

```
void ucdb_SetScopeFlags(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	Scope
flags	Flags to assign to scope

Sets the flags of the specified scope.

ucdb_GetScopeFlag

```
int ucdb_GetScopeFlag(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	Scope
mask	Flag bit to match with scope flags.

Returns 1 if the scope’s flag bit matches the specified mask, otherwise, no match.

ucdb_SetScopeFlag

```
void ucdb_SetScopeFlag(
    ucdbT                   db,
    ucdbScopeT              scope,
    ucdbFlagsT              mask,
```

db	Database
scope	Scope
mask	Flag bits to set
bitvalue	Value (0 or 1) to set mask bits

Sets bits in the scope’s flags fields corresponding to the mask to the specified bit value (0 or 1).

ucdb_GetScopeSourceInfo

```
int ucdb_GetScopeSourceInfo(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	Scope
sourceinfo	Returned source information (file/line/token) Memory for source information string is allocated by the system and must not be de-allocated by the user.

Gets the source information for the specified scope. Returns 0 if successful, or non-zero if error.

ucdb_SetScopeSourceInfo

```
int ucdb_SetScopeSourceInfo(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database.
scope	Scope
sourceinfo	Source information (file/line/token) to store for the specified scope

Sets the source information for the specified scope. Returns 0 if successful, or non-zero if error.

ucdb_SetScopeFileHandle

```
int ucdb_SetScopeFileHandle(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	Scope
filehandle	Filehandle to set for the scope

Sets the filehandle for the specified scope. Does not apply to toggle nodes. API maintains the filehandle string storage; do not free. Returns 0 if successful, or non-zero if error.

ucdb_GetScopeWeight

```
int ucdb_GetScopeWeight(
    ucdbT                   db,
```

db	Database
scope	Scope

Returns the weight for the specified scope, or -1 if error. Toggle nodes have no weight and always return 1.

ucdb_SetScopeWeight

```
int ucdb_SetScopeWeight(
    ucdbT                   db,
    ucdbScopeT              scope,
```

db	Database
scope	Scope

weight	Weight to assign to scope

Sets the weight for the specified scope. Returns 0 if successful, or -1 if error. Not applicable to toggle nodes.

ucdb_GetScopeGoal

```
int ucdb_GetScopeGoal(
    ucdbT           db,
    ucdbScopeT      scope,
```

db	Database
scope	Scope
goal	Goal returned

Gets the goal for the specified scope. For UCDB_CVG_SCOPE type, converts from the integer value (see ucdb_SetScopeGoal). Returns 1 if found, or 0 if not found. Not applicable to toggle nodes.

ucdb_SetScopeGoal

```
int ucdb_SetScopeGoal(
    ucdbT           db,
    ucdbScopeT      scope,
```

db	Database
scope	Scope
goal	Goal value

Sets the goal for the specified scope. For UCDB_CVG_SCOPE types, converts to the integer value (in the SystemVerilog LRM, option.goal and type_option.goal are defined as integers). Returns 0 if successful, or -1 if error. Not applicable to toggle nodes.

ucdb_GetScopeHierName

```
const char* ucdb_GetScopeHierName(
    ucdbT               db,
```

db	Database
scope	Scope

Returns the pointer to the hierarchical name of scope, or NULL if error. Hierarchical path separator is as set for the current database.

ucdb_GetInstanceDU

```
ucdbScopeT ucdb_GetInstanceDU(
    ucdbT                 db,
```

db	Database
scope	Instance scope (that is, scope type is UCDB_INSTANCE)

Returns the handle of the design unit scope of the specified instance scope, or NULL if error. This call can return the UCDB_COVERGROUP scope for a UCDB_COVERINSTANCE as well.

ucdb_GetInstanceDUName

```
char* ucdb_GetInstanceDUName(
    ucdbT         db,
```

db	Database
scope	Instance scope (that is, scope type is UCDB_INSTANCE)

Returns the handle of the design unit scope name of the specified instance scope, or NULL if error. This call can return the UCDB_COVERGROUP scope name for a UCDB_COVERINSTANCE as well. Handle must not to be de-allocated or saved in streaming modes. If not in in-memory mode, handle must be copied.

ucdb_GetNumCrossedCvps

```
int ucdb_GetNumCrossedCvps(
    ucdbT                 db,
    ucdbScopeT            scope,
```

db	Database
scope	Cross scope
num_points	Number of coverpoints returned

Gets the number of crossed coverpoints of the specified cross scope. Returns 0 if successful, or non-zero if error.

ucdb_GetIthCrossedCvp

```
int ucdb_GetIthCrossedCvp(
    ucdbT                 db,
    ucdbScopeT            scope,
    int                   index,
```

db	Database
scope	Cross scope
index	Coverpoint index in the cross scope
point_scope	Crossed coverpoint scope returned

Gets the crossed coverpoint of the scope specified by the coverpoint index in the specified cross scope. Returns 0 if successful, or non-zero if error.

ucdb_GetIthCrossedCvpName

```
char* ucdb_GetIthCrossedCvpName(
    ucdbT        db,
    ucdbScopeT   scope,
```

db	Database
scope	Cross scope
index	Coverpoint index in the cross scope

Returns the handle of the name of the crossed coverpoint of the scope specified by the coverpoint index in the specified cross scope, or NULL if error.

ucdb_GetTransitionItem

```
ucdbScopeT ucdb_GetTransitionItem(
    ucdbT               db,
```

db	Database
scope	Transition scope

Returns the transition item scope, or NULL if error (for example, scope is not a transition scope).

ucdb_GetTransitionItemName

```
char* ucdb_GetTransitionItemName(
    ucdbT               db,
```

db	Database
scope	Transition scope

Returns the transition item scope name, or NULL if error (for example, scope is not a transition scope).

ucdb_NextPackage

```
ucdbScopeT ucdb_NextPackage(
    ucdbT               db,
```

db	Database
package	Package or NULL to return the first package

Returns the next package following the specified package in the database, NULL if package is the last package, or UCDB_SCOPE_ERROR if error.

ucdb_NextDU

```
ucdbScopeT ucdb_NextDU(
    ucdbT               db,
```

db	Database
du	Design unit or NULL to return the first design unit

Returns the next design unit following the specified design unit in the database, NULL if package is the last package, or UCDB_SCOPE_ERROR if error.

ucdb_MatchDU

```
ucdbScopeT ucdb_MatchDU(
    ucdbT               db,
```

db	Database
name	Design unit name to match

Returns the design unit scope with the specified name, or NULL if no match is found.

ucdb_NextSubScope

```
ucdbScopeT ucdb_NextSubScope(
    ucdbT               db,
    ucdbScopeT          parent,
    ucdbScopeT          scope,
```

db	Database
parent	Parent scope or NULL for top-level modules
scope	Previous child scope or NULL to start traversal
scopemask	Scope type mask

Returns the next child scope in the iteration that has a scope type that matches the specified scope mask, or NULL if last element or error. Setting scope == NULL starts the traversal; replacing scope with the previous returned scope runs the next iteration; a return value of NULL indicates the call is the last iteration. If parent scope is NULL, the iteration is through the top-level modules in the design.

ucdb_NextScopeInDB

```
ucdbScopeT ucdb_NextScopeInDB(
    ucdbT               db,
    ucdbScopeT          scope,
```

db	Database
scope	Previous child scope or NULL to start traversal
scopemask	Scope type mask

Returns the next child scope in the iteration that has a scope type that matches the specified scope mask, or NULL if last element or error. Setting scope == NULL starts the traversal; replacing scope with the previous returned scope runs the next iteration; a return value of NULL indicates the call is the last iteration. Traversal starts with the first top level scope in the database and iterates through all matching scopes.

ucdb_NextInstOfDU

```
ucdbScopeT ucdb_NextInstOfDU(
    ucdbT               db,
    ucdbScopeT          instance,
```

db	Database
instance	Previous instance or NULL to start traversal
du	Design unit scope (that is, UCDB_DU_*)

Returns the next instance in the iteration, or NULL if last element or error. Setting instance == NULL starts the traversal; replacing instance with the previous returned instance runs the next iteration; a return value of NULL indicates the call is the last iteration.

ucdb_ScopeIsUnderDU

```
int ucdb_ScopeIsUnderDU(
    ucdbT               db,
```

db	Database
scope	Scope

Returns 1 if scope is under a design unit (scope type is in UCDB_HDL_DU_SCOPE), 0 if not, or -1 if error. Does not work currently for scopes beneath single-instance design units, because of UCDB_INST_ONCE optimization (where the node is under the instance).

ucdb_ScopeIsUnderCoverInstance

```
int ucdb_ScopeIsUnderCoverInstance(
    ucdbT                    db,
```

db	Database
scope	Scope

Returns 1 if scope is under a UCDB_COVERINSTANCE scope (scope type must be UCDB_COVERPOINT or UCDB_CROSS), 0 if not, or -1 if error.

ucdb_CallBack

```
int ucdb_CallBack(
    ucdbT         db,
    ucdbScopeT    start,
    ucdb_CBFuncT  cbfunc,
```

db	Database
start	Starting scope or NULL to traverse entire database
cbfunc	User-supplied callback function
userdata	User-supplied function data

In-memory mode only. Traverses the part of the database rooted at and below the specified starting scope, issuing calls to cbfunc along the way. Returns 0 if successful, or -1 with error.

ucdb_PathCallBack

```
int ucdb_PathCallBack(
    ucdbT                db,
    int                  recurse,
    const char*          path,
    const char*          du_name,
    ucdbScopeMaskTypeT   root_mask,
    ucdbScopeMaskTypeT   scope_mask,
    ucdbScopeMaskTypeT   cover_mask,
    ucdb_CBFuncT         cbfunc,
```

db	Database

recurse

Non-recursive if 0. If non-zero, recurse from matched du_name or scopes specified by path. scope_mask and cover_mask are applied AFTER recursion. Recursion proceeds from all scopes matching the (possibly wildcarded) path, after which callbacks are generated only for scopes and covers (including those specified by the path itself) that share a bit with the scope or cover mask.

path

Path interpreted as follows:

- if du_name==NULL: absolute path.
- if du_name!=NULL: path is relative to design units matching du_name.

If path is "/" it is treated as "*", which matches all roots or all paths under a design unit. Wildcards can be given to match multiple results. Uses UCDB path separator and escaped identifier rules in a context-sensitive fashion. Current wildcard symbols:

* — matches any substring within a level of hierarchy

? — preceding character is optional

[int:int] — matches any integer index in range

{int | *} to {int | *} — matches any integer index in range

{int | *} downto {int | *} — matches any integer index in range

To match wildcard characters literally, use the appropriate escaped identifier syntax.

du_name

Design unit name. Name is specified in the form:

library.primary(secondary)

where secondary matches for VHDL only. Multiple matches are possible if *library* or *secondary* is absent (even for Verilog design units, if the simulator created an artificial secondary). If path is also specified, then path is relative to all matching design units.

root_mask

If set, matches start from a root that satisfies 1 bit of this mask. Ignored if du_name is specified as this field applies to the top level only. Typically set to UCDB_TESTPLAN or UCDB_NON-TESTPLAN_SCOPE to choose a testplan tree or non-testplan tree.

scope_mask

Only match scopes that satisfy 1 bit of this mask

cover_mask

Only match coveritems that satisfy 1 bit of this mask

cbfunc

User-supplied callback function. Only these callback reasons (ucdbCBReasonT) are generated: UCDB_REASON_DU, UCDB_REASON_SCOPE, UCDB_REASON_CVBIN, and UCDB_REASON_ENDSCOPE.

userdata

User-supplied function data

In-memory mode only. This callback mechanism is more flexible than ucdb_CallBack (it implements wildcarded paths, filtering according to type, and so on). Traverses the database as specified, issuing calls to cbfunc as specified along the way. Returns number (0 or more) of matches, or -1 if error. When recursing through a testplan scope, the scope has design or coverage scopes (“virtual children”) with which it is linked through common tags. This link shows that these scopes contribute to the testplan scope’s coverage. When matching children of a testplan scope, both the real testplan children and the scopes linked to the testplan scope with tags are matched.

Examples:

```
ucdb_PathCallBack(db,0,"/top/a*",NULL,UCDB_NONTESTPLAN_SCOPE,\
      UCDB_HDL_INST_SCOPE,0,f,d);
```

Callback for all HDL instance scopes that start with “/top/a”.

```
ucdb_PathCallBack(db,0,NULL,"duname",-1,-1,0,f,d);
```

Callback for all design units with the name “duname”. This may match multiple architectures or library implementations of the design unit.

```
ucdb_PathCallBack(db,0,"myvec*","work.duname(myarch)", \
      -1,UCDB_TOGGLE,0,f,d);
```

Within the VHDL architecture “work.duname(myarch)”, callback for all toggle scopes whose names start with “myvec”.

```
ucdb_PathCallBack(db,1,"/top/a",NULL,UCDB_NONTESTPLAN_SCOPE, \
      UCDB_COVERGROUP|UCDB_COVERPOINT|UCDB_CROSS,0,f,d);
```

Callback for all covergroup, cross, and coverpoint scopes that lie under “/top/a”. Only if “/top/a” is a covergroup scope will “/top/a” itself be a callback.

