Unified Coverage Interoperability Standard (UCIS)

Version 1.0

June 2, 2012

Accellera Systems Initiative, 1370 Trancas Street #163, Napa, CA 94558, USA.

Notices

Accellera Systems Initiative Standards documents are developed within Accellera Systems Initiative and the Technical Committees and Working Groups of Accellera Systems Initiative, Inc. Accellera Systems Initiative develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of Accellera and serve without compensation. While Accellera Systems Initiative administers the process and establishes rules to promote fairness in the consensus development process, Accellera Systems Initiative does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Systems Initiative Standard is wholly voluntary. Accellera Systems Initiative disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Systems Initiative Standard document.

Accellera Systems Initiative does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Systems Initiative Standards documents are supplied "AS IS."

The existence of an Accellera Systems Initiative Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Systems Initiative Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Systems Initiative Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Systems Initiative Standard.

In publishing and making this document available, Accellera Systems Initiative is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera Systems Initiative undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Systems Initiative Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera Systems Initiative, Accellera Systems Initiative will initiate reasonable action to prepare appropriate responses. Since Accellera Systems Initiative Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera Systems Initiative and the members of its Technical Committees and Working Groups are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Systems Initiative Standards are welcome from any interested party, regardless of membership affiliation with Accellera Systems Initiative. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Accellera Systems Initiative, 1370 Trancas Street #163, Napa, CA 94558 USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera Systems Initiative shall not be responsible for identifying patents for which a license may be required by an Accellera Systems Initiative standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera Systems Initiative is the sole entity that may authorize the use of Accellera Systems Initiative -owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera Systems Initiative Inc., provided that permission is obtained from and any required fee, if any, is paid to Accellera Systems Initiative. To arrange for authorization please contact Lynn Bannister, Accellera Systems Initiative, 1370 Trancas Street #163, Napa, CA 94558, phone (707) 251-9977, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera Systems Initiative.

Suggestions for improvements to the Unified Coverage Interoperability Standard are welcome. They should be sent to:
review-ucis@lists.accellera.org

The current Technical Committee web page is:
www.accellera.org/activities/committees/ucis

ACKNOWLEDGEMENTS

This Accellera Unified Coverage Interoperability Standard was specified and developed by experts from many different fields and different companies. This standard would not have been possible without the support and contributions of Accellera member companies, their employees and guest members of the committee. We would like to acknowledge and thank them for their work. In particular, the following individuals have been major contributors to this standard.

C. Richard Ho (D. E. Shaw Research) UCIS Co-Chair

Ambar Sarkar (Paradigm Works) UCIS Co-Chair

Hemant Gupta (Cadence Design Systems)

Sandeep Pagey (Cadence Design Systems)

Abigail Moorhouse (Mentor Graphics)

Andrew Seawright (Mentor Graphics)

Samiran Laha (Mentor Graphics)

Mark Strickland (Cisco Systems)

Amol Bhinge (Freescale Semiconductor)

Nancy Pratt (IBM)

Rajeev Ranjan (Jasper Design Automation)

Darron May (Mentor Graphics)

David C. Scott (Carbon Design Systems)

Stephen Dyer (Paradigm Works)

Michael Burns (Oracle Corporation)

Dan Benua (Synopsys Inc.)

Surrendra Dudani (Synopsys Inc.)

Vernon Lee (Synopsys Inc.)

Mehdi Mohtashemi (Synopsys Inc.)

Robert Porter (Hewlett Packard)

Karen Pieper (Accellera)

John Brennan (Cadence Design Systems)

Hillel Miller (Freescale Semiconductor)

Mor Segal (Intel Corporation)

Harry Foster (Mentor Graphics)

Faisal Haque (Qualcomm)

Initial Version 1.0 released June 2012

Table of Contents

1 Overview

This chapter contains the following main sections:

1.1 Scope

Verification of complex electronic circuits frequently requires the utilization of multiple verification tools, possibly from multiple vendors, and the employment of different verification technologies. The Unified Coverage Interoperability Standard (UCIS) provides an application programming interface (API) that enables the sharing of coverage data across software simulators, hardware accelerators, symbolic simulations, formal tools or custom verification tools. Coverage data is typically used during verification to determine if the verification goals have been met when using different tools and methodologies. The growing complexity of designs now requires that coverage data be shared among different tools to achieve verification closure. This document defines the UCIS, which is a common standard for exchanging verification coverage across a multitude of tools.

This document describes the UCIS, including the UCIS API, an abstract representation of the coverage database called the UCIS database (UCISDB), the XML interchange format for text-based interoperability, and examples of how to use UCIS to develop unified coverage applications.

The goals of this standard include:

— Standardize coverage definitions for commonly-used verification metrics.
— Enable sharing of verification coverage data between different classes of verification tools, i.e. between dynamic and static verification tools.
— Encourage user and electronic design automation (EDA) technology advancements for innovative verification solutions.
— Preserve coverage data across the lifetime of a project as tools evolve.
— Identify interoperability requirements among various coverage sources.
— Define an interface that allows verification coverage information to be exchanged among different EDA tools, both user created and vendor supplied.
— Define standard data models for representing coverage information.
— Define an XML based interchange format that allows verification coverage information to be exchanged in textual form.

1.2 Audience

This document is intended for verification engineers and anyone performing electronic design verification across multiple test platforms or multiple test runs and wishing to merge the coverage information from various sources to help identify verification weaknesses (*holes*) and to measure verification completeness.

1.3 Organization of this document

This document describes the UCIS and covers the following main topics:

- Use models that cover common verification activities
- The data model
- The API
- The XML interchange format

These main topics are discussed within the following sections in this document:

The UCIS is composed of the material in chapters 1-9 only. Annex A and B are provided as supplementary material.

1.4 Document conventions

This document uses conventions shown in Table 1-1, “Document conventions.”

Table 1-1 — Document conventions

Visual cue	Represents
`courier`	The `courier` font represents code. For example: `ucisT ucis_Open(const char* name);`
italic	The *italic* font represents a variable. For example, the *name* argument in the ucis_Open function: `ucisT ucis_Open(const char* name);`
[ ] square brackets	Name choices are indicated by [choice1 | choice2 | choice3]. The ‘[‘, ‘|’ and ‘]’ characters are optional in the definitions. If present, they are not part of the constructed name.

1.5 Definitions, acronyms and abbreviations

1-referenced – When referring to a count or sequence of positive integers, a 1-referenced count starts at number 1.

assertion coverage - See coverage definitions: assertion coverage.

assumption – A specification that a given behavior (e.g., modeled as a property) is believed to hold in a given context, and therefore (a) is not checked, and/or (b) may be used as part of the basis for verifying that some other property holds in that context. An assumption could represent either a constraint or a restriction. In the latter case, only (b) applies.

attribute – A property of a coverage object. Includes static information such as filename and line number of the source code of a coverage object; and dynamic information such as counts and thresholds. These may be specified at the creation of the object or added later. All attributes shall be both readable and writable through the API.

bin – SystemVerilog bins are represented in the UCIS model by coveritems named according to the SystemVerilog bin naming conventions. Strictly, “bin” is SystemVerilog terminology, but UCIS coveritems may also informally be referred to as bins; both are counter constructs.

block coverage - See coverage definitions: line coverage/statement coverage/block coverage.

branch coverage - See coverage definitions: branch coverage.

child – A node that is a descendant of another, where that represents nesting in terms of design hierarchy, coverage hierarchy, or representing a subset of data that could be categorized with the parent.

condition coverage - See coverage definitions: condition/expression coverage.

constraint – A specification that defines legal behaviors of the environment, such as a statement that a given design input value must always be in a certain range, or that design inputs must be mutually exclusive, or that the environment must drive design inputs in a certain manner in response to design outputs.

cover property - A property (assertion) specification marked as a "cover" item rather than an "assert" item. The UCIS definition of cover property follows the IEEE-1800 SystemVerilog definition of cover property.

coverage definitions:

> **assertion coverage** - An assertion is covered if its antecedent becomes true and the consequent is evaluated in verification. The UCIS definition of assertion property follows the IEEE-1800 SystemVerilog definition of assertion.
>
> **branch coverage** - The branches of a conditional operator in the design or verification environment are covered when the respective exit paths of the conditional operator are executed in verification.
>
> **condition/expression coverage** - The conditional terms in the design or verification environment are covered when the condition expression takes one or more of its possible values when executed in verification. Counts may be kept for value assignments to variables and sub-expressions. Each sub-expression of a conditional may be a separate metric.
>
> **line coverage/statement coverage/block coverage** - A line or statement or contiguous block of lines in the design or verification environment is covered if that line/statement/block is exercised in verification. Each metric is measured per textual instance in the in the fully elaborated design or verification simulation environment.
>
> **path coverage** - A path through a directed graph (for example: a finite-state machine) is covered if verification traverses the state/node sequence of the path. The directed graph may represent states of the design or test envi-

ronment or other abstract representation. Transition coverage is a subset of path coverage where the path is between only 2 nodes.

state coverage - A state or node of a directed graph (for example: a finite-state machine) is covered if verification enters the state/node. The directed graph may represent states of the design or test environment or other abstract representation. Nodes may have a type associated with them.

statement coverage - See coverage definitions: line coverage/statement coverage/block coverage.

toggle coverage - A register, wire or signal in the design or verification environment is covered if that register or wire changes from a zero to one back to zero, or from a one to zero back to one during verification. Multi-bit toggle coverage is treated as a union of single-bit toggle coverage. Other interpretations may include partial toggle such as zero to one and one to zero in a single testcase.

transition coverage - A transition of a directed graph (for example: a finite-state machine) is covered if verification traverses the state/node transition arc. The directed graph may represent states of the design or test environment or other abstract representation.

value-transition coverage - A value tuple (value0, …, valueN) consisting of 2 or more values that a signal may take is covered if verification causes the signal to transition from value0 to value1 and so on through the full set of values in the tuple.

coverage data – Data, organized as one or more fields, associated with a coverage object.

coverage object – An instance of a coverage type in the topology for which coverage data is maintained.

coverage scope – A grouping of coveritems or (recursively) other coverage scopes.

coverage type – A coverage metric. For example, line coverage, assertion coverage, block coverage or branch coverage. May be sourced by any element of the topology.

covergroup: The UCIS scoping type to represent covergroups closely follows the IEEE-1800 SystemVerilog definition of the covergroup type, and is used to represent the data for the type collected from SystemVerilog covergroup instances. The UCIS definition is a generalization of the concept, and its use may be extended beyond SystemVerilog.

coverinstance – The UCIS coverinstance scoping type represents the instances of a covergroup type that are brought into existence during a simulation run. Thus there may be many coverinstances for each covergroup type, sourced from one or many variables of the covergroup type.

coveritem – The UCIS coveritem construct is a generalized decorated integral count. The decoration (name, type, attributes, etc.) are used to hold the information as to what was counted. The count itself, holds the information as to how many times the described event was observed. This general model is specialized for each of the many different types of coverage counts supported by the UCIS model.

coverpoint: The UCIS scoping type to represent coverpoints closely follows the IEEE-1800 SystemVerilog definition of coverpoints within a covergroup type definition. As a covergroup scope component, its use may similarly be extended beyond SystemVerilog.

cross-coverage point - A cross-coverage point is the Cartesian product of 2 or more sets of coverage points.

design hierarchy – The part of the UCISDB data model representing the design, testbench, and coverage.

design unit – A scope that represents a module type (in Verilog or SystemVerilog) or entity-architecture (in VHDL).

design unit list – The set of all design units in a UCISDB.