```
ucdb_PathCallBack(db,1,"/top/a",NULL,UCDB_NONTESTPLAN_SCOPE,
      UCDB_COVERGROUP|UCDB_COVERPOINT|UCDB_CROSS,UCDB_CVGBIN,f,d);
```

This callback includes bin callbacks, as well.

ucdb_MatchTests

```
int ucdb_MatchTests(
    ucdbT           db,
    const char*     testname,
    ucdb_CBFuncT    cbfunc,
```

db	Database.

testname

Test name pattern. Current wildcard symbols:

- * — matches any substring within a level of hierarchy
- ? — preceding character is optional

To match wildcard characters literally, the appropriate escaped identifier syntax must be used.

cbfunc

User-supplied callback function. Only UCDB_REASON_TEST callback reasons (ucdbCBReasonT) are generated.

userdata

User-supplied function data.

In-memory mode only. Generates callbacks for tests whose testname attribute matches the specified testname pattern. Returns number (0 or more) of matches, or -1 if error.

ucdb_MatchCallBack

```
int ucdb_MatchCallBack(
    ucdbT               db,
    const char*         pattern,
    const char*         du_name,
    ucdbScopeMaskTypeT   root_mask,
    ucdbScopeMaskTypeT   scope_mask,
    ucdbScopeMaskTypeT   cover_mask,
    ucdb_CBFuncT        cbfunc,
```

db

Database.

pattern

Name pattern. Current wildcard symbols:

- * — matches any substring within a level of hierarchy
- ? — preceding character is optional
- [int:int] — matches any integer index in range
- {int | *} to {int | *} — matches any integer index in range
- {int | *} downto {int | *} — matches any integer index in range

To match wildcard characters literally, use the appropriate escaped identifier syntax.

du_name

Design unit name. Name is specified in the form:

library.primary(secondary)

where *secondary* matches for VHDL only. Multiple matches are possible if *library* or *secondary* is absent (even for Verilog design units, if the simulator created an artificial secondary).

root_mask

If set, matches start from a root that satisfies 1 bit of this mask

scope_mask

Only match scopes that satisfy 1 bit of this mask

cover_mask

Only match coveritems that satisfy 1 bit of this mask

cbfunc

User-supplied callback function

userdata	User-supplied function data

In-memory mode only. Matches the specified name pattern for any name in the entire instance tree or within specified design units. Recursively searches the subtree and generates callbacks for all named objects matching the pattern. Returns number (0 or more) of matches, or -1 if error.

Coverage and Statistics Summaries

The summary coverage statistics interface enables quick access to aggregated coverage and statistics for different kinds of coverage, and some overall statistics for the database.

Summary Coverage Data Type

Summary data type (ucdbSummaryEnumT) has the following nomenclature conventions:

- *_DU

  Coverage numbers that accumulate per-design-unit aggregations. Coverage from all instances of a design unit are merged into, and stored with the design unit itself. The summaries are then computed by traversing design units (not design instances).

- *_INST

  Values that accumulate all results from the entire instance tree. Design instances (not design units) are traversed. UCDB_CVG_INST coverage refers to covergroup instances, not design instances, which is coverage for exactly those covergroup objects that have option.per_instance set to 1 in the SystemVerilog source (weighted by option.weight). If no such covergroup objects exist, UCDB_CVG_INST coverage is 0.

```
/* For backward compatibility in enum literal names. */
#define UCDB_EXPR_INST   UCDB_UDP_EXPR_INST
#define UCDB_EXPR_DU     UCDB_UDP_EXPR_DU
#define UCDB_COND_INST   UCDB_UDP_COND_INST
#define UCDB_COND_DU     UCDB_UDP_COND_DU
typedef enum {
   UCDB_CVG_TYPE,     /* 0 Covergroup type coverage ==
$get_coverage()
                           value */
   UCDB_CVG_INST,     /* 1 Covergroup instances
(option.per_instance==1) ,
                           if any, weighted average */
   UCDB_COVER_INST,   /* 2 Cover directive, weighted average, per
design
                           instance */
   UCDB_SC_INST,      /* 3 SystemC functional coverage, per design
                           instance */
   UCDB_ZIN_INST,     /* 4 0-In checkerware coverage, per design
                           instance */
   UCDB_STMT_INST,    /* 5 statement coverage, per design instance
*/
   UCDB_STMT_DU,      /* 6 statement coverage, per design unit */
   UCDB_BRANCH_INST,  /* 7 branch coverage, per design instance */
   UCDB_BRANCH_DU,    /* 8 branch coverage, per design unit */
   UCDB_UDP_EXPR_INST,/* 9 UDP expression coverage, per design
instance */
   UCDB_UDP_EXPR_DU,  /* 10 UDP expression coverage, per design unit
*/
   UCDB_UDP COND_INST,/* 11 UDP condition coverage, per design
instance */
   UCDB_UDP_COND_DU,  /* 12 UDP condition coverage,per design unit
*/
   UCDB_TOGGLE_INST,  /* 13 toggle coverage, per design instance */
   UCDB_TOGGLE_DU,    /* 14 toggle coverage, per design unit */
   UCDB_FSM_ST_INST,  /* 15 FSM state coverage, per design instance
*/
   UCDB_FSM_ST_DU,    /* 16 FSM state coverage, per design unit */
   UCDB_FSM_TR_INST,  /* 17 FSM transition coverage, per design
instance */
   UCDB_FSM_TR_DU,    /* 18 FSM transition coverage, per design unit
*/
   UCDB_USER_INST,    /* 19 user-defined coverage, per design
instance */
   UCDB_ASSERT_PASS_INST,     /* 20 Assertion directive passes, per
design
                                   instance */
   UCDB_ASSERT_FAIL_INST,     /* 21 Assertion directive failures,
per
                                   design instance */
   UCDB_ASSERT_VPASS_INST,    /* 22 Assertion directive vacuous
passes,
                                   per design instance */
   UCDB_ASSERT_DISABLED_INST, /* 23 Assertion directive disabled,
per
                                   design instance */
   UCDB_ASSERT_ATTEMPTED_INST,/* 24 Assertion directive attempted,
per
                                   design instance */
   UCDB_ASSERT_ACTIVE_INST,   /* 25 Assertion directive active, per
                                   design instance */
   UCDB_CVP_INST,             /* 26 Coverpoint/cross weighted
average, all
                                   coverpoint and cross
declarations */
   UCDB_DIRECTED_TESTS,       /* 27 Reserved */
   UCDB_FEC_EXPR_INST,       /* 28 Focused expression coverage, per
                                   design instance */
   UCDB_FEC_EXPR_DU,         /* 29 Focused expression coverage, per
                                   design unit */
   UCDB_FEC_COND_INST,       /* 30 Focused condition coverage, per
                                   design instance */
   UCDB_FEC_COND_DU,         /* 31 Focused condition coverage, per
                                   design unit */
   UCDB_ASSERT_SUCCESS_INST,  /* 32 Assertion directives that
succeeded:
                                 never failed, passed at least once
(if
                                    pass counts available.) */
    UCDB_EXPRESSION_INST,     /* 33 Expression coverage, per design
                                    instance */
    UCDB_EXPRESSION_DU,  /* 34 Expression coverage, per design unit
*/
    UCDB_CONDITION_INST, /* 35 Condition coverage, per design inst
*/
    UCDB_CONDITION_DU,  /* 36 Condition coverage,per design unit */
    UCDB_FSM_INST,       /* 37 FSM state coverage, per design
instance */
    UCDB_FSM_DU,       /* 38 FSM state coverage, per design unit */
    UCDB_TP_COVERAGE    /* 39 Testplan coverage for merged files
                              with testplans */
    UCDB_N_SUMMARY_ENUM_T /* 40 Can be used for array bounds */
} ucdbSummaryEnumT;
```

Coverage Structure

Stores values for a particular enumerator.

```
typedef struct {
   double  coverage_pct;   /* floating point coverage value, percentage */
   double  goal_pct;        /* floating point goal, percentage */
   int     num_coveritems;  /* total number of coveritems (bins) */
   int     num_covered;     /* number of coveritems (bins) covered */
} ucdbCoverageT;
```

Table 4-1. Values for num_coveritems Dependent on Coverage Type

Enumerator	Type	Number
CVG*	SV covergroup	bins
COVER	SVA or PSL cover	cover directives or statements
STMT*	statement	statements
BRANCH*	branch	branches (including implicit elses)

Table 4-1. Values for num_coveritems Dependent on Coverage Type (cont.)

Enumerator	Type	Number
EXPR*	expression	known-value truth table rows
COND*	condition	known-value truth table rows
TOGGLE*	toggle	toggles (scopes in UCDB)
FSM_ST*	FSM state	FSM states
FSM_TR*	FSM transition	FSM transitions
ASSERT*	SVA or PSL assert	assert directives or statements This value is almost always the number of coveritems covered, except for ASSERT_PASS* (number of assertion passes) and ASSERT_FAIL* (number of assertion failures).
BLOCK*	Block	blocks

Coverage Summary Structure

Stores all statistics returned by ucdb_GetCoverageSummary().

```
typedef enum {
   /* Bit 0 set implies "merge -totals" file          */
   /* Bit 1 set implies "merge -testassociated" file */
   UCDB_SUMMARY_FLAG_none = 0,
   UCDB_SUMMARY_FLAG_is_merge_totals = 1,
   UCDB_SUMMARY_FLAG_is_merge_testassociated = 2,
   UCDB_SUMMARY_FLAG_is_merge = 3
} ucdbSummaryFlagsEnumT;
typedef struct {
   int                    num_instances;   /* number of design instances */
   int                   num_coverpoints; /* number of SV coverpoint and*/
                                          /* cross types                */
   int                  num_covergroups; /* number of SV covergroup types*/
   int                   num_dus;         /* number of design units     */
   ucdbSummaryFlagsEnumT flags;
   ucdbCoverageT          coverage[UCDB_N_SUMMARY_ENUM_T];
} ucdbCoverageSummaryT;
```

Memory Statistics Types

Memory statistics are summary statistics for simulator memory usage. For merged data, the merged output is the maximum of the merged inputs.

The following type is an enumerator for the category of statistics merged.

```
typedef enum {
   UCDB_MEMSTATS_COVERGROUP,    /* covergroup             */
   UCDB_MEMSTATS_ASSERT,        /* assertion              */
   UCDB_MEMSTATS_CONSTRAINT,    /* constraint solver      */
   UCDB_MEMSTATS_CLASS,         /* classes                */
   UCDB_MEMSTATS_DYNAMIC,       /* dynamic objects        */
   UCDB_MEMSTATS_OTHER,         /* other categories       */
   UCDB_MEMSTATS_ENDCATEGORY    /* marker past last value */
} ucdbMemStatsEnumT;
```

The following type is an enumerator for the type of statistic.

```
typedef enum {
   UCDB_MEMSTATS_MAXMEM,        /* All categories: maximum memory usage
                                   high water mark) -- bytes            */
   UCDB_MEMSTATS_PEAKTIME,      /* All categories: peak memory time     */
   UCDB_MEMSTATS_CURRMEM,       /* All categories: current memory
                                   usage (in bytes) at time of saving
                                   the UCDB file                        */
   UCDB_MEMSTATS_NUMOBJECTS,    /* All categories: number of objects    */
   UCDB_MEMSTATS_ENDTYPE        /* marker past last value               */
} ucdbMemStatsTypeEnumT;
```

ucdb_SetGoal

```
int ucdb_SetGoal(
    ucdbT               db,
    ucdbSummaryEnumT    type,
```

db Database

type Summary coverage type

percentage Goal to set for the coverage type

Aggregated coverage is compared to this percentage to determine whether the goal is satisfied.

Sets the goal percentage for the specified type of aggregated coverage. Returns 0 if successful, or non-zero if error.

ucdb_GetGoal

```
float ucdb_GetGoal(
    ucdbT               db,
```

db Database

type Summary coverage type

Returns the goal for the specified type of aggregated coverage. The goal is a percentage, 0.0 to 100.0. Returns non-negative goal value if successful, or -1.0 if error.

ucdb_SetWeightPerType

```
int ucdb_SetWeightPerType(
    ucdbT                 db,
    ucdbSummaryEnumT      type,
```

db	Database.
type	Summary coverage type
weight	Weight to set for the coverage type Weights are non-negative integers, used to compute total coverage numbers as in ucdb_GetTotalCoverage

Sets the weight for the specified type of aggregated coverage. Returns 0 if successful, or non-zero if error.

ucdb_GetWeightPerType

```
int ucdb_GetWeightPerType(
    ucdbT                 db,
```

db	Database.
type	Summary coverage type

Returns the weight for the specified type of aggregated coverage. Returns non-negative goal value if successful, or -1.0 if error.

ucdb_GetCoverageSummary

```
int ucdb_GetCoverageSummary(

const char*              name,
ucdbCoverageSummaryT* data);
```

name	File system path.
data	Coverage summary returned

Gets coverage summary statistics. The specified file is opened, seeked to the location of previously computed summary statistics, and immediately closed. See “Databases and Database Files” on page 121for the “efficient” read option. Returns 0 if successful, or non-zero if error.

ucdb_GetCoverage

```
float ucdb_GetCoverage(
    ucdbT                 db,
    ucdbSummaryEnumT      type,
    int*                  num_covered_bins,
```

db	Database
type	Summary coverage type
num_covered_bins	Number of covered bins for the coverage type, or NULL if not set
num_total_bins	Total number of bins for the coverage type, or NULL if not set

Returns the aggregated coverage of the specified type. The returned value might not equal the following for cases where coveritems can be weighted differently and for SystemVerilog covergroups (for which coverage is not only weighted but is calculated hierarchically).:

```
num_covered_bins /  num_total_bins
```

A return value of -1.0 indicates the coverage is not applicable (that is, no coveritems of the implied type are in the database, so num_total_bins is 0). Other negative return values indicate error.

Note

If any significant data has changed since the last call, this ucdb_GetCoverage call forces an expensive recalculation using the entire database. The aggregated coverage is automatically recalculated with ucdb_Close, if necessary. However, if no significant data changes were made since the file was opened or the last call to ucdb_GetCoverage, this ucdb_GetCoverage call remains an efficient operation; it is maintained as summary data in the database, for fast retrieval.

ucdb_GetStatistics

```
int ucdb_GetStatistics(
    ucdbT                   db,
    int*                    num_covergroups,
    int*                    num_coverpoints,
    int*                    num_instances,
```

db	Database
num_covergroups	Number of covergroup types
num_coverpoints	Number of covergroup coverpoints
num_instances	Number of design instances
num_dus	Number of design units

Gets overall statistics for the database. Returns 0 if successful, or non-zero if error.

Note

If any significant data has changed since the last call, this ucdb_GetStatistics call forces an expensive recalculation using the entire database. The statistics are automatically recalculated with ucdb_Close, if necessary. However, if no significant data changes were made since the last call to ucdb_GetStatistics, this call remains an efficient operation it is maintained as summary data in the database, for fast retrieval.

ucdb_CalcCoverageSummary

```
int ucdb_CalcCoverageSummary(
    ucdbT                      db,
    ucdbScopeT                 scope,
    int                        recurse_instances,
    ucdbCoverageSummaryT*      data,
```

db

Database

scope

Scope

Entire database if NULL.

recurse_instances

Recursion instances flag.

- For non-testplan scopes, this flag causes a recursion into subscopes of types matching the mask UCDB_HDL_INST_SCOPE.
- For testplan scopes, this causes recursion into scopes of type UCDB_TESTPLAN. One type of recursion always occurs with testplan scopes: following non-testplan scopes that share a tag with the "scope" given to this routine.

data

Coverage summary data

test mask

Optional test mask

If set, the database must have been created with all coveritems containing a cover test mask (that is, as a result of running a "test-associated merge"). Only coveritems matching the test mask are considered covered in the calculation, which is prone to some error and can be improved with additional data in the future. Setting test_mask to NULL will calculate coverage based on current bin values only.

In-memory mode only calculates coverage summary statistics, the same data as provided in the argument table, on a subset of an opened database. When called on an instance, this function reports by-DU coverage only for the case where UCDB_INST_ONCE is set for the instance. Here, by-DU coverage and instance coverage are identical. When called on the entire database, coverage from all DUs and all instances are counted.

Note

If called with a NULL scope and NULL test_mask, this call can be made on an open database handle without fully populating the in-memory data image. See “Databases and Database Files” on page 121.

ucdb_GetTotalCoverage

```
int ucdb_GetTotalCoverage(
    ucdbT                        db,
    ucdbObjT                     obj,
    float*                       total_coverage,
```

db

Database

obj

Object type (ucdbScopeT or ucdbTestT)

All roots if NULL.

total_coverage

Total coverage

- For a coverage scope, this is the total coverage calculated in a way similar to ucdb_CalcCoverageSummary().
- For a design instance, this is the weighted average of coverage per type, for all types found in the design subtree rooted at that instance. This coverage uses weights as set from ucdb_SetWeightPerType() and retrieved by ucdb_GetWeightPerType().
- For a leaf testplan scope, coverage is the weighted average of all design instance or coverage scopes sharing the same tag.
- For a non-leaf testplan scope, coverage is the weighted average of coverage of all children. If the non-leaf testplan scope shares a tag with design or coverage scopes, those collectively are equally weighted as one child testplan instance, as if a virtual child testplan scope shared a tag with all the other design and coverage scopes.
- Test data records with status attribute values UCDB_TESTSTATUS_OK and UCDB_TESTSTATUS_WARNING count as 100%; other test data records count as 0%.

Assertion results are included in the form of “% non-vacuously passed,” which is the percentage of assertions that non-vacuously passed at least once (that is, non-zero non-vacuous pass count).

test mask	Optional test mask. If set, the database must have been created with all coveritems containing a cover test mask (that is, as a result of running a "test-associated merge"). Only coveritems matching the test mask are considered covered in the calculation, which is prone to some error and can be improved with additional data in the future. Setting test_mask to NULL will calculate coverage based on current bin values only.

This calculates a single coverage number (as a percentage, 0.0 - 100.0) for a scope in the database. Returns 1 if the scope had any coverage data. Returns 0 if none were found and sets total_coverage to -1.0. Returns -1 if error.

ucdb_GetMemoryStats

```
int ucdb_GetMemoryStats(
    ucdbT                      db,
    ucdbMemStatsEnumT          category,
    ucdbMemStatsTypeEnumT      type,
```

db	Database
category	Memory statistics category
type	Statistics type for the memory statistics category
value	Memory statistics value returned

Gets memory usage statistics for the specified statistics type for the specified statistics category. Returns 0 if successful, 1 if the statistic does not apply, or -1 if error.

ucdb_SetMemoryStats

```
int ucdb_SetMemoryStats(
    ucdbT                      db,
    ucdbMemStatsEnumT          category,
    ucdbMemStatsTypeEnumT      type,
```

db	Database
category	Memory statistics category
type	Statistics type for the memory statistics category
value	Memory statistics value to set

Sets memory usage statistics for the specified statistics type for the specified statistics category. Returns 0 if successful, or non-zero if error.

Coveritems

This section lists cover types, coveritem types, and flags for cover item data.

Cover Types

```
typedef unsigned int ucdbCoverTypeT;
/* Bits for ucdbCoverTypeT: */
#define UCDB_CVGBIN         UCIS_CVGBIN
                           /* For SV Covergroups                 */
#define UCDB_COVERBIN       UCIS_COVERBIN
                           /* For cover directives: pass         */
#define UCDB_ASSERTBIN      UCIS_ASSERTBIN
                           /* For assert directives: fail        */
#define UCDB_SCBIN          INT64_LITERAL(0x0000000000000008)
                           /* For SystemC transactions           */
#define UCDB_ZINBIN         INT64_LITERAL(0x0000000000000010)
                           /* For 0-in Checkerware               */
#define UCDB_STMTBIN        UCIS_STMTBIN
                           /* For Code coverage(Statement)       */
#define UCDB_BRANCHBIN      UCIS_BRANCHBIN
                           /* For Code coverage(Branch)          */
#define UCDB_EXPRBIN        UCIS_EXPRBIN
                           /* For Code coverage(Expression)      */
#define UCDB_CONDBIN        UCIS_CONDBIN
                           /* For Code coverage(Condition)       */
#define UCDB_TOGGLEBIN      UCIS_TOGGLEBIN
                           /* For Code coverage(Toggle)          */
#define UCDB_PASSBIN        UCIS_PASSBIN
                           /* For assert directives: pass count  */
#define UCDB_FSMBIN         UCIS_FSMBIN
                           /* For FSM coverage                   */
#define UCDB_USERBIN        UCIS_USERBIN
                           /* User-defined coverage              */
#define UCDB_GENERICBIN     UCDB_USERBIN
#define UCDB_COUNT          UCIS_COUNT
                           /* user-defined count, not in coverage*/
#define UCDB_FAILBIN        UCIS_FAILBIN
                           /* For cover directives: fail count   */
#define UCDB_VACUOUSBIN     UCIS_VACUOUSBIN
                           /* For assert: vacuous pass count     */
#define UCDB_DISABLEDBIN    UCIS_DISABLEDBIN
                           /* For assert: disabled count         */
#define UCDB_ATTEMPTBIN     UCIS_ATTEMPTBIN
                           /* For assert: attempt count          */
#define UCDB_ACTIVEBIN      UCIS_ACTIVEBIN
                           /* For assert: active thread count    */
#define UCDB_IGNOREBIN      UCIS_IGNOREBIN
                           /* For SV Covergroups                 */
#define UCDB_ILLEGALBIN     UCIS_ILLEGALBIN
                           /* For SV Covergroups                 */
#define UCDB_DEFAULTBIN     UCIS_DEFAULTBIN
                           /* For SV Covergroups                 */
#define UCDB_PEAKACTIVEBIN  UCIS_PEAKACTIVEBIN
                           /* For assert: max active thread count*/
#define UCDB_RESERVEDBIN     UCIS_RESERVEDBIN
                            /* Reserved                          */
```

Coveritem Types

```
#define UCDB_COVERGROUPBINS ((ucdbCoverMaskTypeT)\
   (UCDB_CVGBIN | UCDB_IGNOREBIN | UCDB_ILLEGALBIN | UCDB_DEFAULTBIN))
#define UCDB_FUNC_COV ((ucdbCoverMaskTypeT)\
   (UCDB_COVERGROUPBINS | UCDB_COVERBIN | UCDB_SCBIN))
#define UCDB_CODE_COV ((ucdbCoverMaskTypeT)\
   (UCDB_STMTBIN | UCDB_BRANCHBIN | UCDB_EXPRBIN | UCDB_CONDBIN \
   | UCDB_TOGGLEBIN | UCDB_FSMBIN))
#define UCDB_ASSERTIONBINS ((ucdbCoverMaskTypeT)\
   (UCDB_ASSERTBIN | UCDB_PASSBIN | UCDB_VACUOUSBIN | UCDB_DISABLEDBIN \
   | UCDB_ATTEMPTBIN | UCDB_ACTIVEBIN | UCDB_PEAKACTIVEBIN))
#define UCDB_NO_BINS  ((ucdbCoverMaskTypeT)INT64_ZERO)
#define UCDB_ALL_BINS ((ucdbCoverMaskTypeT)INT64_NEG1)
```

Coveritem Data Type

```
typedef ucisCoverDataValueT ucdbCoverDataValueT;
typedef ucisCoverDataT ucdbCoverDataT;
```

Flags for Coveritem Data

```
#define UCDB_IS_32BIT       UCIS_IS_32BIT  /* data is 32 bits           */
#define UCDB_IS_64BIT       UCIS_IS_64BIT  /* data is 64 bits           */
#define UCDB_IS_VECTOR      UCIS_IS_VECTOR /* data is actually a vector */
#define UCDB_HAS_GOAL       UCIS_HAS_GOAL  /* goal included             */
#define UCDB_HAS_WEIGHT     UCIS_HAS_WEIGHT  /* weight included         */
#define UCDB_EXCLUDE_PRAGMA  UCIS_EXCLUDE_PRAGMA  /* excluded by pragma */
#define UCDB_EXCLUDE_FILE    UCIS_EXCLUDE_FILE  /* excluded by file;
                                       does not count in total coverage */
#define UCDB_LOG_ON          UCIS_LOG_ON  /* for cover/assert directives;
                                          controls simulator output    */
#define UCDB_ENABLED         UCIS_ENABLED  /* generic enabled flag; if
                                          disabled, still counts in total
                                          coverage                     */
#define UCDB_HAS_LIMIT      UCIS_HAS_LIMIT  /* for limiting counts      */
#define UCDB_HAS_ACTION      UCIS_HAS_ACTION  /* for assert directives,
                                       refer to "ACTION" in attributes  */
#define UCDB_IS_FSM_RESET    UCIS_IS_FSM_RESET  /* For fsm reset states */
#define UCDB_IS_TLW_ENABLED  UCIS_IS_TLW_ENABLED/* for assert directives*/
#define UCDB_IS_FSM_TRAN     UCIS_IS_FSM_TRAN  /* for FSM coveritems, is a
                                             transition bin            */
#define UCDB_IS_BR_ELSE      UCIS_IS_BR_ELSE  /* for branch ELSE
                                             coveritems   */
#define UCDB_CLEAR_PRAGMA    UCIS_CLEAR_PRAGMA
#define UCDB_IS_EOS_NOTE     UCIS_IS_EOS_NOTE  /* for directives active at
                                             end of simulation         */
#define UCDB_EXCLUDE_INST    UCIS_EXCLUDE_INST  /* for instance-specific
                                           exclusions                  */
#define UCDB_EXCLUDE_AUTO    UCIS_EXCLUDE_AUTO /* for automatic
                                          exclusions                   */
#define UCDB_IS_CROSSAUTO    UCIS_IS_CROSSAUTO  /* covergroup auto cross
                                          bin */
#define UCDB_COVERFLAG_MARK  UCIS_COVERFLAG_MARK  /* flag for temporary
                                          mark    */
#define UCDB_USERFLAGS      UCIS_USERFLAGS  /* reserved for user flags */
#define UCDB_FLAG_MASK       UCIS_FLAG_MASK
#define UCDB_EXCLUDED         ( UCDB_EXCLUDE_FILE | UCDB_EXCLUDE_PRAGMA \
                              | UCDB_EXCLUDE_INST | UCDB_EXCLUDE_AUTO)
```

ucdb_CreateNextCover

```
int ucdb_CreateNextCover(
    ucdbT                db,
    ucdbScopeT           parent,
    const char*          name,
    ucdbCoverDataT*      data,
```

db	Database
parent	Scope in which to create the coveritem
name	Name to give the coveritem. Can be NULL
data	Associated data for coverage
sourceinfo	Associated source information

Creates the next coveritem in the given scope. Returns the index number of the created coveritem, -1 if error.