DU – Design Unit.

DUT – Device Under Test.

DUT coverage – Coverage data gathered on the code that implements the device under test.

dynamically-allocated coverage object – A coverage object which is not fully specified until verification run is under way, and hence could not be present in a UCISDB at completion of elaboration.

embedded covergroup – Embedded covergroup is SystemVerilog terminology, defined in IEEE-1800. The SystemVerilog rules governing embedded covergroups are slightly different from regular covergroups. Specifically, the covergroup variable in the class instance may only be associated with at most a single coverinstance. There may however be multiple instances of the enclosing class, thus there may also be multiple instances of the coverinstance. Class typing may also be differentiated by parameterization; in this case the embedded covergroups are also type-differentiated, even if the covergroup is not specifically affected by the class parameters. Also note that a class definition is not restricted to a single embedded covergroup definition, it may contain multiple independent embedded covergroups.

expression coverage - See coverage definitions: condition/expression coverage.

extract – An operation which creates a new UCISDB from one or more input UCISDBs, upon which all API operations can be performed. An extracted UCISDB may be a subset, union, intersection or some other combination of the input UCISDBs.

FSM – Finite State Machine.

HDL – Hardware Description Language.

HLD – High Level Design.

heterogeneous – A property of the relationship between two UCISDBs which indicates that one contains coverage data of a different type than the other.

instance – A scope that represents a design instance (e.g., module instantiation) in design hierarchy.

interface coverage – Coverage data gathered on the interface between the testbench and the DUT.

node – A general term for a scope or coveritem.

parent – A node that is an ancestor of another, representing higher level of design hierarchy or coverage hierarchy or grouping.

path coverage - See coverage definitions: path coverage.

PSL – Property Specification Language.

query – An operation which yields a report based on matches to user-specified data found in a UCISDB.

restriction – A specification that limits a verification run so that it considers only a proper subset of the legal behaviors of the environment or the design. A restriction may be used to define each distinct sub-case of a complex behavior (case-splitting), in order to reduce the difficulty of formally verifying the general behavior by partitioning it into a collection of simpler behaviors. Restrictions are also used in simulation, to focus a particular verification run on a given subcategory of legal environment behavior.

scope – A hierarchical object in a UCISDB, capable of having child nodes.

spatial – A property of the relationship between two UCISDBs which indicates that one contains data from a part of the design that is not present or changed in the other. The data may be overlapping, such that one UCISDB is a subset of the other, or it may be disjoint.

state coverage - See coverage definitions: state coverage.

statement coverage - See coverage definitions: statement coverage.

SVA – SystemVerilog Assertions language.

tag – A name associated with a scope; characteristically used to link verification plan scopes with instance, coverage, or design unit scopes; similar to a user-defined attribute but with only a name, not a value.

temporal – A property of the relationship between two UCISDBs which indicates that one was generated at a different time than another. Note that a pair of UCISDBs may have both a temporal relationship, and as a result, also have a spatial relationship due to design changes.

test data record – Part of the UCISDB representing the test from which the UCISDB was originally created.

test instance – The attribute values associated with a single verification run.

test plan hierarchy – See **verification plan hierarchy**.

testbench coverage – Coverage data gathered on the code that implements the testbench.

toggle coverage - See coverage definitions: toggle coverage.

topology – A representation of the design and verification environment. Design and verification components on which a simulation is run; design element may be as small as a single module, or as large as multiple chips.

transition coverage - See coverage definitions: transition coverage.

UCIS – Unified Coverage Interoperability Standard. This standard.

UCISDB – Unified Coverage Interoperability Standard DataBase. A database which is accessed through the UCIS API that contains data related to verification of a hardware design. UCISDB data content can also be written in a text-based interchange format.

unified coverage database – See UCISDB

unique – A unique name represents exactly one coverage point within a single database.

use case – A scenario describing a series of actions taken by the user using the standard to achieve a specific goal.

user-defined attribute – A name/value pair explicitly added by the user, not part of the predefined UCISDB data model.

value-transition coverage - See coverage definitions: value-transition coverage.

verification component – A testbench, stimulus generator, assertion, functional coverage point, checker, or any other non-design element that is part of a topology. May produce, consume or affect changes in coverage data.

verification plan hierarchy (or **test plan hierarchy**) – A part of the UCIS data model whose nodes are linked to coverage or instance or design unit data structures for purposes of analyzing coverage in context of a verification plan.

verification run – An application of a verification tool to a topology, resulting in generation of a UCISDB. Evaluation of a topology which produces coverage information. The evaluation mechanism may be a simulation, formal verification, hardware emulation or other process that is capable of producing a UCISDB.

1.6 Introduction to UCIS

In design verification, **coverage metrics** are commonly used to measure effectiveness and to highlight verification shortcomings that require attention. A comprehensive verification methodology employs multiple verification processes, including (but not limited to) simulation, static design checking, functional formal verification, sequential equivalence checking, and/or emulation. Each verification process generates one or more coverage metrics which may be disjoint, overlapping or subsets. One of the key roles of the verification team is to gather, merge and interpret this multitude of coverage data to provide an overall assessment of verification quality and progress toward providing complete validation.

The **coverage database** (UCISDB) is a single repository of all coverage data from all verification processes. See Figure 1. The UCISDB is conceptually a single repository of all coverage data from all verification processes. It may, however, be one or more physical files in non-volatile storage. The verification processes are **coverage producers** that may access the data to produce reports, annotate the design description, update test/verification plans or other analysis functions as users require. Note that each of these analysis functions may manifest in one or more actual tools. In addition, some processes may manipulate the data and hence be both a consumer and a producer.

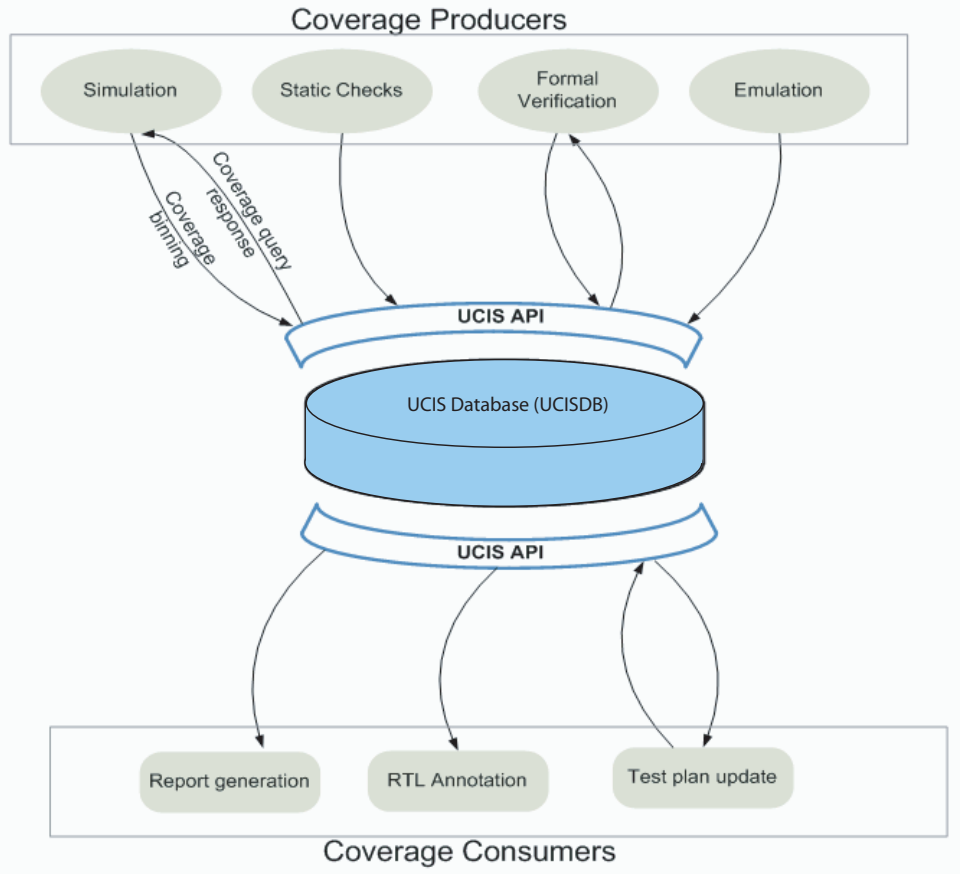

Figure 1—UCISDB

The standardized UCIS API layer between the coverage database and the coverage producers and coverage consumers enables interoperability.

As verification and analysis technologies improve, the individual coverage producers or consumers may change. Also, the database implementation may change, but the standardized API is intended to allow them to continue to operate together.

1.6.1 Motivation

The UCIS API may be used for the following:

— To import data into a coverage database or your verification tool from another source.

— To export data into a format not supported by a tool. For example, an SQL database or graphing package. Over time, some of these may be supported by your vendor, but if you cannot wait, you may use the API.

— To perform analysis of coverage data in a way not supported in any tool.