```
    int ucdb_CloneCover(
        ucdbT                   targetdb,
        ucdbScopeT              targetparent,
        ucdbT                   sourcedb,
        ucdbScopeT              sourceparent,
        int                     coverindex,
```

targetdb	Database context for clone
targetparent	Parent scope of clone
db	Source database
parent	Source scope
coverindex	Source coverindex
cloneflags	UCDB_CLONE_ATTRS or 0

Has no effect when targetdb is in streaming mode. Creates a copy of the specified coveritem in the specified scope (targetparent). Predefined attributes are created by default. Returns the coverindex if successful, or -1 if error.

ucdb_RemoveCover

```
    int ucdb_RemoveCover(
        ucdbT                   db,
        ucdbScopeT              parent,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem to remove

Has no effect when db is in streaming mode. Removes the specified coveritem from its parent. Returns 0 if successful, or -1 if error. Coveritems cannot be removed from scopes of type UCDB_ASSERT (instead, remove the whole scope). Similarly, coveritems from scopes of type UCDB_TOGGLE with toggle kind UCDB_TOGGLE_SCALAR, UCDB_TOGGLE_SCALAR_EXT, UCDB_TOGGLE_REG_SCALAR, or UCDB_TOGGLE_REG_SCALAR_EXT cannot be removed (instead, remove the whole scope).

```
    int ucdb_MatchCoverInScope(
        ucdbT                   db,
        ucdbScopeT              parent,
```

db	Database

parent	Parent scope of coveritem
name	Coveritem name to match

Gets covereitem from database if it exists in the specified scope. Returns coveritem index, or -1 if error.

ucdb_MatchCoverInScope

```
int ucdb_MatchCoverInScope(
ucdbT db,
ucdbScopeT parent,
const char* name);
```

db	Database
parent	Parent scope of coveritem
name	Coveritem name to match

ucdb_IncrementCover

```
intucdb_IncrementCover(
ucdbT db,
ucdbScopeT parent,
int coverindex,
int64_t increment);
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
increment	Increment count to add to current count

Increments the data count for the coveritem, if not a vector item. Returns 0 if successful, or -1 if error.

ucdb_GetCoverFlags

```
ucdb_FlagsT ucdb_GetCoverFlags(
    ucdbT               db,
    ucdbScopeT          parent,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope

Returns the flags for the specified coveritem, or NULL if error.

ucdb_GetCoverFlag

```
int ucdb_GetCoverFlag(
    ucdbT                   db,
    ucdbScopeT              parent,
    int                     coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
mask	Flag mask to match

Returns 1 if coveritem's flag bit matches the specified mask, 0 if the coveritem has flag bits not matching the specified mask, or -1 if the coveritem does not have any flag bits.

ucdb_SetCoverFlag

```
void ucdb_SetCoverFlag(
    ucdbT                   db,
    ucdbScopeT              parent,
    int                     coverindex,
    ucdbFlagsT              mask,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
mask	Flag mask
bitvalue	Value to set: 0 or 1

Sets bits in the coveritem's flag field with respect to the given mask.

ucdb_GetCoverType

```
ucdbCoverTypeT ucdb_GetCoverType(
    ucdbT                   db,
    ucdbScopeT              parent,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope

Returns the cover type of the specified coveritem. or 0 if error.

ucdb_GetCoverData

```
int ucdb_GetCoverData(
    ucdbT                 db,
    ucdbScopeT            parent,
    int                   coverindex,
    char**                name,
    ucdbCoverDataT*       data,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
name	Name returned (failbin, passbin, vacuousbin, disabledbin, attemptbin, activebin or peakactivebin)
data	Data returned
sourceinfo	Source information returned

Gets name, data and source information for the specified coveritem. Returns 0 if successful, or non-zero if error. You must save the returned data, as the next call to this function can invalidate the returned data. Any of the data arguments can be NULL (that is, that data is not retrieved).

ucdb_SetCoverData

```
int ucdb_SetCoverData(
    ucdbT                 db,
    ucdbScopeT            parent,
    int                   coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
data	Data to set

Sets data for the specified coveritem. Returns 0 if successful, or non-zero if error. You must ensure the data fields are valid.

ucdb_SetCoverCount

```
int ucdb_SetCoverCount(
    ucdbT                 db,
    ucdbScopeT            parent,
    int                   coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
count	Cover count value to set

Sets the count for the specified coveritem. Returns 0 if successful, or non-zero if error.

ucdb_SetCoverGoal

```
int ucdb_SetCoverGoal(
    ucdbT                   db,
    ucdbScopeT              parent,
    int                     coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
goal	Cover goal value to set

Sets the goal for the specified coveritem. Returns 0 if successful, or non-zero if error.

ucdb_SetCoverLimit

```
int ucdb_SetCoverLimit(
    ucdbT                   db,
    ucdbScopeT              parent,
    int                     coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
limit	Cover limit value to set

Sets the limit for the specified coveritem. Returns 0 if successful, or non-zero if error.

ucdb_SetCoverWeight

```
int ucdb_SetCoverWeight(
    ucdbT                   db,
    ucdbScopeT              parent,
    int                     coverindex,
```

db	Database

parent	Parent scope of coveritem
coverindex	Coverindex of coveritem in parent scope
weight	Cover weight value to set

Sets the weight for the specified coveritem. Returns 0 if successful, or non-zero if error.

ucdb_GetScopeNumCovers

```
int ucdb_GetScopeNumCovers(
    ucdbT                db,
```

db	Database
scope	Scope

Returns the number of coveritems in the specified scope (which can be 0), or -1 if error.

ucdb_GetECCoverNumHeaders

```
int ucdb_GetECCoverNumHeaders(
    ucdbT                db,
```

db	Database
scope	Scope

Returns the number of UDP header columns for Expression and Condition coverage in the specified scope (which can be 0), or -1 if error. For example, to get all the header columns:

```
num_columns = ucdb_GetECCoverNumHeaders(db, cvitem);
for (i = 0; i < num_columns; i++) {
   char* header;
   status = ucdb_GetECCoverHeader(db, cvitem, i, &header);
}
```

ucdb_GetECCoverHeader

```
int ucdb_GetECCoverHeader(
    ucdbT                db,
    ucdbScopeT           scope,
    int                  index,
```

db	Database
scope	Scope
index	Index
header	Header string returned

Gets the indexed UDP header string of Expression and Condition coverage. Returns 0 if successful, or 1 if error.

ucdb_NextCoverInScope

```
int ucdb_NextCoverInScope(
    ucdbT                   db,
    ucdbScopeT              parent,
    int*                    coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Index of coveritem in parent
covermask	Mask for type of coveritem

Given a coveritem and cover type mask, gets the next coveritem from the scope. Start with a coverindex == -1 to return the first coveritem in the scope. Returns 0 at end of traversal, -1 if error.

ucdb_NextCoverInDB

```
int ucdb_NextCoverInDB(
    ucdbT                   db,
    ucdbScopeT              parent,
    int*                    coverindex,
```

db	Database
parent	Parent scope of coveritem
coverindex	Index of coveritem in parent
covermask	Mask for type of coveritem

Given a coveritem and cover type mask, gets the next coveritem from the scope. Start with a coverindex == -1 and parent == NULL to return the first coveritem in the database. Returns 0 at end of traversal, -1 if error.

Toggles

Toggles are the most common type of object in a typical code coverage database. Therefore, they have a specific interface in the API that can be restricted for optimization purposes. Net toggles can be duplicated throughout the database through port connections. They can be reported once rather than in as many different local scopes as they appear (this requires a net id).

```
typedef enum {
    UCDB_TOGGLE_ENUM,           /* Enum type object */
    UCDB_TOGGLE_INT,            /* Integer type object */
    UCDB_TOGGLE_REG_SCALAR=4,   /* Scalar, one bit reg */
    UCDB_TOGGLE_REG_SCALAR_EXT, /* Extended toggle of scalar reg */
    UCDB_TOGGLE_SCALAR,         /* Scalar net or std_logic_bit */
    UCDB_TOGGLE_SCALAR_EXT      /* Ext toggle of scalar net or
                                    std_logic_bit */
    UCDB_TOGGLE_REAL            /* Real type object */
} ucdbToggleTypeT;
typedef enum {
    UCDB_TOGGLE_INTERNAL,       /* non-port: internal wire or variable */
    UCDB_TOGGLE_IN,             /* input port */
    UCDB_TOGGLE_OUT,            /* output port */
    UCDB_TOGGLE_INOUT           /* inout port */
} ucdbToggleDirT;
```

ucdb_CreateToggle

```
ucdbScopeT ucdb_CreateToggle(
    ucdbT                   db,
    ucdbScopeT              parent,
    const char*             name,
    const char*             canonical_name,
    ucdbFlagsT              flags,
    ucdbToggleTypeT         toggle_type,
```

db	Database
parent	Scope in which to create the toggle
name	Name to give the toggle object
canonical_name	Canonical name for the toggle object Identifies unique toggles. Toggles with the same canonical_name must count once when traversed for a report or coverage summary.
flags	Exclusion flags
toggle_type	Toggle type
toggle_dir	Toggle direction

Creates the specified toggle scope beneath the given parent scope. Returns a handle to the created scope (type UCDB_TOGGLE), or NULL if error.

ucdb_GetToggleInfo

```
int ucdb_GetToggleInfo(
    ucdbT                   db,
    ucdbScopeT              toggle,
    const char**            canonical_name,
    ucdbToggleTypeT*        toggle_type,
```

db	Database
toggle	Toggle scope containing the information
canonical_name	Canonical name for the toggle object May be NULL for unconnected nets, enum, int, and reg type toggles. Memory for canonical_name is allocated by the system and must not be de-allocated by the user.
toggle_type	Toggle type
toggle_dir	Toggle direction

Returns toggle-specific information associated with the specified toggle scope. Returns 0 if successful, -1 if error.

ucdb_GetToggleCovered

```
int ucdb_GetToggleCovered(
    ucdbT                   db,
```

db	Database
toggle	Toggle scope containing the information

Returns 1 if toggle is covered, 0 if toggle is uncovered and -1 if an error.

ucdb_GetBCoverInfo

```
int ucdb_GetBCoverInfo(
    ucdbT                   db,
    ucdbScopeT              coveritem,
    int*                    has_else,
    int*                    iscase,
```

db	Database
coveritem	Coveritem
has_else	1 if branch has else clause; 0 otherwise
iscase	1 if branch is a CASE statement; 0 otherwise
num_elmts	Number of elements in branch. 1 if a CASE branch

Returns 1 if branch is a CASE statement; 0 otherwise (IF statement).

Groups

Groups are used to maintain bus structures in the database. They provide additional support for part-select toggle nodes, particularly with the support for wildcard ranges provided by group scopes.

Group Kind Type

```
#define UCDB_GROUP_MASK_PACKED  0x1000
#define UCDB_GROUP_MASK_ORDERED 0x2000
typedef enum {
   UCDB_GROUP_BASIC            = 0x0001,
   UCDB_GROUP_UNPACKED_STRUCT = 0x0002,
   UCDB_GROUP_UNPACKED_UNION  = 0x0003,
   UCDB_GROUP_UNPACKED_ARRAY  = (0x0004|UCDB_GROUP_MASK_ORDERED),
   UCDB_GROUP_ASSOC_ARRAY     = 0x0005,
   UCDB_GROUP_PACKED_STRUCT   =
      (UCDB_GROUP_UNPACKED_STRUCT|UCDB_GROUP_MASK_PACKED),
   UCDB_GROUP_PACKED_UNION    =
      (UCDB_GROUP_UNPACKED_UNION|UCDB_GROUP_MASK_PACKED),
   UCDB_GROUP_PACKED_ARRAY    =
      (UCDB_GROUP_UNPACKED_ARRAY|UCDB_GROUP_MASK_PACKED)
} ucdbGroupKind;
```

Wildcard Matching

General wildcard matching supports the following wildcard symbols:

* Matches one or more characters. Only spans one scope, so * matches [2], but does not match [2][4].

? Matches a single character.

The following range pattern searches require group scopes:

- (number)
- [number]
- [number:number]
- (number to number)
- (number downto number)

ucdb_CreateGroupScope

```
ucdbScopeT ucdb_CreateGroupScope(
    ucdbT               db,
    ucdbScopeT          parent,
    ucdbGroupKind       kind,
    char*               name,
    ucdbFlagsT          flags,
    int                 numberOfRangePairs,
```

db	Database
parent	Parent scope
kind	Group kind
name	Name to assign to the group scope
flags	Flags
numberOfRangePairs	Number of range pairs Only used for ordered groups.
rangePairs	Range pairs Only used for ordered groups.

Creates the specified group scope beneath the parent scope. Returns the scope handle if successful, or NULL if error. In write-streaming mode, name and rangePairs are not copied, so they should be kept unchanged until the next ucdb_WriteStream* call or the next ucdb_Create* call.

ucdb_GetGroupInfo

```
int ucdb_GetGroupInfo(
    ucdbT               db,
    ucdbScopeT          group,
    ucdbGroupKind*      kind,
    const char**        name,
    int*                numberOfRangePairs,
```

db	Database
group	Group scope
kind	Group kind
name	Name of the group scope
numberOfRangePairs	Number of range pairs Only used for ordered groups.
rangePairs	Range pairs Only used for ordered groups.

Gets the group-specific information (kind, name, numberOfRangePairs, and rangePairs) for the specified group scope. Returns 0 if successful, or -1 if error.

ucdb_ExpandOrderedGroupRangeList

```
    int ucdb_ExpandOrderedGroupRangeList(
        ucdbT               db,
        ucdbScopeT          group,
        int                 numberOfRangePairs,
```

db	Database
group	Group scope Must be UCDB_GROUP_PACKED_ARRAY or UCDB_GROUP_UNPACKED_ARRAY type.
numberOfRangePairs	Number of range pairs
rangePairs	Range pairs

Expands the range pairs for the specified group with the specified list of range pairs according to the following rules:

- A range that does not overlap an existing range is added to the range list.
- A range that encloses one or more existing ranges replaces the enclosed ranges.
- A range that (partially) overlaps an existing range expands that range.
- A range completely enclosed in an existing range is ignored.

Returns 0 if successful, -1 if error.

ucdb_GetOrderedGroupElementByIndex

```
    ucdbScopeT ucdb_GetOrderedGroupElementByIndex(
        ucdbT               db,
        ucdbScopeT          parent,
```

db	Database
group	Parent ordered group scope Must be UCDB_GROUP_PACKED_ARRAY or UCDB_GROUP_UNPACKED_ARRAY type.
index	Index of the child

Returns the handle of the child element of the specified ordered group scope that has the specified index, or NULL if error or if no element corresponds to the index. For example, for the ordered group corresponding to bus[3:0]:

- index = 1 returns the rightmost range number (0)
- index = 4 returns the leftmost range number (3)

Function is used in memory mode only.

Tags

A tag is a group of strings associated with a scope. Scopes can have associated tags for grouping; when items share a tag they are associated together. In particular, when UCDB_TESTPLAN scopes share tags with coverage scopes that contain coveritems, the association can be used to do traceability analysis tests.

The following example traverses all non-testplan scopes that share a tag with a given testplan scope:

```
if (  ucdb_ObjKind(db,obj)==UCDB_OBJ_SCOPE &&
      ucdb_GetScopeType(db,(ucdbScopeT)obj)==UCDB_TESTPLAN ) {
   int t, numtags = ucdb_GetScopeNumTags(db,scope);
   const char* tagname;
   for ( t=0; t<numtags; t++ ) {
      int found;
      ucdbObjT taggedobj;
      ucdb_GetScopeIthTag(db,scope,t,&tagname);
       for ( found=ucdb_BeginTaggedObj(db,tagname,&taggedobj);
                 found; found=ucdb_NextTaggedObj(db,&taggedobj) ) {
         if (  ucdb_ObjKind(db,taggedobj)==UCDB_OBJ_SCOPE &&
               ucdb_GetScopeType(db,(ucdbScopeT)taggedobj)==UCDB_TESTPLAN
         ) continue;
         /* Now taggedobj is a non-testplan obj sharing a tag with */
         /* obj -- put your code here                              */
      }
   }
}
```

Here is an example of traversing all scopes for all tags in a UCDB file:

```
ucdbT db = ucdb_Open(filename);
const char* tagname = NULL;
while (tagname = ucdb_NextTag(db,tagname)) {
   int found;
   ucdbScopeT scope;
   for ( found=ucdb_BeginTagged(db,tagname,&scope);
         found;
         found=ucdb_NextTagged(db,&scope) ) {
         /* Put your code here */
   }
}
```

Note

This traversal cannot nest. Code inside this loop cannot reuse the BeginTagged/NextTagged functions.

Object Mask Type

```
typedef enum {
    UCDB_OBJ_ERROR = 0,     /* Start of the db, apply initial settings */
    UCDB_OBJ_TESTDATA = 1,  /* Testdata object                         */
    UCDB_OBJ_SCOPE = 2,     /* Scope object                            */
    UCDB_OBJ_COVER = 4,     /* Cover object                            */
    UCDB_OBJ_ANY = -1       /* ucdbScopeT or ucdbHistoryNodeT          */
} ucdbObjMaskT;
```

Enum type for different object types. This is a bit mask for the different types of objects that are tagged. Mask values can be ANDed and ORed together.

ucdb_ObjKind

```
ucdbObjMaskT ucdb_ObjKind(
    ucdbT               db,
```

db	Database
obj	Obj

Returns object type (ucdbScopeT or ucdbTestT) for the specified object, or UCDB_OBJ_ERROR if error.

ucdb_GetObjType

```
ucdbObjTypeT ucdb_GetObjType(
    ucdbT               db,
```

db	Database
object	Object

Polymorphic function (aliased to ucdb_GetHistoryKind) for acquiring an object type. Returns UCDB_HISTORYNODE_TEST (object is a test data record), UCDB_HISTORYNODE_TESTPLAN (object is a testplan record), UCDB_HISTORYNODE_MERGE (object is a merge record), scope type ucdbScopeTypeT (object is not of these), or UCDB_SCOPE_ERROR if error. This function can return a value with multiple bits set (for history data objects). Return value *must not be used* as a mask.

ucdb_AddObjTag

```
int ucdb_AddObjTag(
    ucdbT               db,
    ucdbObjT            obj,
```

db	Database
obj	Object (ucdbScopeT or ucdbTestT)
tag	Tag

Adds a tag to a given object. Returns 0 if successful, or non-zero if error. Error includes null tag or tag with “\n” character.

ucdb_RemoveObjTag

```
int ucdb_RemoveObjTag(
    ucdbT               db,
    ucdbObjT            obj,
```

db	Database
obj	Object (ucdbScopeT or ucdbTestT)
tag	Tag

Removes the given tag from the object. Returns 0 if successful, or non-zero if error.

ucdb_GetObjNumTags

```
int ucdb_GetObjNumTags(
    ucdbT               db,
```

db	Database
obj	Object (ucdbScopeT or ucdbTestT)

Gets the number of tags from a given object. Returns number of tags, or 0 if error or no tags.

ucdb_GetObjIthTag

```
int ucdb_GetObjIthTag(
    ucdbT               db,
    ucdbObjT            obj,
    int                 index,
```

db	Database
obj	Object (ucdbScopeT or ucdbTestT)

index	Tag index
tag	Tag

Gets an indexed tag from a given object. Returns 0 if successful, or non-zero if error.

ucdb_SetObjTags

```
int ucdb_SetObjTags(
    ucdbT                   db,
    ucdbObjT                obj,
    int                     numtags,
```

db	Database
obj	Object (ucdbScopeT or ucdbTestT)
numtags	Size of tag_array, 0 to clear all flags
tag_array	Array of string handles

Sets all tags for a given a object (replaces previous tags). Returns 0 if successful, or non-zero if error.

ucdb_BeginTaggedObj

```
int ucdb_BeginTaggedObj(
    ucdbT                   db,
    const char*             tagname,
```

db	Database
tagname	Tag to match
p_obj	Object (ucdbScopeT or ucdbTestT)

In-memory mode only. Gets the first object that exists with the given tag. Returns 1 if the tag exists in the database, or 0 if not. When the function returns 1, *p_obj is non-NULL.

ucdb_NextTaggedObj

```
int ucdb_NextTaggedObj(
    ucdbT                 db,
```

db	Database
p_obj	Object (ucdbScopeT or ucdbTestT)

In-memory mode only and must be called immediately after ucdb_BeginTaggedObj. The function reuses the tag from the previous call. Gets the next obj that exists with the given tag.

Returns 1 if the next object exists in the database, or 0 if not. When it returns 1, *p_obj is non-NULL.

ucdb_NextTag

```
int ucdb_NextTag(
    ucdbT               db,
```

db	Database
tagname	Tag name

In-memory mode only. Iterator function for returning the set of all tags in the UCDB file. Returns NULL when traversal is done or -1 with error.

Formal Data

A UCDB test is the result of functional verification analysis performed by a simulator or a formal verification tool. A formal test is a ucdbTestT object that is also associated with special information that describes a particular formal analysis session (ucdb_AssocFormalInfoTest).

This formal analysis information describes the following factors:

- How, when, and where the formal test ran
- Scope of the formal analysis
- Location of detailed results
- Environment assumptions

Formal analysis gives two types of results:

- Assertion information — Formal analysis of an assertion results in the formal status of the assertion based on the test assumptions for the scope of the assertion. For example: the assertion is proven; a counterexample exists that makes the assertion fail; or the formal analysis is inconclusive.
- Coverage information — Formal analysis of a cover statement or an assertion returns coverage information such as cover statement coverage, line coverage, stimulus coverage, and assertion witnesses that the assertions can be exercised. You model this functionality using the same scopes and coverage items as for simulation, in conjunction with additional facilities for formal verification.

A UCDB formal environment attribute indicates the context for interpreting the coverage data obtained from a formal analysis session. Coverage contexts support various formal coverage use models, for example:

- Coverage reachability is the primary objective of the formal analysis session, or it is an ancillary by-product of the formal analysis session.
- Coverage describes the controllability of the design based on the formal assumptions, or it indicates the design logic observable by assertions.

Formal coverage context shows how different types of coverage information were obtained and how you should interpret them.

Note

In general, all arguments returned by the formal routines are only valid as long as the *db* database remains open. Once the *db* database is closed, these arguments are invalid and should not be accessed in any way. If a caller of the formal routines needs access to the returned values beyond the lifetime of the *db* database, it must make copies of returned values.

Formal Status Enum

```
typedef enum {
   UCDB_FORMAL_NONE,           /* No formal info (default) */
   UCDB_FORMAL_FAILURE,        /* Fails */
   UCDB_FORMAL_PROOF,          /* Proven to never fail */
   UCDB_FORMAL_VACUOUS,        /* Assertion is vacuous as defined by the
                                  assertion language */
   UCDB_FORMAL_INCONCLUSIVE, /* Proof failed to complete */
   UCDB_FORMAL_ASSUMPTION,     /* Assertion is an assume */
   UCDB_FORMAL_CONFLICT        /* Data merge conflict */
} ucdbFormalStatusT;
```

Formal test result for a particular asserted or assumed property.

Formal Environment Type

```
typedef void* ucdbFormalEnvT;
```

Formal Tool Info Type

```
typedef ucisFormalToolInfoT ucdbFormalToolInfoT;
```

Structure identifying the test as a formal test and indicating tool-specific information about the formal analysis run:

- formal_tool — tool name
- formal_tool_version — tool version
- formal_tool_setup — setup file (text)

- formal_tool_db — database file (binary)
- formal_tool_rpt — report file (text)
- formal_tool_log — log file (text)

Formal Coverage Context

```
#define UCDB_FORMAL_COVERAGE_CONTEXT_STIMULUS \
   "UCDB_FORMAL_COVERAGE_CONTEXT_STIMULUS"
#define UCDB_FORMAL_COVERAGE_CONTEXT_RESPONSE \
   "UCDB_FORMAL_COVERAGE_CONTEXT_REPONSE"
#define UCDB_FORMAL_COVERAGE_CONTEXT_TARGETED \
   "UCDB_FORMAL_COVERAGE_CONTEXT_TARGETED"
#define UCDB_FORMAL_COVERAGE_CONTEXT_ANCILLARY \
   "UCDB_FORMAL_COVERAGE_CONTEXT_ANCILLARY"
#define UCDB_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS \
   "UCDB_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS"
```

Formal coverage context is a string that indicates the context for interpreting formal coverage information. This string can be one of the following predefined UCDB formal context attribute values, a user-defined string specific to the tool/application, or NULL (that is, no formal coverage context specified).

- UCDB_FORMAL_COVERAGE_CONTEXT_STIMULUS

  Coverage information associated with the test approximates the set of legal stimuli permitted within the constraints of the formal verification run. For example, for this formal coverage context, you can check that the test’s formal assumptions do not over- or under-constrain the formal analysis.

- UCDB_FORMAL_COVERAGE_CONTEXT_RESPONSE

  Coverage information associated with the test identifies the structures under observation by the assertions. For example, knowing the logic verified by formal analysis helps you determine the “completeness” of the assertion instrumentation of the design.

- UCDB_FORMAL_COVERAGE_CONTEXT_TARGETED

  Coverage information associated with the test is used for comprehensive coverage analysis. For example, one purpose might be to identify the controllable elements of the design. Another might be to evaluate the particular assumptions applied.

- UCDB_FORMAL_COVERAGE_CONTEXT_ANCILLARY

  Coverage information associated with the test is a by-product of formal analysis and is not the primary objective for the formal test. Results provide coverage information helpful in understanding what was exercised, but that information is not necessarily comprehensive. For example, the main objective of the formal verification test might be to prove assertions and find counterexamples. Here, parts of the design not in the fanin of the formal properties are typically ignored by the formal tool. So, coverage is a side effect of the formal analysis.

- UCDB_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS

  Coverage information associated with the test helps you analyze assertions with inconclusive formal analysis results (that is, assertions with UCDB_FORMAL_INCONCLUSIVE status).

ucdb_SetFormalStatus

```
int ucdb_SetFormalStatus(
    ucdbT                       db,
    ucdbTestT                   test,
    ucdbScopeT                  assertscope,
```

db	Database
test	UCDB test object
assertscope	Scope of the assertion
formal_status	Assert formal status

Sets the formal status of the specified assertion with respect to the specified test. Not supported in read-streaming mode. This is a routine that sets a value, so in write-streaming mode this routine can only be called while the scope of the assertion is actively being written. Returns 0 if successful, or non-zero if error (and formal status is unchanged). Returns an error if any argument is NULL.

ucdb_GetFormalStatus

```
int ucdb_GetFormalStatus(
    ucdbT                       db,
    ucdbTestT                   test,
    ucdbScopeT                  assertscope,
```

db	Database
test	UCDB test object
assertscope	Scope of the assertion
formal_status	Assert formal status returned

Gets the formal status of the specified assertion with respect to the specified test. Not supported in write-streaming mode. This is a routine that gets a value, so in read-streaming mode this routine can only be called while the scope of the assertion is actively being read. Neither iteration of assertscopes paired with a given test nor iteration of test with a given assertscope is supported. Returns 0 if successful, or non-zero if error (and formal status is not returned). Returns an error if any argument is NULL.

ucdb_SetFormalRadius

```
int ucdb_SetFormalRadius(
    ucdbT                   db,
    ucdbTestT               test,
    ucdbScopeT              assertscope,
    int                     radius,
```

db	Database
test	UCDB test object
assertscope	Scope of the assertion
radius	Radius returned (expressed in clock cycles) Exact meaning depends on the assertion’s status: • UCDB_FORMAL_INCONCLUSIVE Proof radius (if a bounded proof is reported) or -1 (if no bounded proof is reported). • UCDB_FORMAL_FAILURE Counterexample depth.
clock	Assertion clock specified as a hierarchical name string Can be NULL.

Sets the formal radius (proof radius or counterexample depth) for the specified assertion with respect to the specified test. Not supported in read-streaming mode. This is a routine that sets a value, so in write-streaming mode this routine can only be called while the scope of the assertion is actively being written. Returns 0 if successful, or non-zero if error (and formal radius is unchanged). Returns an error if any argument except clock is NULL.

ucdb_GetFormalRadius

```
int ucdb_GetFormalRadius(
    ucdbT                   db,
    ucdbTestT               test,
    ucdbScopeT              assertscope,
    int*                    radius,
```

db	Database
test	UCDB test object
assertscope	Scope of the assertion

radius

Radius returned (expressed in clock cycles)

Exact meaning depends on the assertion’s status:

- UCDB_FORMAL_INCONCLUSIVE

  Proof radius (if a bounded proof is reported) or -1 (if no bounded proof is reported).

- UCDB_FORMAL_FAILURE

  Counterexample depth.

clock

Assertion clock returned (specified as a hierarchical name string)

If NULL, the clock is NULL or the formal radius was not set.

Gets the formal radius for the specified assertion with respect to the specified test and gets the associated clock for the radius. Not supported in write-streaming mode. This is a routine that gets values, so in read-streaming mode this routine can only be called while the scope of the assertion is actively being read. Neither iteration of assertscopes paired with a given test nor iteration of test with a given assertscope is supported. Returns 0 if successful, or non-zero if error (and radius/clock are not returned). Returns an error if any argument is NULL.

ucdb_SetFormalWitness