1.6.2 Unified view of coverage data

The primary goal of this standard is to enable data produced by multiple coverage producers to be semantically interpretable by multiple coverage consumers. Therefore, the content of this standard is a semantic information model of the coverage data, plus a set of standardized methods to access this data.

1.6.2.1 Bins and coverpoints

The core data held in a UCISDB (UCIS database) is a collection of counts or bins that have real-world meaning within the context of a design. The information model of the UCISDB expresses this semantic meaning for all the count values that have been recorded - essentially this makes the data held in every counter semantically interpretable. These counters may be referred to as **bins**, or **coveritems**, depending on context. They represent the occurrence of specific events that have been recorded. These events are taken from the diverse set comprising functional coverage, code coverage, assertion coverage, formal coverage and other user-defined purposes. The unifying concept is that they are integral numbers, decorated with sufficient semantic information to make them meaningful to verification engineers in the specific context of a known elaborated design.

Bin counts represent the raw data obtained from verification tool runs. Metrics derived from this raw data are also considered within both language definitions and this standard.

At the bin level, many derived metrics rely on a Boolean understanding of the counter - i.e. a coverage event either happened at least once, or it never happened. In theory, this would be expressed using a 1-bit counter. This is a basic coverage model used extensively to investigate coverage holes. Reducing coverage holes is a common goal for verification engineers.

The UCISDB default is to assume collection of more differentiated data using 32- or 64-bit counters. This still allows for metrics based on the 1-bit model, but also extends information capture for purposes of, for example, tracking redundancy in testing, or other analyses.

1.6.3 Coverage data from formal verification

Unified coverage may include coverage data from formal verification activities. Formal verification coverage may include whether or not an assertion was proven, and whether or not the proof was vacuous.

The UCISDB also tracks restrictions used for each formally-proven property, assumptions, initialization sequence, initial value abstractions, inserted cutpoints, and any other modifications made to achieve a proof.

1.6.4 Operations supported by the UCIS API

The following operations are supported by the UCIS API.

1.6.4.1 Defining coverage items

The UCIS API contains functions that enable you to define coverage items and test information. These functions are called at the start of a verification process to define the coverage model that will be used. For each coverage item, information is saved about the semantic type of the item. During the verification run, these coverage items will have their counts updated.

1.6.4.2 Defining user-defined coverage types

In addition to predefined types, the API enables you to create user-defined semantic types and user-defined operations. Functions are provided to set an external type **subtype**, list all registered subtypes, and remap subtypes to a different subtype number. Remapping enables you to merge data from multiple databases where different external coverage types may have been given the same subtype number. Figure 2 demonstrates the merging of simulation and formal verification data.

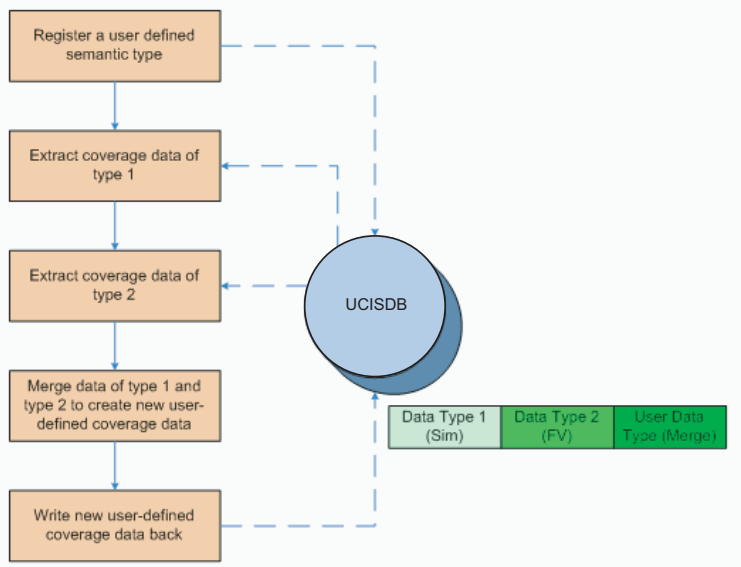

Figure 2—Defining user-defined coverage type

1.6.4.3 Reporting coverage data

There are numerous ways to access coverage data, including sorting coverage points and filtering coverage points. These operations may be qualified by name matching or hierarchy level matching to include wildcarding.

Here are some types of data reporting that may be implemented by applications using the UCIS API:

- Cumulative report: Summation or union of coverage data across temporal, spatial and heterogeneous data sets.
- Filtered report: Selected data based on user-specified criteria to extract coverage views from the full data set. Filtering can be done based on design or based on time. For example, only show coverage data generated within the last day.
- Ranked report: Rank verification processes according to highest improvement to incremental coverage. Can be used to generate minimum test sets to achieve set percentages of coverage.
- Unique coverage report: Identify tests that exercise coverage points not covered by other tests.
- Missing coverage report: Identify coverage points that have not been exercised.

Figure 3 illustrates accessing data for different reports.

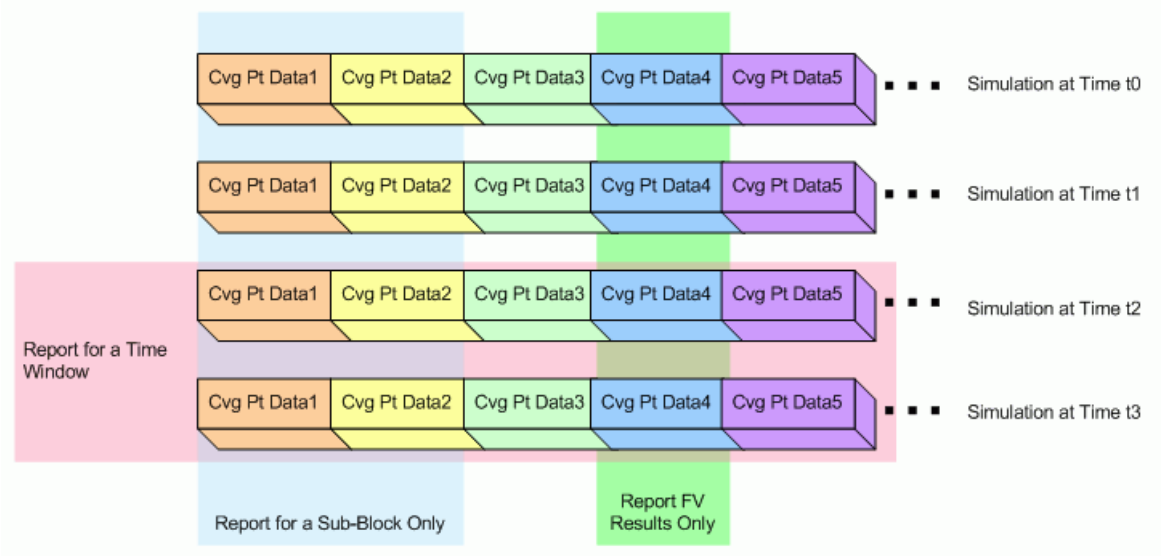

Figure 3—Types of coverage data reports

2 Use cases and data flow

This chapter contains the following sections:

2.1 Use cases

This section describes user goals supported by the UCISDB and the UCIS API. The UCISDB stores information for each uniquely-identifiable element of the topology exercised by a verification run. For example, in the case of a verilog device under test (DUT), information is uniquely stored for each module instance.

Here are a few supported use cases:

— ***Generating one UCISDB from one verification run with one coverage type enabled***

— ***Generating one UCISDB from one verification run with multiple coverage types enabled***

— ***Comparing UCISDBs from multiple verification runs across one or more coverage types to evaluate and rank coverage efficiency***

— ***Generating a global UCISDB by aggregating similar databases*** - Merging data for a coverage type which is present in all input databases involves traversing the structure in each database and merging the coverage data in a manner relevant to the coverage type. A key element of this case is that the topology which produced the databases to be merged is the same for the entire set of databases. The coverage data may be associated with verification components (including testbench), design components or interfaces.

Note: Data merging is not destructive; data is not deleted from any database.

— ***Generating a global UCISDB by aggregating dissimilar databases*** - Different verification components may be associated with a design object depending on the topology. Topologies may be completely disjoint from one another or have partial overlap. You may need to provide information to merge data in a meaningful way. Vendor tools may not support certain types of merging.

Different verification components for the same design object (e.g. instantiation of a module) from different databases cannot be merged, except if the verification components are not a source of coverage data; for example, two different testbenches for the same design object. The data may be stored separately for each verification component and associated with the design object. In this case, you may specify which data to include in the merged UCISDB. Interface coverage data is similar in that data from different topologies may cover different portions of the interface coverage space. However, an interface has a definable coverage space, into which all interface coverage data will fit independently of source.

Incremental changes to the design also challenge merging because they create different topologies, since one or more design elements are different. The specific coverage type(s) present in the databases determine whether the mechanics of merging can be executed, and it may be acceptable for vendor tools to prevent merging or require additional information from the user in some cases. In cases where a merge is allowed, the user must determine the value of the merged data.

In some cases, it may be possible to merge a database into another database in multiple locations. In these cases, you must specify one or more target locations.

— ***Extracting a UCISDB for an individual RTL component from a global UCISDB*** - Data may be extracted from a global UCISDB using either the instance method or module method. Instance walks the data structure and outputs everything that is present. Module merges instance data appropriately where instances share a common module. Through the API, you can specify subsets of instances that are merged to produce the module report.

Note: If you are merging subtrees of a design, a set of module instantiations may appear multiple times within the design. Consider a module A, which instantiates module B twice, as B0 and B1. Within a design, A and its children may be instantiated in multiple places. There may be other places where B is instantiated, but it is important for coverage purposes to know how B is covered when instantiated as a child of A separately from its behavior when it is instantiated on its own. It would be valuable to extract data with a module-type view, where the *module* is defined as a sub-hierarchy consisting of A, B0 and B1. In other words, all instances of this trio would be found and coverage data from each module would be merged (e.g. all "A" merged, all "B0" merged, all "B1" merged, but "B0" and "B1" kept unique from each other, and from all other "B" that are not children of "A" elsewhere in the design).

— ***Analyzing UCISDBs for unhit coverage points*** - A global UCISDB or an individual design/verification component UCISDB may have unhit (unreachable) coverage points. A subset of unreachable points can be specified in a form which can be read and applied by the API. Coverage point naming must be unchanged across multiple versions of the design. The API is not concerned with the specific classification of an unreachable point, either don't-care or predicted-unreachable, although you may specify this in order to yield meaningful warnings when a predicted-unreachable point is covered.

— ***Analyzing unhit data over time to track progress*** - Design implementations change over time as bugs are found and fixed and modifications are made to improve performance or meet timing constraints. However, not all of the design will change and continuity of coverage results will be desired. The UCIS enables you to merge databases over time.

2.1.1 Accessing data

The UCIS API is primarily used to access coverage data. There are two methods of accessing the data in a UCISDB:

— **Application Programming Interface** (**API**): The functions defined in the API (Chapter 8, “UCIS API Functions,”) can be used with a UCIS-compliant implementation to build or customize applications for analysis or reporting of the coverage data. The API provides an abstraction layer to the physical database file so that the same code can be ported to other UCIS-compliant implementations.

— **Interchange Format**: The contents of a UCISDB can also be accessed through an interchange format (Chapter 9, “XML Interchange Format”). A UCIS-compliant implementation can write out the contents of a UCISDB into the XML-based interchange format described in Chapter 9 of this standard. The coverage data can be accessed through any UCIS-compliant implementation or through other XML-based tools.

2.1.2 Merging data

The UCIS API can be used to implement the following types of coverage data merging operations:

— ***Merging coverage data across multiple runs of the same verification process*** (temporal merge) - This includes separate runs that may execute at the same time, but on different processors. The same DUT with the same coverage items are being exercised through different scenarios.

— ***Merging coverage data across different parts of a design*** (spatial merge) - For example, some verification may occur on sub-systems of a design independently. This data needs to be merged with verification on the full design or some overlapping section of the design to fully incorporate all coverage across all verification runs.

— ***Merging data from different verification processes*** (heterogeneous merge) - Coverage data items from different verification processes may be semantically equivalent, but more frequently, they are semantically different. They need to be stored within the UCISDB, but they should not be automatically merged within the database. If an analysis tool has an algorithm for merging heterogeneous data, it should submit a new data item with a semantic tag indicating merged data.

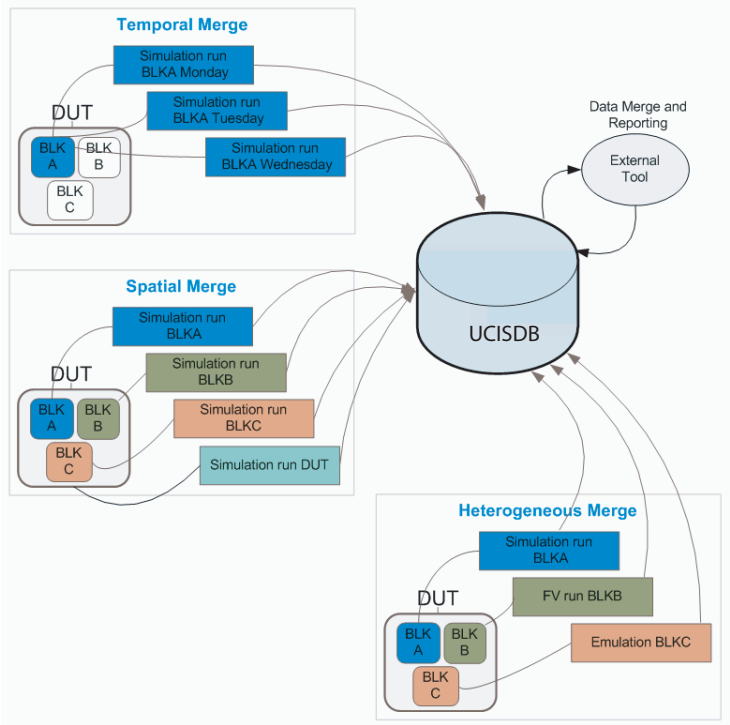

Figure 4—Merging data

2.2 Data flow

The UCIS covers not only the API for reading/writing data from/to a UCIS implementation, but also the interchange format that should be used to transport data between implementations.

The following sections identify a few data flow configurations supported by the UCIS. These are somewhat orthogonal to the use cases in that they do not necessarily imply any particular verification goal.

2.2.1 Browsing or updating a database

An application can use the UCIS API to open a database, selectively retrieve information from it, and selectively update information in the database. Since the access is selective, the application need not consider the full range of information stored in the database. In particular, it might focus on a particular subset of the coverage data available in the database. The XML interchange format can be used for browsing by dumping the database to an interchange format file, then traversing it using an XML reader. Changes may be made to the interchange format file and users may then load it back into a UCIS application to create a new coverage database.

2.2.2 Exporting or importing data to/from a database

An application can use the UCIS API to open a database and extract all of the data necessary to reconstruct the database in another database instance. Similarly, an application could use the UCIS API to create a new database and populate it with complete information. This is typically done as part of a verification run. This is most useful if you have built your own tools and want to integrate those tools into the coverage analysis flow of a given vendor.

You can export data from any database implementation to a standard interchange format and then import data from that standard interchange format to another database implementation.

2.2.3 Concurrent access to the same database

The API supports simultaneous access to a database from two or more tools. Either one writer or any number of readers can access a given database at any given time. You must specify, when a database is opened, whether the database is to be read or written.

2.2.4 Accessing multiple databases from the same vendor

The API supports simultaneous access to two or more databases. For example, a tool may open two or more UCIS-compliant databases from the same vendor at the same time to compare their contents or transfer data from one to the other directly without using an intermediate format.

Each API call can determine which database is to be addressed, based on the parameters that are passed to it. Each call has a parameter to indicate the DB it is to operate on.

3 UCISDB organization

This chapter provides an overview of the methods for accessing a UCISDB, and the components and structure that may be found in it. A UCISDB holds a collection of coverage data. Generally a single UCISDB will hold coverage data associated with a single design, although compliant implementations are not explicitly constrained in this way.

This chapter contains the following sections:

3.1 Physical representation and access

The physical representation of the data held by a UCISDB shall be implementation-dependent. The compliant API library that was used to construct the UCISDB shall provide the only access mechanism to this data required by this standard. It shall not be required for the data in a UCISDB to be accessible by any other UCIS-compliant API implementation than the implementation that created it; in general UCISDBs are not required to be interoperable with UCIS APIs from different vendors.

The physical representation may be volatile (fully held in computer memory) or non-volatile (on a computer storage medium).

Non-volatile UCISDB repositories may be accessed using serialized streaming methods. When streaming methods are used, the transfer of non-volatile data to or from memory is not an atomic operation. Data transfer is managed through a series of callbacks, each of which partially stages the data transport, such that the whole database need not be present in memory at any one time.

A compliant API implementation shall support both forms of access, but may have functionally different behavior between streaming and in-memory accesses. However, all data in a UCISDB shall be accessible via both methods, unless external factors such as maximum available memory or file permissions prevent this.

Single-call API routines are provided to translate between the in-memory and non-volatile storage.

— The ucis_Write() routine shall take a UCISDB handle which is associated with data in the memory owned by an application, and write it out to non-volatile file-system storage associated with a UCISDB name.
— The ucis_Open() routine shall operate on a non-volatile UCISDB identified by a UCISDB name, and transfer its entire contents into memory, associating it with a handle which is returned to the user. This action assumes that sufficient memory is available to hold the entire database.

An application may support access to multiple UCISDBs simultaneously. That is, an application may manage multiple database handles for any purpose, if each handle is associated with an independent UCISDB. This may include any mix of streaming or in-memory database accesses.

Concurrent access by multiple applications to a single UCISDB shall not be supported, with the exception of concurrent read-only accesses, which may be supported by some implementations.

3.2 Run-time handle-to-database resolution

A stored UCISDB shall be identified by a name represented as a string datatype. The API implementation shall map this UCISDB name to storage accessible from the operating system on which the application is running. This mapping is implementation-dependent.

An in-memory UCISDB shall be identified by an opaque handle. The API implementation shall initialize and maintain the association between the handle and the data. In this model, the handle may point to data that was imported from non-volatile storage, or it may be a handle to data that the application is in the process of constructing or generating.

3.2.1 Streaming Modes

In the streaming flows, the API shall operate through an opaque handle to the non-volatile storage, following a call to open a non-volatile UCISDB. Application usage of a streaming handle is more restricted than a fully in-memory handle. These restrictions are specified along with the API descriptions in Chapter 8, "UCIS API Functions", where applicable. The handle itself is non-persistent and is chosen by the implementation at run-time.

3.3 Data model object overview

The structures that form the content of a UCISDB fall into these categories:

- History Nodes to record UCISDB construction information.
- Scopes to create hierarchical structure that describes design and coverage components.
- Coveritems to hold the actual counts of recorded events. A coveritem is a leaf construct, it cannot have children, and is essentially an informational wrapper around an integral count.
- Source file tables.
- Data decoration elements such as attributes, tags and flags, which are associated with the composite objects history nodes, scopes, and coveritems.

Scopes are UCISDB hierarchy constructs, they are used to represent design units, instantiated design elements organized hierarchically, and coverage structure. The sense of a 'scope ' in the UCIS context is similar to a scope in the HDL design but the UCIS sense is extended to include coverage constructs not found in the HDL. This is covered further in Chapter 4, "Introduction to the UCIS Data Model". UCIS scopes shall be allowed to nest indefinitely, however the constructed hierarchies shall be tree structures. Specifically, loops and multiple parents shall not be allowed in the tree graph. Multiple root scopes (with a NULL parent) shall be allowed.

Coveritems hold counts, which in most cases are used to compute coverage. It is not necessary to retain UCISDB scope hierarchies that contain no coveritems; such hierarchies cannot contribute to coverage scoring.

It is a general observation that most of the information (and therefore physical storage required) in a UCISDB is under the scopes that record the elements of the instantiated design hierarchy. This is the data that is serialized in read-streaming mode. Other data, specifically the history nodes, design unit elements, and source file tables, shall always be available to the relevant API query routines for an open UCISDB, even when read-streaming.