```
int ucdb_SetFormalWitness(
    ucdbT                      db,
    ucdbTestT                  test,
    ucdbScopeT                 assertscope,
```

db

Database

test

UCDB test object

assertscope

Scope of the assertion

witness_file_or_dir

Path to a waveform file or directory containing waveform files, expressed as a string

Waveform files can be in any standard or widely-used format.

Sets witness waveforms for the specified assertion with respect to the specified test. A witness is a counterexample (for a failed property) or a sanity waveform (for a proven property). Not supported in read-streaming mode. This is a routine that sets a value, so in write-streaming mode this routine can only be called while the scope of the assertion is actively being written. Returns 0 if successful, or non-zero if error (and witness waveform information is unchanged). Returns an error if any argument is NULL.

ucdb_GetFormalWitness

```
int ucdb_GetFormalWitness(
    ucdbT                      db,
    ucdbTestT                  test,
    ucdbScopeT                 assertscope,
```

db	Database.
test	UCDB test object
assertscope	Scope of the assertion
witness_file_or_dir	Witness string returned. String is the path to a witness waveform file or a directory containing witness waveform files (expressed in a standard or widely-used format)

Gets witness waveforms for the specified assertion with respect to the specified test. A witness is a counterexample (for a failed property) or a sanity waveform (for a proven property). Not supported in write-streaming mode. This is a routine that gets a value, so in read-streaming mode this routine can only be called while the scope of the assertion is actively being read. Neither iteration of assertscopes paired with a given test nor iteration of test with a given assertscope is supported. Returns 0 if successful, or non-zero if error (and witness_file_or_dir is not returned). Returns an error if any argument is NULL.

ucdb_SetFormallyUnreachableCoverTest

```
int ucdb_SetFormallyUnreachableCoverTest(
    ucdbT                   db,
    ucdbTestT               test,
    ucdbScopeT              coverscope,
```

db	Database
test	UCDB test object
coverscope	Scope of the cover item
coverindex	Index of the cover item in the cover scope

Sets the formally-unreachable status flag for the specified cover item with respect to the specified test. Use this function in conjunction with ucdb_AssocCoverTest, which indicates whether or not the coverage item is reachable with respect to the test. With these two flags, you can indicate the status of the cover item with respect to a formal test covered by formal, proven unreachable, or unknown coverage status (that is, if both flags are clear).

Not supported in read-streaming mode. This is a routine that sets a value, so in write-streaming mode this routine can only be called while the scope of the cover item is actively being written. Returns 0 if successful, or non-zero if error (and formally-unreachable status flag is unchanged). Returns an error if any argument is NULL.

ucdb_ClearFormallyUnreachableCoverTest

```
int ucdb_ClearFormallyUnreachableCoverTest(
    ucdbT                   db,
    ucdbTestT               test,
    ucdbScopeT              coverscope,
```

db	Database
test	UCDB test object
coverscope	Scope of the cover item
coverindex	Index of the cover item in the cover scope

Clears the formally-unreachable status flag (ucdb_SetFormallyUnreachableCoverTest) for the specified cover item with respect to the specified test. Not supported in read-streaming mode. This is a routine that sets a value, so in write-streaming mode this routine can only be called while the scope of the cover item is actively being written. Returns 0 if successful, or non-zero if error (and formally-unreachable status flag is unchanged). Returns an error if any argument is NULL.

ucdb_GetFormallyUnreachableCoverTest

```
int ucdb_GetFormallyUnreachableCoverTest(
    ucdbT                      db,
    ucdbTestT                   test,
    ucdbScopeT                  coverscope,
    int                         coverindex,
```

db	Database
test	UCDB test object
coverscope	Scope of the cover item
coverindex	Index of the cover item in the cover scope
unreachable_flag	Flag value returned: • 0 — coverage item possibly reachable • 1 — coverage item formally unreachable

Gets the formally-unreachable status flag for the specified cover item with respect to the specified test. Not supported in write-streaming mode. This is a routine that gets a value, so in read-streaming mode this routine can only be called while the scope of the cover item is actively being read. Neither iteration of coverscopes paired with a given test nor iteration of test with a given coverscope is supported. Returns 0 if successful, or non-zero if error (and formally-unreachable status flag is not returned). Returns an error if any argument is NULL.

ucdb_AddFormalEnv

```
ucdbFormalEnvT ucdb_AddFormalEnv(
    ucdbT                       db,
    const char*                 name,
```

db	Database

name	Environment name.
scope	Scope indicating the part of the design analyzed by formal verification

Creates a new formal environment object. A formal environment describes the scope of a formal test and the environmental assumptions used to perform the formal analysis. Returns the handle for the new environment (if successful); returns the handle for an existing environment (if name and scope match those of an existing formal environment); or returns NULL if error. Names of formal environments must be unique, so it is an error if name matches an existing formal environment's name, but the two scopes do not match. Not supported in read-streaming mode. This is a routine that writes information, so in write-streaming mode this routine can only be called while the scope of the environment is actively being written.

Once a formal environment is created, use ucdb_AssocAssumptionFormalEnv repeatedly to associate assumption scopes with the environment. Then, use ucdb_AssocFormalInfoTest to associate the formal environment with formal tests run under those environmental constraints.

ucdb_AssocAssumptionFormalEnv

```
int ucdb_AssocAssumptionFormalEnv(
    ucdbT                 db,
    ucdbFormalEnvT        formal_env,
```

db	Database
formal_env	UCDB formal environment
assumption_scope	Scope of an assumption

Adds the specified assumption to the specified formal environment (created with ucdb_AddFormalEnv). Not supported in read-streaming mode. This is a routine that writes a value, so in write-streaming mode this routine can only be called while the scope of the assumption is actively being written. Returns 0 if successful, or non-zero if error (and assumption is not added to the environment).

ucdb_AssocFormalInfoTest

```
int ucdb_AssocFormalInfoTest(
    ucdbT                 db,
    ucdbTestT             test,
    ucdbFormalToolInfoT*  formal_tool_info,
    ucdbFormalEnvT        formal_env,
```

db	Database
test	UCDB test object
formal_tool_info	Formal tool information

formal_env	UCDB formal environment
formal_cov_context	Formal coverage context

Adds a formal environment, tool-specific information and a formal coverage context to the information for a test, which in effect makes test a formal test. Returns 0 if successful, or non-zero if error (and the formal information is not added to the test).

ucdb_NextFormalEnv

```
ucdbFormalEnvT ucdb_NextFormalEnv(
    ucdbT                   db,
```

db	Database
formal_env	UCDB formal environment (or NULL, to return the first formal environment)

Returns the handle for the first formal environment (if formal_env is NULL), or the next formal environment after formal_env, or NULL (if formal_env is the last environment added by ucdb_AddFormalEnv or if error).

ucdb_NextFormalEnvAssumption

```
ucdbScopeT ucdb_AssocAssumptionFormalEnv(
    ucdbT                 db,
    ucdbFormalEnvT        formal_env,
```

db	Database
formal_env	UCDB formal environment
assumption_scope	Scope of an assumption added to formal_env using ucdb_AssocAssumptionFormalEnv or NULL

Returns the handle for the first assumption added to formal_env (if assumption_scope is NULL), or the next formal environment after formal_env, or NULL (if assumption_scope is the last assumption added to formal_env or if error). Not supported in streaming mode (only supported in memory mode).

ucdb_FormalEnvGetData

```
int ucdb_FormalEnvGetData(
    ucdbT                   db,
    ucdbFormalEnvT          formal_env,
    const char**            name,
```

db	Database
formal_env	UCDB formal environment

name	Environment name returned
scope	Scope returned indicating the part of the design analyzed by formal verification

Gets the name and scope of the specified formal environment. Not supported in streaming mode (only supported in memory mode). Returns 0 if successful, or non-zero if error (and the formal environment information is not updated).

ucdb_FormalTestGetInfo

```
int ucdb_FormalTestGetInfo(
    ucdbT                   db,
    ucdbTestT               test,
    ucdbFormalToolInfoT** formal_tool_info,
    ucdbFormalEnvT*         formal_env,
```

db	Database
test	UCDB test object
formal_tool_info	Formal tool information returned
formal_env	UCDB formal environment returned
formal_cov_context	Formal coverage context returned

Gets the formal environment, tool information and formal coverage context for the specified formal test (from data created by ucdb_AssocFormalInfoTest). This function allocates and owns the memory for the returned values formal_tool_info and formal_cov_context, so the calling code should not “free” the memory these arguments point to. Returns 0 if successful, or non-zero if error (and the formal test information is not returned).

Test Traceability

API for associating tests and coverage objects. Coveritems or scopes may be associated with one of the ucdbTestT records in the database through this API.

Note

In regards to tests and coverage object association; for compactness, this association is implemented as a bit vector associated with each coverage object, where each bit corresponds to a test in the list of test data records in the database. Consequently, this association is dependent on the ordering of test data records being stable. If test data records are removed (with ucdb_RemoveTest()), all test-coverage associations can be invalidated.

NOTE on the

Some test traceability support functions use the ucdbBitVectorT structure, which contains a vector whose bits correspond to the test data records in the database.

```
typedef struct {
    unsigned char*      bitvector;      /* LSBs are filled first */
    int                 bitlength;      /* length in bits */
    int                 bytelength;     /* length in bytes */
} ucdbBitVectorT;
```

This structure is used for efficient implementation. When using ucdb_SetCoverTestMask() or other functions reading the bit vector, bitlength takes priority over bytelength, either will be ignored if set to -1. Both may not be set to -1. Setting length to 0 will erase the attribute.

The following optional defines enforce the conventions for bitlength versus bytelength in ucdbBitVectorT structures:

```
#define ucdb_SetBitVectorLengthBits(bitvector,numbits) \
    { (bitvector).bitlength = (numbits); \
      (bitvector).bytelength = (((bitvector).bitlength)/8) \
                               + (((bitvector).bitlength)%8) ? 1 : 0));}
#define ucdb_SetBitVectorLengthBytes(bitvector,numbytes) \
    { (bitvector).bytelength = (numbytes); \
      (bitvector).bitlength = ((bitvector).bytelength) * 8 ;}
#define ucdb_GetBitVectorLengthBytes(bitvector) \
    ((bitvector).bitlength >= 0 ? \
       (((bitvector).bitlength/8) + (((bitvector).bitlength%8) ? 1 : 0)) \
     : (bitvector).bytelength)
#define ucdb_GetBitVectorLengthBits(bitvector) \
     ((bitvector).bitlength >= 0 ? \
        (bitvector).bitlength \
     : (bitvector).bytelength * 8)
```

ucdb_AssocCoverTest

```
int ucdb_AssocCoverTest(
    ucdbT                   db,
    ucdbTestT               testdata,
    ucdbScopeT              scope,
```

db	Database
testdata	Test data record
scope	Scope
coverindex	Index of coveritem If -1, associate scope.

Associates a scope or coveritem with the given test data record. This may be done for any purpose, but is most logically done to indicate that the given test incremented or covered the bin; in-memory mode only. Returns 0 if successful, -1 for failure (for example, coverindex out-of-bounds.)

ucdb_NextCoverTest

```
ucdbTestT ucdb_NextCoverTest(
    ucdbT               db,
    ucdbScopeT          scope,
    int                 coverindex,
```

db	Database
scope	Scope
coverindex	Index of coveritem. If -1, scope only
test	Test

In-memory mode only. Gets the next test record associated with the given scope or coveritem. Returns the first record with NULL as input, or returns NULL when list is exhausted.

ucdb_GetCoverTestMask

```
int ucdb_GetCoverTestMask(
    ucdbT               db,
    ucdbScopeT          scope,
    int                 coverindex,
```

db	Database
scope	Scope
coverindex	Index of coveritem If -1, scope only
mask	Database bit vector

Gets a bit vector whose bits correspond to the associated test data records in the database. First bit (mask.bitvector[0]&0x01) corresponds to first test retrieved by ucdb_NextTest(), subsequent bits correspond in order to subsequent test data records. If tests are saved in an array, this enables quick retrieval of all associated tests in a single call. Returns 0 if successful, or -1 if error. mask.bitvector == NULL if none, lengths == 0.

This function always sets both bitlength and bytelength on the bitvector. bitvector storage is not to be de-allocated by the user.

ucdb_SetCoverTestMask

```
int ucdb_SetCoverTestMask(
    ucdbT               db,
    ucdbScopeT          scope,
    int                 coverindex,
```

db	Database

scope	Scope
coverindex	Index of coveritem If -1, scope only
mask	Database bit vector

Writes a bit vector whose bits correspond to the associated test data records in the database. This is for write-streaming versions of the API and is not as foolproof as ucdb_AssocCoverTest(). Returns 0 if successful, or -1 if error.

When initializing a mask, be careful with the rules for setting bitlength and bytelength, (see above). bitvector storage is copied by this routine.

ucdb_OrCoverTestMask