3.3.1 The coverage model abstraction

Each coveritem in the UCISDB exists to resolve the meaning of what was counted by the tool that generated the data. An abstract theory of coverage considers that the collection of coverage data has the form:

@event if (condition) event_counter++

In the UCISDB model, the generalized event of interest might be a statement execution, variable value change, branch decision, finite state machine transition, clock edge, and so on. The event type is represented by the scope and coveritem typing system.

The condition that is tested relates to many circumstances that can affect whether coverage is collected, for example the variable value, start and destination FSM states, exclusion choices, and so on. There is a very large universe of coverage models to express. In the UCISDB, the scope hierarchy and coverage metric design describe the conditions under which the counters were incremented. Exclusions (objects that are ignored for coverage) are represented by flags.

The event_counter that is incrementing will ultimately become count data in a UCISDB. This count is decorated with enough information to identify it, categorize it, and perform useful post-collection processes on it. Thus a UCISDB can be visualized as a large collection of counters.

The precise semantics of resolving what each counter actually counted, are more fully explored in later chapters of this standard.

3.3.2 Information loss

Many coverage models involve information loss at collection. If discrimination is lost at collection it cannot, in the general case, be recovered from the UCISDB later. For example, if multiple variable values are collected in the same SystemVerilog bin, it shall not be possible to determine, from the UCISDB data, how many of the recorded counts were from a single one of those values.

3.3.3 Relating UCISDB data to the design HDL

This standard does not require that the design source code shall be fully represented in the UCISDB. In practice, this means that applications designed to relate the collected code coverage to the original source code will need access to that source code to provide this mapping.

3.4 Data associated with objects

UCIS composite objects (scopes, coveritems, and history nodes) have some predefined semantics and data fields, and an extensible attribute system. The predefined fields have specialized meanings relating to HDL coverage; the intent is to express the canonical coverage models. For example consider the specialized scope constructor routine ucis_CreateInstance(). This has an argument list as follows:

- **db** - The opaque handle to identify the UCISDB in which the instance is to be created.
- **parent** - The scope parent of the instance being created. In the general case the UCISDB parental relationship refers to the UCISDB hierarchy, not necessarily the instantiation hierarchy. In this case, the parent is in fact the HDL instantiation parent.
- **name** - Scopes must be named, the scope name is a primary key component. Generally an HDL scope will inherit its name from the HDL.
- **type** - The scope type is a primary key component and primary semantic differentiator.
- **fileinfo** - Every HDL construct has an anchor point within a design file, which is recorded.
- **du_scope** - Links the design unit of which this is an instance, to this instance. Note that design units are represented by UCISDB scopes; thus this argument is a scope handle.
- **source** - Records the design language, from a predefined set of supported languages.
- **weight** - Records the relative user perception of the importance of this scope, relative to its siblings, under a pari-mutuel system.
- **flags** - Further semantic annotation.

The semantics of the items on this list are recognizable to anyone familiar with HDL design.

3.4.1 Counts

Counts are at the heart of coverage collection. Coverage scoring algorithms are all based on the raw data expressed in coveritem counts.

Every coveritem in the UCISDB shall have an associated count datum (scopes and history nodes do not have built-in counts). The data model for this count shall be a composite entity of a type defined in the interface header file. The count type is composite to allow for different maximum counts. Counts are integral. Default 32 and 64 bit count resolutions are available, and an implementation may customize a byte vector representation of the counts. The semantic of count values is that they do not wrap. A maximum value for the count type indicates that either that value or a greater number of events were collected. If the number is not at the maximum value, it records precisely how many events were collected. For some coverage items where it only matters whether the item is covered or not, applications can use a maximum count of "1".

3.4.2 Attributes

Attributes are name-value pairs that may be associated with history nodes, scopes, or coveritems. Attributes shall be defined by a C type in the API interface header file.

Each attribute name shall be unique within the attribute list owned by a single construct. The name shall be a C string restricted to the alphanumeric, underscore '_', hyphen '-', space, colon ':', and '#' character set. It is not recommended that API applications construct user-defined attributes with names containing the '#' character; API implementations may use special handling for these names.

The attribute value component shall be a composite item, and shall be defined by a C type in the API interface header file.

The value shall consist of a type and value of that type. The memory block attribute shall also record a size that defines the number of bytes in the memory block. Memory blocks may contain any data values and NULL-termination shall not be assumed. Usage of the data in a memory block shall not be constrained, an implementation shall accept the MEMBLK bytes of data and return the same number of bytes containing the same values in the same order whenever the attribute is queried.

The defined set of types is:

- UCIS_ATTR_INT - 32-bit signed integer.
- UCIS_ATTR_FLOAT - single-precision floating point number.
- UCIS_ATTR_DOUBLE - double-precision floating point number.
- UCIS_ATTR_STRING - C-style NULL-terminated string.
- UCIS_ATTR_MEMBLK - block of bytes that can be used for any purpose.
- UCIS_ATTR_INT64 - 64-bit signed integer.

The attribute name shall carry the semantic of the attribute. The attribute system is extensible, and the extended set of user-defined attributes may have semantics that are only interpretable from knowledge external to this standard.

3.4.3 Flags

Scope and coveritem objects have flag fields. Flags are Boolean (1-bit) entities that are always present, and either true or false (values 1 or 0). UCIS access to flags is via a 32-bit flag field, the flag positions in this field are defined in the interface header file. Flag meanings may be further qualified by the typing of their parent.

3.4.4 Tags

Tags are strings that can be associated with composite objects. Tag names have the same restrictions as attribute names; but tags do not have values associated with them.

Tags are a grouping construct. Collections of similar or dissimilar objects may be associated by tagging. This provides a generalized extension to the inherent grouping forms such as typing and structural grouping.

A composite object may have an unrestricted number of tags, however each tag name shall not occur more than once on an object's tag list.

API routines are provided to iterate tag relationships, both to discover the list of items sharing a given tag, and to list the tags owned by an item.

The semantics of the tag purpose are carried by the tag name. As is the case for attributes, user-defined tags have semantics that can only be understood by external knowledge.

3.4.5 Weights, Goals, and Limits

Functional coverage UCISDB data models are derived from the SystemVerilog functional coverage model. This model defines weight and goal semantics - the UCISDB goal is equivalent to the SystemVerilog at_least concept. For the functional coverage scopes, the weight and goal fields are designed to transmit the SystemVerilog data model intent.

Generally, for other scopes and coveritems, weights express the importance of an item to the user. Goals express when coverage can be assumed to have been met. Limits express when an application can stop collecting coverage.

The default weight value is that all child scopes under a parent are weighted equally, and all coveritems under a scope are weighted equally. When a scope owns both child scopes and coveritems, the weight resolution is that the coveritems are evaluated together as a virtual scope of weight 1, before being combined with the child scopes, which may have user-defined weights, otherwise are also weighted 1 each.

The default goal value is 1, i.e. a coveritem is covered by any count other than 0. This can be adjusted upwards so that a coveritem is not covered until a higher count is reached. However for the purposes of scoring, coveritem coverage is Boolean, a coveritem is either covered or it is not covered.

Scope coverage is not Boolean, it is a real number between 0 and 100.

The default limit count is the maximum expressible in the count variable, which may be a 32-bit limit, a 64-bit limit, or a user-sized variable limit.

3.4.6 Properties

Properties are basic attributes of history nodes, scopes, coveritems or version object that have been pre-defined by UCIS and can be associated with these objects. Properties can be considered similar to pre-defined attributes because properties involve a name-value pair, except that value component of some properties can even be a database object thereby allowing association of 1:1 relationship between these objects. The set of property name components, the type of corresponding values and the type of corresponding object with which these name-value pairs can be associated, is fixed by the standard.

Some properties are read-only, for example the UCIS_STR_UNIQUE_ID property may be queried but not set. Some properties are in-memory only, for example the UCIS_INT_IS_MODIFIED property applies only to an in-memory database and is not stored in UCISDB. However, it is up to the implementation to enforce the *read-only* or *memory-only* feature and issue errors or warnings when violations occur.

3.4.7 Source file tables

Scope and coveritem elements within the UCISDB are inherently associated with source code locations. The standard supports a basic information model to identify a source code location comprising the source file, the line number, and the token number. This information is encapsulated in the ucisSourceInfoT type defined in the header file, and a variable of this type may be associated with every scope and coveritem in the UCISDB.

The storage of the filename strings is optimized so that the string is not stored for every copy of this variable. One or more indexed file tables is used to hold the filename strings instead, and the ucisSourceInfoT holds the file table number and index instead of holding the filename directly. The filename can be recovered from the appropriate file table.

The file tables shall be available after a database has been opened and shall remain available until it is closed, in both read-streaming and in-memory modes.

3.4.8 Error Handling

This standard provides an error-handler registration routine, ucis_RegisterErrorHandler(). The underlying model is that the API implementation shall assign a number, a severity, and a string to any condition thought to require user notification that is encountered within the implementation routines. The variable ucisErrorT, defined in the header file, is used to collate these three pieces of information. A variable of this type is then used as an argument to the registered error-handler routine.

The notifications and the conditions that trigger them are not standardized; nor are the precise error numbers or error strings. The error severities are taken from an enumerated type of three possible severities: informational, warning, and error. This enumerated type (ucisMsgSeverityT) is defined in the header file.

The error-handler registered by the application shall be triggered when the API implementation has a notification to pass to the application. The application error-handler routine may use the error number, error string, and severity in any way it chooses to determine what action to take.

4 Introduction to the UCIS Data Model

This chapter contains the following sections:

4.1 Introduction

The UCISDB is a general-purpose database schema for storing verification data. It is designed to be flexible and extensible, but there is also a requirement to store some forms of data in a universally-recognizable way.

The ability to recognize primary source objects and the coverage metrics on primary source objects in different UCISDB contexts is referred to as **Universal Object Recognition**. It is derived from a mapping of design-specific information to UCISDB objects. This ability underlies the stated UCIS goal of cross-tool and cross-vendor coverage reconciliation. Universal Object Recognition is restricted to that class of objects for which there is agreement as to a canonical set of user coverage data.

Each object in a UCISDB is assigned a **Unique Identifier** (Unique ID). When the guidelines for Universal Object Recognition are applicable to generate an object's Unique ID, this name is considered the **canonical name** of the object.

Flexibility and extensibility are made possible by the normative structural elements and primary key definitions. These aspects are addressed in Chapter 5, "Data Model Schema," which covers the normative data model schema.