```
int ucdb_OrCoverTestMask(
    ucdbT               db,
    ucdbBitVectorT*     mask,
```

db	Database
mask	Database bit vector
test	Test

ORs the required bit for the given test data record. Returns 0 if successful, non-zero if error.

Appendix A
UCDB Organization

A UCDB file is organized into two sections: test and coverage.

Test Section 201

Coverage Section 203

Test Section

The test section of a UC database contains information about the test or set of tests that were used to generate the coverage data. If the file was created by merging multiple databases, the database contains multiple test records.

When creating a database, first define information about the test from which coverage data is acquired (see ucdb_AddTest). In addition to a fixed list of fields, any of which may be NULL or unused, there are user-defined attributes.

Table A-1. Fields of a Test Record

Field	Value	Description
testname	string	Name of the coverage test.
simtime	double	Simulation time of completion of the test.
simtime_units	string	Units for simulation time: "fs", "ps", "ns", "us", "ms", "sec", "min", "hr".
realtime	double	CPU time for completion of the test.
seed	string	Randomization seed for the test. (Same as the seed value provided by the "-sv_seed" vsim option.)
command	string	Test script arguments. Used to capture "knob settings" for parameterizable tests, as well as the name of the test script.
date	string	Time file was saved. For example, this might be a string like "20060105160030", which represents 4:00:30 PM January 5, 2006 (output of strftime with the format "%Y%m%d%H%M%S").
simargs	string	Simulator command line arguments.
userid	string	User ID of user who ran the test.
compulsory	boolean	Whether (1) or not (0) this test should be considered compulsory (that is, a “must-run" test.
comment	string	String (description) saved by the user associated with the test

Table A-1. Fields of a Test Record (cont.)

Field	Value	Description
test_status	int	Status of test: fatal error ($fatal was called), error ($error was called), warning ($warning was called) or OK.
filename	string	Name of the original file, to which the test was first written.

Test records are a subset of history nodes, have the following attributes:

Table A-2. Attributes of a History Node

Attribute	Value	Description
filename	string	Pathname of the merged file (UCDB_HISTORYNODE_MERGE), test file (UCDB_HISTORYNODE_TEST), or testplan file (UCDB_HISTORYNODE_TESTPLAN).
cmdline	string	Command line used to create resulting UCDB file associated with filename.
runcwd	string	Working directory where cmdline was executed.
cputime	double	(Optional) CPU time for the execution of cmdline.
histcomment	string	(Optional) String used as a general-purpose comment.
path	string	(UCDB_HISTORYNODE_TESTPLAN only) Testplan path.
xmlsource	string	(UCDB_HISTORYNODE_TESTPLAN only) XML filepath.
signature	string	(UCDB_HISTORYNODE_TESTPLAN only, optional) Source-based signature used to determine if the xmlsource file is stale.

Coverage Section

The coverage section of a UC database contains the coverage data, organized in a hierarchy of scopes related to the design, testbench, and testplan.

Scope Nodes 203
Coveritems 203
Nesting Rules 204
Attributes 208

Scope Nodes

Coverage data in the database form a tree of nodes, called scopes, generally corresponding to the design hierarchy. All nodes except the root node have a pointer to their parent. If the design hierarchy is not relevant to coverage, it need not be represented in the UCDB.

Nodes can have children: other scope nodes or coverage items. Design units (for example, Verilog modules or VHDL architectures) also are represented as scopes, because sometimes coverage for a design unit is often represented as a union of the coverage of all instances of the design unit. Typically, only code coverage is represented under the design unit. A design unit with a single instance a higher-level design are not stored (only the instance is stored).

Scope nodes can represent:

- Design hierarchy: instances of modules, function scope, packages, and so on.
- Hierarchy for coverage counts. For example:
  - Scopes to contain different counts for expression rows in expression coverage.
  - Scopes to represent SystemVerilog covergroups.

  If there is no coverage hierarchy (for example, with statement coverage) none is used.
- testplan items.

  These are optional, but are required for some use models of test traceability analysis. In particular, if you want the UCDB to represent associations between testplan items and coverage items using built-in "tags" (see “Tags” on page 183), then a testplan item scope should exist in the database.

Coveritems

Coveritems (coverage items) are always children of parent scopes and each coverage item is only accessible through its parent scope. This property of a UCDB enables optimizations related to efficiently storing a sets of coverage items that always lie in certain scopes.

A coveritem is a single count or vector of bits, generally used to compute coverage, represented in the database. In some coverage models (for example, SystemVerilog covergroups) coveritems these represent “bins”—the UCDB architecture is expanded to represent more types of coverage data.

A coveritem is only accessed through a handle to its parent scope and an index uniquely identifying it within the scope. The user can query a scope for how many coveritems it contains.

Nesting Rules

The UCDB does some light enforcement of HDL nesting rules, but strictly enforces nesting rules for coverage scopes, coveritems and testplan scopes.

The “covergroup” scopes are for generic use. For clarity, different types of coverage (assertion, statement, FSM, and so on) are given separate scopes, although the UCDB coverage hierarchy could have been built using only “covergroup” scopes only (COVERGROUP, COVERINSTANCE, COVERPOINT, and CROSS).

Table A-3. Nesting Rules Enforced by UCDB

Hierarchical Object	Rules
HDL SCOPE	Can contain any of: HDL SCOPE, COVER SCOPE and STANDALONE COVERITEM. Is one of the following scope types: UCDB_INSTANCE, UCDB_PACKAGE, UCDB_PROGRAM, UCDB_PACKAGE, UCDB_INTERFACE, UCDB_PROCESS, UCDB_GENERATE, UCDB_TASK, UCDB_FUNCTION, UCDB_FORKJOIN, UCDB_BLOCK, UCDB_CLASS, or UCDB_GENERIC
UCDB_INSTANCE	Contains a "DU" (design unit) or a "type" pointer to one of: UCDB_DU_MODULE or UCDB_DU_ARCH.
UCDB_PACKAGE	Contains a "DU" (design unit) or a "type" pointer to a UCDB_DU_PACKAGE.
UCDB_PROGRAM	Contains a "DU" (design unit) or a "type" pointer to a UCDB_DU_PROGRAM.
UCDB_INTERFACE	Contains a "DU" (design unit) or a "type" pointer to a UCDB_DU_INTERFACE.
DU SCOPE (that is, UCDB_DU_*)	Can contain: code coverage coveritems.

Table A-3. Nesting Rules Enforced by UCDB (cont.)

Hierarchical Object	Rules
COVER SCOPE	Is one of the following scope types: UCDB_COVERGROUP, UCDB_COVERINSTANCE, UCDB_COVERPOINT, UCDB_CROSS, UCDB_BRANCH, UCDB_EXPR, UCDB_COND, UCDB_TOGGLE, UCDB_FSM, UCDB_ASSERT, UCDB_COVER, UCDB_BLOCK, UCDB_CVGBINSCOPE, UCDB_ILLEGALBINSCOPE, UCDB_IGNOREBINSCOPE, UCDB_CROSSPRODUCT, UCDB_CROSSPRODUCT_ITEM.
STANDALONE COVERITEM	Is one of the following coveritem types: UCDB_STMTBIN, UCDB_USERBIN, UCDB_COUNT.
UCDB_TESTPLAN	Can contain only a UCDB_TESTPLAN scope.
UCDB_COVERGROUP	Can contain only the following scope types: UCDB_COVERINSTANCE, UCDB_COVERPOINT, UCDB_CROSS.
UCDB_CROSS	Must refer to at least two scopes of type UCDB_COVERPOINT, which must have the same parent as the UCDB_CROSS. UCDB_CROSS scope can contain only: • UCDB_CVGBINSCOPE scopes • UCDB_ILLEGALBINSCOPE scopes • UCDB_IGNOREBINSCOPE scopes • UCDB_CVGBIN coveritems • UCDB_ILLEGALBIN coveritems • UCDB_IGNOREBIN coveritems • UCDB_DEFAULT coveritems
UCDB_COVERPOINT	UCDB_COVERPOINT scope can contain only: • UCDB_CVGBINSCOPE scopes • UCDB_ILLEGALBINSCOPE scopes • UCDB_IGNOREBINSCOPE scopes • UCDB_CVGBIN coveritems • UCDB_ILLEGALBIN coveritems • UCDB_IGNOREBIN coveritems • UCDB_DEFAULT coveritems (can be ORed with each of the other bin types to indicate a default bin of the given type).

Table A-3. Nesting Rules Enforced by UCDB (cont.)

Hierarchical Object	Rules
UCDB_CVGBINSCOPE	UCDB_CVGBINSCOPE scope can contain only: • UCDB_CVGBIN coveritems • UCDB_ILLEGALBIN coveritems • UCDB_IGNOREBIN coveritems • UCDB_DEFAULT coveritems
UCDB_ ILLEGALBINSCOPE	UCDB_UCDB_ILLEGALBINSCOPE scope can contain only: • UCDB_CVGBIN coveritems • UCDB_ILLEGALBIN coveritems • UCDB_IGNOREBIN coveritems • UCDB_DEFAULT coveritems
UCDB_ IGNOREBINSCOPE	UCDB_IGNOREBINSCOPE scope can contain only: • UCDB_CVGBIN coveritems • UCDB_ILLEGALBIN coveritems • UCDB_IGNOREBIN coveritems • UCDB_DEFAULT coveritems
UCDB_ COVERINSTANCE	Can contain the only the following scope types: UCDB_COVERPOINT and UCDB_CROSS.
UCDB_ASSERT	Must contain UCDB_ASSERTBIN and can contain any of the following coveritems: UCDB_VACUOUSBIN, UCDB_DISABLEDBIN, UCDB_ATTEMPTSBIN, UCDB_ACTIVEBIN, UCDB_PEAKACTIVEBIN or UCDB_PASSBIN. No coveritem type can be represented more than once. Note: UCDB_ASSERTBIN indicates assertion failures. UCDB_PASSBIN contributes toward aggregated coverage.
UCDB_ASSERTBIN	Contains assert-fail count or boolean. Can be a direct descendant of the enclosing instance scope.
UCDB_COVER	Must contain exactly one UCDB_COVERBIN (indicating non-vacuous coverage passes or successes).
UCDB_COVERBIN	Contains non-vacuous cover pass count or boolean. Can be a direct descendant of the enclosing instance scope.
UCDB_STMTBIN	Can appear in any HDL scope.
UCDB_BRANCH	Must contain only UCDB_BRANCHBIN coveritems.

Table A-3. Nesting Rules Enforced by UCDB (cont.)

Hierarchical Object	Rules
UCDB_EXPR	Used in a 3-level hierarchy: • UCDB_EXPR top node contains name and source info. • UCDB_EXPR second-level nodes are named "FEC" and "UDP" for different representations of expression coverage UCDB_EXPRBIN coveritems. The coveritem name is a description of the expression truth table row. Can appear in any HDL scope or another UCDB_EXPR scope. Must contain only UCDB_EXPR scopes and UCDB_EXPR coveritems.
UCDB_COND	Used in a 3-level hierarchy: • UCDB_COND top node contains name and source info. • UCDB_COND second-level nodes are named "FEC" and "UDP" for different representations of condition coverage UCDB_CONDBIN coveritems. The coveritem name is a description of the expression truth table row. Can appear in any HDL scope or another UCDB_COND scope. Must contain only UCDB_COND scopes and UCDB_COND coveritems.
UCDB_TOGGLE	Must contain only UCDB_TOGGLEBIN coveritems (coveritem name is the name of toggle transition). For extended toggles: coveritems 0 and 1 are the low->high and high->low transitions, and coveritems 2-5 are the Z transitions. Toggle nodes, because of their abundance, are lighter-weight structures than all other types in the database, lacking some data that other scopes have.
UCDB_FSM	Must contain the two subscopes UCDB_FSM_STATES and UCDB_FSM_TRANS.
UCDB_FSM_STATES	Must contain UCDB_FSMBIN coveritems.
UCDB_FSM_TRANS	Must contain UCDB_FSMBIN coveritems.
UCDB_BLOCK	Can appear in any HDL scope or another UCDB_BLOCK scope. Must contain only UCDB_BLOCK scopes, UCDB_BLOCKBIN coveritems and UCDB_STMTBIN.

Table A-3. Nesting Rules Enforced by UCDB (cont.)

Hierarchical Object	Rules
UCDB_HIERARCHY	Light-weight hierarchy node that can have any other scope nodes as parents or children. Supports the user-defined attribute mechanism but not other attributes (such as design unit, source references, and so on). Useful for representing hierarchies that can be merged. The following functions cannot use the UCDB_HIERARCHY scope: ucdb_*File*, ucdb_InstanceSetDU, ucdb_*ScopeFlags, ucdb_*ScopeSourceType, ucdb_*ScopeSourceInfo, ucdb_*ScopeWeight, ucdb_*ScopeGoal, ucdb_GetInstanceDU*, ucdb_*Tag*.

Attributes

UCDB attributes provide a faster access mechanism for some frequently accessed attributes, compared to user-defined attributes.

Table A-4. UCDB Defined Attributes

Attribute	Type	Macro	Definition
Test Attributes			
SIMTIME	double	UCDBKEY_ SIMTIME	Simulation time.
TIMEUNIT	string	UCDBKEY_ TIMEUNIT	Time unit for SIMTIME.
CPUTIME	string	UCDBKEY_ CPUTIME	CPU time.
DATE	string	UCDBKEY_ DATE	Time at which the UCDB save was initiated.
VSIMARGS	string	UCDBKEY_ SIMARGS	Simulator command line arguments.
USERNAME	string	UCDBKEY_ USERNAME	Name of the user who ran the test.
TESTSTATUS	ucdbTest-StatusT	UCDBKEY_ TESTSTATUS	Status of the simulation run.
TESTNAME	string	UCDBKEY_ TESTNAME	Name of the test.

Table A-4. UCDB Defined Attributes (cont.)

Attribute	Type	Macro	Definition
ORIGFILE-NAME	string	UCDBKEY_ FILENAME	Database filename that the test was originally written to.
SEED	string	UCDBKEY_ SEED	0 or the seed provided by the -sv_seed vsim option.
TESTCMD	string	UCDBKEY_ TESTCMD	String provided by the user intended for test arguments.
TESTCOMMENT	string	UCDBKEY_ TESTCOMMENT	General-purpose comment provided with the test.
COMPULSORY	int (0|1)	UCDBKEY_ COMPULSORY	Whether (1) or not (0) the test is compulsory.
RUNCWD	string	UCDBKEY_ RUNCWD	When this attribute exists, it holds the working directory of the simulation from which the UCDB was saved.
Code Coverage Attributes			
#SINDEX#	int (>0)	UCDBKEY_ STATEMENT_INDEX	Statement number of a statement or expression in a design unit, starting at 1.
#SLINENO#	int (>0)	UCDBKEY_ START_LINENO	Line number of a statement or expression in a design unit, starting at 1.
#STOKNO#		UCDBKEY_ START_TOKNO	
#BCOUNT#	int	UCDBKEY_ BRANCH_COUNT	Total count of a branch scope (sum of true counts of individual branch cover items plus the count of the else branch).
#BTYPE#	int (0|1)	UCDBKEY_ BRANCH_ISCASE	Branch type: if-else (0) or case (1).
#BHASELSE#	int (0|1)	UCDBKEY_ BRANCH_HASELSE	Whether (1) or not (0) branch has an else clause.

Table A-4. UCDB Defined Attributes (cont.)

Attribute	Type	Macro	Definition
#EHEADER#	string	UCDBKEY_EXPR_HEADERS	Header strings for each column of the table separated by ’;’. Used on expression or condition scopes.
#FSMID#	string	UCDBKEY_FSM_ID	Symbolic name for an FSM state, usually derived from the state variable. Used with FSM coverages
#FSTATEVAL#	int	UCDBKEY_FSM_-STATEVAL	Value of an FSM state. Used on FSM coverage state coveritems.
#FSMCOND#		UCDBKEY_FSMCOND	
#NFSMPROCS#		UCDBKEY_NUMFSMPROCESSES	
#FSMPROCS#		UCDBKEY_FSMPROCESSES	
#CLOCK#		UCDBKEY_FSMCLOCK	
#FECSTR#		UCDBKEY_FECSTR	(Deprecated)
#CEXPRSTR#		UCDBKEY_CEXPR_STR	
SystemVerilog covergroups Attributes			
BINRHS	string	UCDBKEY_BINRHSVALUE	RHS value of a bin, a string that describes the sampled values that potentially could cause the particular bin to increment. Used on SV coverpoint coveritems (bins).
#GOAL#	int	UCDBKEY_GOAL	The option.goal or type_option.goal of the object. Used on SV covergroup, coverpoint or cross scopes.

Table A-4. UCDB Defined Attributes (cont.)

Attribute	Type	Macro	Definition
#GOAL#	float	UCDBKEY_ GOAL	Arbitrary goal that can have an effect (as for TESTPLAN scopes) in GUIs or reports. Used on other types of scopes.
ATLEAST	int	UCDBKEY_ ATLEAST	The option.at_least or type_option.at_least of the object. Used on SV covergroup, coverpoint or cross scopes.
COMMENT	string	UCDBKEY_ COMMENT	The option.comment or type_option.comment of the object. Used on SV covergroup, coverpoint or cross scopes.
AUTOBINMAX	int	UCDBKEY_ AUTOBINMAX	The option.auto_bin_max of the object. Used on SV covergroup or coverpoint scopes.
DETECT- OVERLAP	int (0|1)	UCDBKEY_ DETECTOVERLAP	The option.detect_overlap of the object. Used on SV covergroup or coverpoint scopes.
PRINT- MISSING	int	UCDBKEY_ NUMPRINTMISSING	The option.cross_num_print_missing of the object. Used on SV covergroup or cross scopes.
STROBE	int (0|1)	UCDBKEY_ STROBE	The type_option.strobe of the object. Used on SV covergroup scopes.
PERINSTANCE		UCDBKEY_ PERINSTANCE	
GETINSTCOV		UCDBKEY_ GETINSTCOV	
MERGEINSTANCES		UCDBKEY_ MERGEINSTANCES	

Table A-4. UCDB Defined Attributes (cont.)

Attribute	Type	Macro	Definition
MERGEINST_ISAUTO		UCDBKEY_ MERGEINST_ISAUTO	
#CROSSERR#	int (0|1)	UCDBKEY_ CROSSERROR	When 1, indicates a cross type coverage calculation not supported by the simulator (that is, when crossed coverpoints are parameterized with different numbers of bins in different covergroup instances). Used on SV covergroup scopes.
REAL_INTERVAL		UCDBKEY_ REAL_INTERVAL	
NUMSAMPLED	int	UCDBKEY_ NUMSAMPLED	Optional sample count for covergroups
Cover and Assertion Memory Profile Attributes			
MEM_ASSERT		UCDBKEY_MEM_ ASRTCURR	Current memory.
MEM_ASSERT		UCDBKEY_MEM_ ASRTPEAK	Peak memory.
CMLTTHREADS _ASR		UCDBKEY_ CMLTTHREADS_ ASRT	Cumulative threads.
TIME_ PEAKMEM		UCDBKEY_MEM_ PEAKTIME	Time of peak.
#SAMPLES#		UCDBKEY_ SAMPLES	Array of sample counts, for level 2 merge
Covergroup Memory Profile Attributes			
PERSISTMEM_CVG		UCDBKEY_MEM_ CVGPERSIST	Persistent memory.
TRANSMEM_ CVG		UCDBKEY_MEM_ CVGTRANS	Transient memory.

Table A-4. UCDB Defined Attributes (cont.)

Attribute	Type	Macro	Definition
TRANSPEAK_CVG		UCDBKEY_MEM_CVGTRANS_PEAK	Transient peak.
UCDBKEY_MEM_PEAKTIME		UCDBKEY_MEM_CVGTRANS_PEAKTIME	Time of peak.
Assertion Directive Attributes			
#ACTION#	int (0|1|2)	UCDBKEY_ASSERT_ACTION	Simulator action performed when the assertion fails: continue (0), break (1) or exit (2). Used on assertion objects.
PROOFRADIUS	int	UCDBKEY_ASSERT_PROOFRADIUS	Proof radius from formal analysis of the assertion.
SEVERITY		UCDBKEY_ASSERT_SEVERITY	Severity metric for the assertion.
General Attributes			
#	binary: bit vector	UCDBKEY_TESTVECTOR	Indicates which tests caused the object to be covered. Used on bins and UCDB_TOGGLE coverage scope.
MERGED		UCDBKEY_TESTDATA_MERGED	
TAGCMD	string	UCDBKEY_TAGCMD	Semicolon-separated arguments to "coverage tag" command. This supports implicit tagging during merge, so as to associate testplans with coverage for test traceability. Used for UCDB_TESTPLAN scopes.

Table A-4. UCDB Defined Attributes (cont.)

Attribute	Type	Macro	Definition
#SECTION#	string	UCDBKEY_SECTION	Section number within testplan. Used for UCDB_TESTPLAN scopes.
#DUSIGNATURE#	string	UCDBKEY_-DUSIGNATURE	MD5 signature string of a source design unit.
#COV#	float	UCDBKEY_COV	Used by coverage analysis to cache a computed total coverage number. Used for any scope.
MERGELEVEL	int (1|2)	UCDBKEY_MERGELEVEL	Used with merge files.[1][2]

[1] Default merge, test data is merged, the union of bins are merged, with integer counts incremented and vector counts ORed.

[2] Tests are associated with most bins as a bit vector indicating what test caused them to be covered. For vector bins, this means non-zero. For UCDB_COVER scopes, this means cover count > at_least; for UCDB_ASSERT scopes, this means fail count > 0; for UCDB_TOGGLE scopes, this means all bins covered (>0) except for UCDB_TOGGLE_ENUM types, where individual bins >0. Also: NUMSAMPLED attributes for UCDB_COVERGROUP and UCDB_COVERINSTANCE scopes are combined into a binary attribute called "SAMPLED" that is an array of as many integers as there are tests.

Table A-5. UCDB Defined Objects

Attribute	Macro	Definition
Some UCDB bin names are predefined to identify which count value is for a particular coveritem. These names are the names of coveritems, where applicable.		
true_branch	UCDBBIN_BRANCH_T	Branch true bins.
false_branch	UCDBBIN_BRANCH_F	Branch true bins.
else_branch	UCDBBIN_BRANCH_E	else count
all_false_branch	UCDBBIN_BRANCH_AF	All false count when there is no else part.
toggle_low	UCDBBIN_TOGGLE_L	2-state toggle bins
toggle_high	UCDBBIN_TOGGLE_H	2-state toggle bins
toggle_h_l	UCDBBIN_TOGGLE_EXT_H_L	3-state (extended) toggles
toggle_l_h	UCDBBIN_TOGGLE_EXT_L_H	3-state (extended) toggles
toggle_z_l	UCDBBIN_TOGGLE_EXT_Z_L	3-state (extended) toggles
toggle_l_z	UCDBBIN_TOGGLE_EXT_L_Z	3-state (extended) toggles

Table A-5. UCDB Defined Objects (cont.)

Attribute	Macro	Definition
toggle_h_z	UCDBBIN_TOGGLE_EXT_H_Z	3-state (extended) toggles
toggle_z_h	UCDBBIN_TOGGLE_EXT_Z_H	3-state (extended) toggles
unknown	UCDBBIN_EXPRCOND_UNKNOWN	Unknown value row.
Some of the UCDB scope names are hard coded to distinguish between different natures of scopes.		
FEC	UCDBSCOPE_FEC	Name of FEC scope.
UDP	UCDBSCOPE_UDP	Name of UDP scope.
UCDB select flags used to specify different objects types in various routines, such as making clones, printing objects, and so on.		
0x0001	UCDB_SELECT_TAGS	Select scope tags.
0x0002	UCDB_SELECT_ATTRS	Select user defined attributes.
0x0004	UCDB_SELECT_COVERS	Select covers (does not work with copy in streaming modes).
0x0008	UCDB_SELECT_FILETABS	Select file tables.
0x0010	UCDB_SELECT_SOURCEINFO	Select source information (print only).
0xffffffff	UCDB_SELECT_ALL	Select all flags above.

Generic UCDB Handle

```
#ifndef DEFINE_UCDBT
#define DEFINE_UCDBT
typedef void* ucdbT;          /* generic handle to a UCDB */
#endif
```

Size-critical Types

```
#if defined (_MSC_VER)
typedef unsigned __int64 uint64_t;
typedef signed __int64 int64_t;
typedef unsigned __int32 uint32_t;
#elif defined(__MINGW32__)
#include <stdint.h>
#elif defined(__linux)
#include <inttypes.h>
#else
#include <sys/types.h>
#if defined(__STRICT_ANSI__)
#ifdef  _LP64
typedef long int64_t;
typedef unsigned long uint64_t;
#else
typedef long long int64_t;
typedef unsigned long long uint64_t;
#endif
#endif
#endif
#ifdef WIN32
#define INT64_LITERAL(val)     ((int64_t)val)
#define INT64_ZERO             ((int64_t)0)
#define INT64_ONE              ((int64_t)1)
#define INT64_NEG1             ((int64_t)-1)
#else
#define INT64_LITERAL(val)     (val##LL)
#define INT64_ZERO             (0LL)
#define INT64_ONE              (1LL)
#define INT64_NEG1             (-1LL)
#endif
typedef uint64_t ucdbCoverTypeT;           // typedef for one of these
typedef uint64_t ucdbCoverMaskTypeT;       // typedef for a set of these.
```

Appendix B
UCDB Diff BNF

This appendix has the BNF definition of the UCDB sytntax.

UCDB Diff BNF Syntax. **217**

UCDB Diff BNF Syntax

UCDB Diff BNF Definition.

any_diff_line :== *diff_line* | *diff_comment* | *summary_line*

diff_comment :== -- *comment_text* --

summary_line :== SS *tbd_format*

diff_line :== *diff_file_location diff_text*

diff_file_location :== <> | << | >>

diff_text :== *ucdb_structural_type primary_key diff_aspect* [*diff_details*]

ucdb_structural_type :== Scope | Bin | Historynode | UCDBRoot

primary_key :== *scope_key* | *bin_key* | *historynode_key*

scope_key :== *ucdb_scope_type_string* "*ucdb_hiername*"

bin_key :== *ucdb_bin_type_string* "*ucdb_hiername*" "*coveritemname*"

ucdb_scope_type_string :== Branch | Toggle | Covergroup | ...

ucdb_bin_type_string :== BranchBin | ToggleBin | StatementBin | ...

historynode_key :== "*historynode_logical_name*"

diff_aspect :== Structural | Attribute | Flag | Flagfield | Tag | DU | Source | Count | Goal | Weight | Limit | Bitlen | Kind | Sourceinfo | Version

diff_value :== *attribute_diff_value* | *integer integer* | *float float* | *first_value second_value*

attribute_diff_value :== "*attribute_name*" *attribute_type* [*attribute_type*] "*attribute_value*" ["*attribute_value*"] *attribute_type* :== Int | Float | Double | String | Memblk | Long | Handle | Array

attribute_value :== *numeric_value* | *string* | *memblk_representation*

memblk_representation :== *num_bytes*bytes:MEMBLK | *num_bytes*bytes:*hex_byte_list*

historynode_type_string :== Test | Merge | Testplan

num_bytes :== *integer*

hex_byte_list :== *xx*[_*xx*]

x := *hex_digit*

Glossary

Child

Node that is a descendant of another, where *descendant* means nesting in a design hierarchy, in a coverage hierarchy or as a subset of data categorized with the parent.

Coverage Scope

Scope that represents a coverage grouping of some kind.

Coveritem

Leaf node in a UCDB (that is, a node not capable of having child nodes) used to store a coverage count.

Design Hierarchy

Part of the UCDB data model representing the design, testbench, and coverage.

Design Unit

Scope that represents a Verilog (or SystemVerilog) module or a VHDL entity-architecture.

Design Unit List

Set of all design units in a UCDB.

History Node

Generalized test data record that captures information about the database merges and testplan imports used to create the UCDB.

Instance

Scope that represents a component instance (for example, a module instantiation) in the design hierarchy.

Node

General term for a scope or coveritem.

Parent

Ancestor node (of a child), which represents a higher level of design hierarchy, a higher level of coverage hierarchy or a grouping.

Scope

Hierarchical object in a UCDB (that is, a node capable of having child nodes).

Tag

Name associated with a scope—typically used to link testplan scopes with instance, coverage, or design unit scopes—similar to a user-defined attribute with a name but not a value.

Testplan Hierarchy

Data model structure (whose nodes are linked to coverage, instance, or design unit data structures) used to analyze coverage in the context of a testplan.

Testplan Scope (or Testplan Section)

scope that represents part of a testplan.

Test data record

Data model structure that stores information about the test and the tool from which the UCDB was created.

User-Defined Attribute

Name-value pair (explicitly added by the user) that is not part of the UCDB primary data model.

End-User License Agreement
with EDA Software Supplemental Terms