Universal Object Recognition and the ability to assign meaningful structure and names to specific types of UCISDB objects is addressed in Chapter 6, "Data Models," which covers the advisory data model data.

The remainder of this chapter introduces some common data model concepts and definitions used in the schema and data analyses.

4.2 Target Coverage Flows

The UCIS API use-model is targeted at data warehousing and data mining coverage flows; it is not a full-featured transactional database. The API does not offer rollback or natively maintain data coherence and normalization as a relational database does. Limited random access is supported under certain circumstances. For example, when the database is small enough to be entirely represented in memory, the in-memory mode may be used for fast read and write access to the entire database. In read-streaming mode, the same degree of random read access within the instantiation hierarchy is available to current object and the ancestral objects of the current object only. Supporting objects may be available throughout the read-streaming process as noted. For example, access to history nodes is assumed to be available from any open database at any time.

The target UCIS flow is fast primary data collection in-memory by the simulator or formal tool, with export to disk of a complete UCISDB on completion. Multiple runs of the tool(s) result in multiple primary UCISDBs in the non-volatile data warehouse. A primary UCISDB is constructed from data that did not come from another UCISDB.

These primary UCISDBs can then be post-analyzed or combined (merged) by custom UCIS API applications.

4.3 History Nodes and Test Records

4.3.1 History Nodes

A **merged UCISDB** is a UCISDB constructed from multiple input UCISDBs, which may themselves be primary (see "Target Coverage Flows" on page 26) or merged UCISDBs. The merging algorithm is not fully specified by this standard, but the **History Node** records are designed to log the construction and merge history in an extensible way as new merge applications are developed.

The History Node records exist to track the historical construction processes for a UCISDB, and to record important attributes of the environment in effect when the resulting primary and combined databases were saved. Each history node is a container for a list of environmental attributes associated with the construction of one of the UCISDBs. Some history node attributes have been identified as universally useful and are predefined, but this is an area where new tools and processes continually force the need for extensions. Thus history nodes also accept user-defined attributes for which the only normative restriction is that they not be named with strings that have the UCIS: prefix. The attribute routines may be used to manage all the attributes owned by the record.

A history node therefore consists of:

- Record properties (logical name, physical filename, parent, and typing)
- Pre-defined attributes
- Optional user-defined attributes

The specialized history node to describe a primary coverage UCISDB is called a **Test Record**. The test record specialization (type UCIS_HISTORYNODE_TEST) is the history node with which coveritem counts may be associated.

Merged UCISDBs may retain the original history node and test records from their individual contributors but are not required to. They shall additionally add a history node for their own merge construction information.

Merging can be viewed as a hierarchical construction process that can be defined by a tree of merge operations. This tree shall be singular, acyclic, and unidirectional. A history node cannot be the child of more than one parent. It shall be an error to attempt to construct a cycle within this tree such that a history node record has a child hierarchy containing itself.

The history node for the final merge, i.e. the top-level history node in any database, has no parent.

An example of a multi-stage UCISDB merge process is shown in the diagram below.

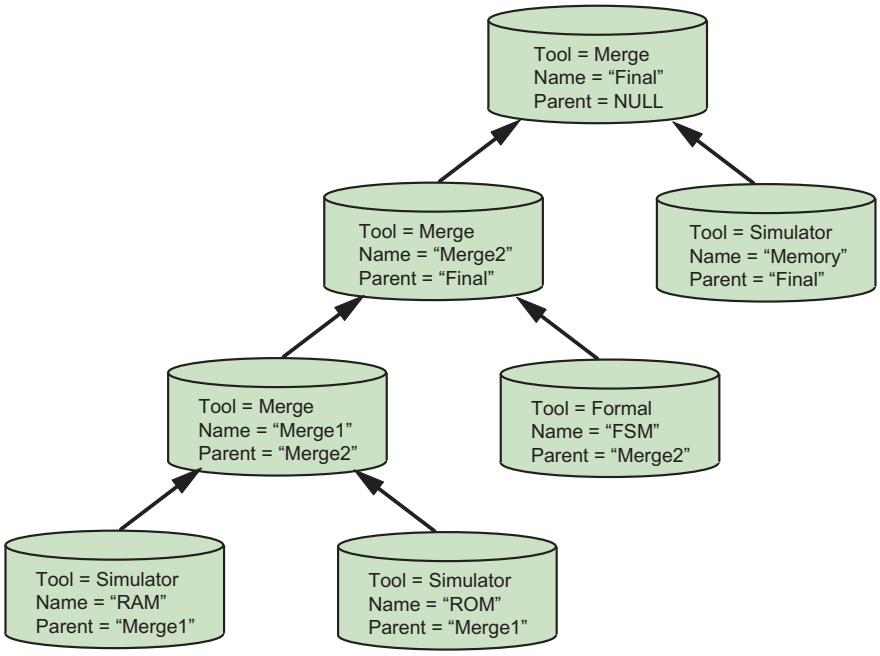

Figure 5—Multi-Stage UCISDB Merge

4.3.1.1 History Node Identification and Primary Keys

A UCIS database shall contain a set of one or more history nodes created by the UCIS API application. Each node is a container for a list of pre-defined attributes and user-defined attributes.

The history node constructor properties are a **logical name**, a **physical file or directory name**, a **parental history node**, and a **type**. Logical names shall not be NULL.

The logical name shall be a primary key within the entire hierarchy of history node records. This name may be user-assigned or implementation-assigned.

User-assigned names may be chosen to give meaning to the UCIS dataset that was constructed by the operation being recorded.

Both naming schemes shall support the uniqueness requirement.

All the logical names in the set of history nodes in a merged UCISDB shall be unique within the resulting database, and the API shall enforce this by generating an error if an attempt is made to construct a duplicate. It is recommended when choosing the history node logical name to encode something unique about the constructed UCIS database; and therefore the logical name of the top history node is in some sense the name of the database itself. The logical name character set is restricted to alphanumeric plus the underscore character (_) and matches are case-sensitive.

The precise nature of the physical name in the history node record shall be implementation-defined. It may be a file or directory name, or any piece of data that has some meaning for where the UCISDB physical storage was found at the time the record was created. There shall be no requirement for this field to continue to refer to real storage – obviously data storage of earlier records can be moved or deleted. Physical names are not required to be primary, but tool behavior when they are not is undefined; this case may represent a re-introduction of a file that has already been merged. The physical name character set is not restricted, except that it shall be a NULL-terminated ‘C’-style string.

The history node type is ucisHistoryNodeKindT. Standardized values are #defined in the standard header file. All values below decimal 1000 are reserved for UCIS use.

Predefined history node types are:

Table 4-2 — History node types

#define constant	Purpose
UCIS_HISTORYNODE_NONE	None or error.
UCIS_HISTORYNODE_MERGE	Construction was a merge operation.
UCIS_HISTORYNODE_TEST	Atomic test construction.

4.3.1.2 Description

The summary of the History Node UCISDB object is that it is conceptually the base class for an attribute collection container. There are some predefined extensions indicated by the typing, and users may extend the typing to construct their own extensions. Freedom from collision cannot be absolutely guaranteed but implementers may choose a base number at random within the 32-bit integer, above decimal 1000, to get a reasonable degree of uniqueness assurance.

4.3.1.3 Construction

The history node base class constructor ucis_CreateHistoryNode() shall require a logical name, a physical name, a type, and a parent history node pointer. Depending on the type, the physical name may be NULL. The parent may also be NULL – this indicates that the history node records the creation of the current database.

A specialized interface to the predefined attribute set owned by a test record history node (UCIS_HISTORYNODE_TEST) is also provided. This comprises a convenience wrapper struct (ucisTestDataT) for the pre-defined attribute collection associated with a single-run test, and the ucis_SetTestData()/ucis_GetTestData() routines to manage it. These are defined in Chapter 8, “UCIS API Functions” on page 109. As previously mentioned, optional user attributes may also be attached to a history node, these are managed by the attribute routines.

The type=UCIS_HISTORYNODE_MERGE requires a non-NULL physical name.

4.3.1.4 Pre-defined Attribute (Property) List

The test record attributes may be accessed as a list from a test record with ucis_SetTestData()/ucis_GetTestData(). All attributes may also be accessed individually from any record type with the routines.

Table 4-3 — Pre-defined attributes

Enumeration Constant	Notes	Test Record	Test Plan	Merge
UCIS_INT_TEST_STATUS	Enumerated values indicating test run result (e.g., fail, pass) or errors in merging.	X		
UCIS_REAL_TEST_SIMTIME	Total time simulated by the test. See also TIMEUNIT.	X		
UCIS_STR_TEST_TIMEUNIT	See also SIMTIME.	X		
UCIS_STR_HIST_RUNCWD	The directory in which the construction tool was executed to create this node.	X	X	X
UCIS_REAL_HIST_CPUTIME	CPU time for completion of the test.	X	X	X
UCIS_STR_TEST_SEED	Randomization seed for the test.	X		
UCIS_STR_HIST_CMDLINE	Invocation name of construction tool.	X	X	X
UCIS_STR_TEST_SIMARGS	Command line arguments to construction tool.	X	X	X
UCIS_STR_COMMENT	User-defined comment.	X	X	X
UCIS_INT_TEST_COMPULSORY	Boolean. Is it a must run test?	X		
UCIS_STR_TEST_DATE	Wall clock time at the start of the tool operation.	X		
UCIS_STR_TEST_USERNAME	Who ran the tool.	X		
UCIS_REAL_TEST_COST	Implementation-defined.	X		
UCIS_STR_HIST_TOOLCATEGORY	The attribute may take any string value except with the UCIS: prefix. Extensions are allowed. Pre-defined values for this string are: — `UCIS:Simulator` — `UCIS:Formal` — `UCIS:Analog` — `UCIS:Emulator` — `UCIS:Merge` — `UCIS:Comparison`	X	X	X

4.3.1.5 Search and Query

The ucis_HistoryIterate() and ucis_HistoryScan() routines are used to iterate for the history nodes in a database.

The ucis_GetStringProperty() routine is used to recover the constructor logical name and physical name (UCIS_STR_HIST_LOG_NAME and UCIS_STR_HIST_PHYS_NAME respectively).

The type information is accessed with the ucis_GetHistoryKind() macro.

The ucis_GetHistoryNodeParent() routine is used to recover the parent of a history node.

The ucis_GetTestData() routine is used to query the pre-defined attribute set. The generalized attribute routines are used to query any additional user-defined attributes.

4.3.1.6 Destruction

History nodes can be removed from the data base with the ucis_RemoveHistoryNode() routine. All children of the node are also removed; however references to the removed node(s) may remain in the database after this operation, therefore it must be used with care.

4.3.2 Universal Object Recognition and History Nodes

Unlike design scopes and coveritems across UCISDBs, history nodes across UCISDBs are expected to be different, and shall therefore have different logical names within a single UCISDB. Combining the results from two identical tests is not an interesting use-mode.

The logical name primary keys for history records in a merge attempt are required to be unique between the UCISDBs, so that the primary key requirement is not violated after the merge.

It is possible for a merge application to be presented with multiple UCISDBs that have duplicate history records. The solution to this problem lies in the merge application domain and is not further discussed here. There is no attempt made here to infer user intent when two UCISDBs with identical logical names are submitted to a merge operation, either at the same level, or due to reappearance of similar names at higher levels of merge.

4.3.3 History Node Records in Read Streaming Mode

In read-streaming mode the HDL Scope hierarchy is only partially in memory at any given time. However there is related, non-design data that is used by the HDL Scopes that is always kept in memory during read streaming. The history node records are in this category, a compliant implementation shall make them available after successful completion of the UCISDB open process, and until the UCISDB is closed. The history node callbacks (UCIS_REASON_MERGEHISTORY) shall occur before the HDL scope callbacks when read streaming.

4.3.4 API Routines

There are API commands to manage the list of history nodes owned by a UCISDB. These are:

```
ucisHistoryNodeT ucis_CreateHistoryNode(
    ucisT db,
    ucisHistoryNodeT parent,
    char* logicalname,
    char* physicalname,
    ucisHistoryNodeKindT kind
    );

typedef struct {                  // Pre-defined Attribute
    ucisTestStatusT teststatus,   //
    double simtime,               //
    const char* timeunit,         //
    const char* runcwd,           //
    double cputime,               //
    const char* seed,             //
    const char* cmd,              //
    const char* args,             //
    int compulsory,               //
    const char* date,             //
    const char* username,         //
    double cost,                  //
    const char* toolcategory      //

} ucisTestDataT;
```

Note: The precise meaning of Boolean compulsory flag depends on the implementation. It is intended to indicate a required test in the verification flow in some sense.

Note: ucisTestStatusT type is an enumerated type. When accessed directly, the returned value can be cast to the type defined below.

```
typedef enum {
    UCIS_TESTSTATUS_OK,
    UCIS_TESTSTATUS_WARNING,    /* test warning ($warning called) */
    UCIS_TESTSTATUS_ERROR,      /* test error ($error called) */
    UCIS_TESTSTATUS_FATAL,      /* fatal test error ($fatal called) */
    UCIS_TESTSTATUS_MISSING,    /* test not run yet */
    UCIS_TESTSTATUS_MERGE_ERROR /* testdata record was merged with
                                   inconsistent data values */
} ucisTestStatusT;

int ucis_SetTestData(
ucisT db,
ucisHistoryNodeT testhistorynode,
ucisTestDataT* testdata);

int ucis_GetTestData( ucisT db,
ucisHistoryNodeT testhistorynode,
ucisTestDataT* testdata);

ucisIteratorT ucis_HistoryIterate (
ucisT db,
ucisHistoryNodeT historynode,
ucisHistoryNodeKindT kind);

ucisHistoryT ucis_HistoryScan (
```

```
    ucisT db,
    ucisIteratorT iterator);

    ucisHistoryNodeT ucis_GetHistoryNodeParent(
        ucisT                   db,
        ucisHistoryNodeT        childnode);


    int ucis_RemoveHistoryNode (
        ucisT                   db,
        ucisHistoryNodeT        historynode);
```

Nodes that are ucisHistoryNodeT type objects may be used in the attribute and tag management routines where scope and coveritem objects are used:

```
ucis_AttrAdd()
ucis_AttrMatch()
ucis_AttrNext()
ucis_AttrRemove()
ucis_AddObjTag()
ucis_RemoveObjTag()
```

4.4 Schema versus Data

This standard guarantees the primary key behavior defined in Chapter 5, "Data Model Schema", but explicitly *does not guarantee universal object recognition in all cases*. There are several reasons why this second guarantee is not possible.

Firstly, name-assignment is a database-entry activity where the name data is user-supplied and assigned meaning by user-intent. The standard codifies recommendations which, if followed by the data generation tools, support universal object recognition, however the API cannot prevent data generation tools entering non-compliant data. Another way to state this is that there is no way for an API implementation to verify that user-intent has been correctly represented.

Secondly, there are some items for which no recommendation is currently available. These are discussed where applicable.

Thirdly, the standard supports extended data typing within the existing UCISDB definitions. No naming recommendations are available for extensions if they are not defined below.

Where data is compliant with the Universal Object Recognition schemes described in this standard, the data generation application may set one of the UCIS_UOR_SAFE* flags on the applicable scopes/coveritems. See "The Universal Object Recognition Compliance Flags" on page 57.

4.5 Read-streaming model versus in-memory model

The full data model is supported when the UCISDB data is in memory. Some aspects of this data model are denormalized. For example, the availability of rolled-up coverage data under DU scopes, which is a composite (union) of instance data. Data is denormalized in the sense that a single piece of coverage information can affect different data queries in this case; the DU data is derived from data in a different location. An in-memory API can hide the implementation of this denormalization.

When UCIS databases are stored, the API implementation may make optimization decisions as to whether to store only normalized or additional derived (denormalized) data. In read-streaming mode, these optimizations may be exposed, such that the read-streaming application may need to recompute missing data.

4.6 Diagram conventions

Diagram conventions are used for both database schema and database data examples as follows:

- blue represents schema examples
- orange represents data examples
- rounded rectangular boxes represent scope objects
- square boxes represent coveritems
- Multiple objects may also be contained in a lighter colored box to indicate common behavior. If a grouping box is named, the name represents the class of objects and is italicized. When an italicized class name is used in a schema drawing the intent is to show that all objects within that class share similar behavior, not that there is an object of that name that can be constructed in a UCIS database.
- The macro (#define) definitions of the scope and coveritem types such as UCIS_MODULE and UCIS_FSMBIN from the normative ucis.h file are used to indicate scope or coveritem types.
- An unbroken line with an arrow indicates the hierarchical has-a relationship. A scope of the type shown can have children of the type the arrow points to. The multiplicity restriction is indicated as text near the arrow, for example [0:n] means that the parental scope may contain 0, 1 or many of the child type. The notation 1 means that the parental type should contain exactly 1.
- A broken line with an arrow symbol represents any generic form of linked scope relationship. For example, an instance scope is linked to its design unit scope.

For clarity, other information may be omitted in the diagrams. Figure 6 illustrates schema and data diagrams:

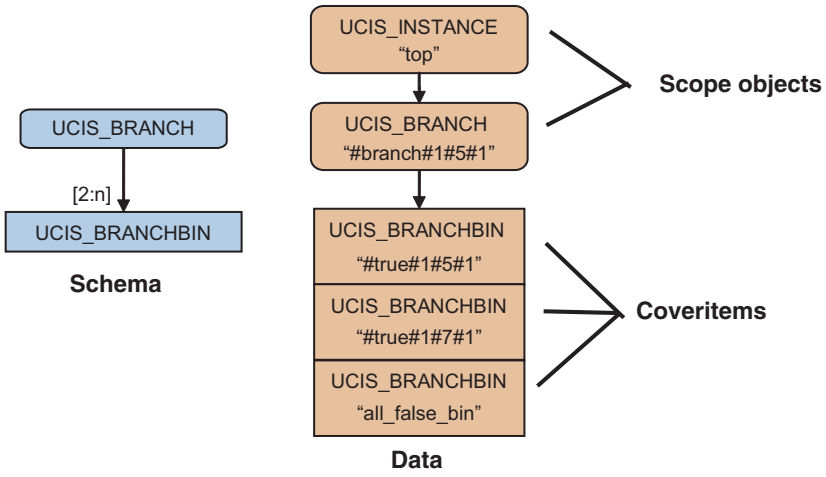

Figure 6—Schema and Data Diagrams

4.6.1 Unique ID List

Unique IDs may be listed for the example diagrams. If so, they are listed in diagram order from top to bottom, and left to right. The full or relative Unique ID form may be used, depending on whether the example shows a tree snippet from a top-level object or not. Whichever form is used, all objects on the diagram will be fully named from the top of the parental hierarchy that is visible in the associated diagram.

These listings assume that the library name is work and that the path separator is the / (forward slash) character. The enclosing double-quote characters are for clarity and are not actually part of the name. For example, the following are the Unique IDs for Figure 6.

```
/4:top
/4:top/1:#branch#1#5#1
/4:top/1:#branch#1#5#1/:6:#true#1#5#1#
/4:top/1:#branch#1#5#1/:6:#true#1#7#1#
/4:top/1:#branch#1#5#1/:6:all_false_bin
```

4.7 Coverage Object Naming

Coverage object naming is a major component of Universal Object Recognition. This section is a general introduction to the topic; see the section, "Metrics" on page 40 and the chapter, "Data Model Schema" on page 49 for specific type-based naming recommendations.

Local scope and coveritem names are database components specified as arguments to the API constructor routines, and returned by reference from API query routines. The responsibility for constructing the data in the local name string rests with the application that calls the API constructor routine.

Note: Names presented to the constructor routines must follow the escape syntax rules appropriate to the path separator currently in effect.

4.7.1 Name sources

Several sources of coverage object name content are identified:

- User-supplied names targeting UCISDB objects. For example, a user-defined covergroup instance name.
- User-supplied names for HDL objects, re-purposed to name UCISDB objects. For example, class, variable, or entity names.
- Standardized descriptive names. For example, elsif or 1->0.
- Items generated from the HDL that do not appear as names in the HDL. For example, line or item numbers, or expressions.
- Implementation-assigned names.
- Compound names comprising components of one or more of the previous types.

The first four of these naming methods are designed to support universal object recognition by having a clearly-defined relationship back to the source HDL.

However, universal naming is not possible in all cases. A class of names is defined to be implementation-assigned. In some cases, this name property is inherited from the source HDL language, therefore the UCIS does not determine the name.

No restriction is placed on implementation-assigned names beyond the normative requirements. Implementations may choose to use constructed names with partially-meaningful components, or completely generated names. Data consumers may not make any assumptions about the repeatability or predictability of implementation-assigned names based on this standard.

4.7.2 Name templates

Name templates are specified using the following syntax:

— Each definition is introduced with the phrase:

```
<name> :==
```

where <name> is the name type being defined. For example, toggle-name

— Name choices are indicated by [choice1 | choice2 | choice3]. The '[', '|' and ']' characters are optional in the definitions. If present, they are not part of the constructed name.

— Repetitive components are indicated by {repeat-component}. The '{' and '}' characters are optional in the definitions, if present, they are not part of the constructed name.

The syntactic pieces that are used to build the name follow the conventions defined below. The definitional form may contain white space. If so, the white space is omitted in the constructed name. Any of the following types of name portions may be specified in any order:

— user-source-hdl – Indicated in plain font, a name portion of this type is a user-supplied name from the original HDL or other user-input source. If the name is preceded by a standard number, the name identifier is taken from the standard definition, for example 1800bin_identifier is the bin_identifier SystemVerilog BNF component.

— **textualized-name** – Indicated in red, this name portion is a pre-defined string that is part of the UCISDB definition, for example # is used to separate integers.

— **integer** – Indicated in bold, this name portion is a positive decimal integer without signage.

— **binary-with-dontcare –** Indicated in **bold**, this name portion is a binary string of arbitrary length containing the characters 0, 1 and – (for don't care) representing the bits in a variable.

— *implementation-assigned* – Indicated by the *italics*, this name portion is assigned by the coverage generation tool and is not reliable for universal object recognition.

— ***metric-name, metric-coveritem-name –*** This is a special class of names, indicating that scope and coveritem names are derived from a metric definition.

4.7.3 Component representation alternate form

See the section, "Primary key design for scopes and coveritems" on page 50 for a full description of type representation within unique IDs.

Each 64-bit one-hot type from the scope typing set and the coverage typing set is given a string form for component representation in unique ID forms. This form uses the 0-referenced bit position of the one-hot bit expressed as an unsigned decimal integer. The scope type representation has a ':' (colon) character suffix and the coveritem form has a : prefix and suffix. An example of a scope type component representation would be 22: and a coveritem type :16:.

4.7.4 Aliases

UCIS supports the property enum UCIS_STR_UNIQUE_ID_ALIAS to allow aliases to be associated with elements of the database. It is the responsibility of the data generation application to set its value appropriately, if it chooses to save aliases in the database.

4.8 Source-derived name components

4.8.1 General Rules

File-numbers and line-numbers used as naming components are represented as positive decimal integers without signage.

It is necessary to establish general rules for determining the anchor-point for the purposes of file-number and line-number calculation. This is because multi-word tokens may span multiple lines or even multiple files.

The first rule is that abstract tokenization is applied to the HDL language. This is described in the relevant sections below, however the underlying principle is to relate specific language tokens to abstract coverage definitions. For example, branching syntaxes are assumed to comprise the *initiatory*, *true* and *default* tokens used to construct coverage scopes (see "Branch Coverage" on page 74).

When these tokens have been identified in a language, the second rule is that the final word in the token is the anchor point. For single-word tokens this is unnecessarily specific, it is not possible for the word if to span multiple lines or files. However, the token *else if* can do so, and this definition enforces the rule that it is the position of the if in the else if that is used for naming the associated coverage. Blocking and precedence tokens and punctuation are excluded ('begin', 'then', '{', '(', ';' and so on).

Expression and condition coverage covers expression values but is actualized by the presence of the top-operator of the expression that is being modeled for coverage, and is determined by respective language' precedence and associativity rules. See "Condition and Expression Coverage" on page 86.

For example

```
if      // line 1
( (a == // line 2
b)      // line 3
)       // line 4
c       // line 5
=       // line 6
d | e;  // line 7
```

The 'if' token on line 1 potentially initiates collection of both **branch** and **condition** coverage. It is an initiatory token for **branching** and the line number of '1' will be used in naming this branch.

The (a == b) expression is identified by line 2, which contains the '==' operator, even though the expression itself spans lines 2, 3, and 4. This makes the file-number and line-number determination unambiguous for the purposes of universal object recognition. UCIS applications may store more information to resolve the actual location of the expression text, but it is universally identified as described.

A similar analysis is performed for expression coverage where the expression may or may not be embedded in an assignment statement. The example has an assignment statement which spans lines 5, 6 and 7. The line number identifying expression coverage on 'd | e' is line 7 – the line associated with the '|' operator.

4.8.1.1 File-numbers

Both code coverage scope names and code coverage coveritem names may use file data to disambiguate potential collisions. For example, a design unit may be defined using two include files, each of which has statements on the same specific line number. The multi-file compilation unit design style is another way that multiple files can contribute to the same design unit scope.

A code coverage scope or coveritem constructed under a design unit scope cannot therefore use only line and token number – it must also recognize that there may be multiple files contributing to the design unit. Similarly, the coveritems under the code coverage scope may need to identify tokens that were found in different source files.

The file-number used for this purpose is defined to be the *1-referenced count of files contributing to a design unit definition, in the order in which the compiler encountered them*. The stability and cross-vendor guarantee on this is undefined in some complex cases, however for simpler cases such as a SystemVerilog module defined in a single file, universal object recognition is achievable. In the single-file case the file-number is 1 for all the module code coverage.

Consider a module defined using an include style, for example:

```
module my_module;
`include my_module_for_today.v
endmodule
```

In this case there are two files, 1 and 2. File 1 is the file containing the code above. File 2 is the included file, in which the file-number for the purposes of code coverage name construction would be 2.

For a multi-file module, compiled using a command line syntax similar to

```
verilog-compiler first_half_of_my_module.v second_half_of_my_module.v
```

the file-numbers would be 1 and 2 in command-line order. This order is meaningful on the command line and therefore cross-vendor reliable.

File-numbers are 1-referenced to the parental scope design unit source, not the UCISDB. A change to the source code for the design unit is potentially a data-invalidation event for this scheme as it is for any code-coverage scheme.

The file-number is not intended to function as the back-link to the source file. Its function is as a naming component only. Source file back-tracing information is available from database fields which are maintained for that purpose, specifically the source information field. The difference between these two information models is that the source information field is intelligently managed when databases are combined. The embedded file-number in the constructed name is not revisited or regenerated in any way after its initial determination at compile time.

4.8.1.2 Line-numbers

Line numbers are counted within each HDL source file, and are 1-referenced. A potential data-invalidation event for this datum is any change within the source file, including but not limited to comment changes or changes outside the design unit definition code.

4.8.2 Basic blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

4.8.3 Variable names as naming components

Variables and parts of variables are identified by language-appropriate string names with no unnecessary white space. This category includes top-level variable names, single or multi-indexing representation, and class and structure member references. Variable parental hierarchy is not included in the name; it is assumed to be implicit in the UCISDB parental scoping. No method for placing UCISDB data for a hierarchical reference under the referring scope is defined; such data is assumed to be collected under the parental scope of the variable itself.

4.8.4 Expressions as naming components

UCIS does not support expressions as a naming component for universal recognition because expression decompilation is implementation dependent. If needed, implementations can use complex expressions as naming components as-written in the source code with the following exceptions:

— All comment syntax is removed including embedded carriage return/line feeds but excluding comment-terminating carriage return/line feeds.

— Multiple contiguous white-space characters, including tabs, carriage return and line feed characters, are collapsed to a single space.

— Enclosing precedence tokens and white-space characters are removed.

— File-spanning expressions are treated as if the file-break was not present and the source text lines were contiguous; the implicit line-break character then follows white-space rules.

For example, the following expression, used as a conditional, has two input integers, two input Booleans, and a lot of extraneous precedence tokens and white spaces:

```
if ((((my_int1==(my_int2)) ? (((my_bool1))) : ((my_bool2)))))
```

Note: This example is not a recommended coding style.

In this expression, the input Boolean naming for input-contribution metrics has three naming components. Regarding the Boolean result of the equality test on my_int1 and my_int2, my_int1 and my_int2 are not individually acting as Booleans. Therefore, the equality test, not the input to the equality test, is the Boolean input to the metric.

The multiple white spaces in this expression are collapsed to a single space and external precedence and space characters are stripped so that the naming component string is:

```
"my_int1== (my_int2)"
```

The input Boolean variable naming components are my_bool1 and my_bool2.

4.9 Metrics

4.9.1 Introduction

In this context, a metric is defined as *the system of rules that defines how observed data for a multi-valued entity are assigned to bins*. Thus a metric defines a collection of associated coveritems owned by a scope.

Metrics are not a dynamic representation of the data; they define a post-collection data set and are not re-configurable after collection. In most cases metric collection involves calculated information loss, and the information lost during collection, is not re-constructable in the general case.

A metric scope containing the coveritem collection is identified by its name, as are its coveritems. A number of metrics have been standardized and are described below. The standardized metric names and structures below them are not required to be collected, but metric scopes that conform to the advisory descriptions below support universal object recognition.

In the general case, metric design is an open field and the UCISDB structure schema designed for the collection of metric data is intentionally left open to new types of metric. The only restriction on new metric scope types is that they shall not be named with a name that starts with the UCIS: component.

The coverage model allows for multiple different metric scopes under a parental scope. Each of these child scopes is named for its metric and contains the appropriate collection of coveritems.

Metric scope *types* reflect their target usage, for example UCIS_EXPR, UCIS_COND or UCIS_TOGGLE scopes may all be metric scopes. Metric coveritem types reflect their parental scope type, for example UCIS_TOGGLEBIN.

It should be noted that the same metric names are allowed to be used under different metric scopes. For example, UCIS:FULL as a metric name can be used under both metric scopes UCIS_COND and UCIS_EXPR.

The information model for a metric comprises these components:

- The name of the metric
- The data type or types for which the metric is targeted
- The naming rules by which the metric coveritems can be identified
- The rules by which observed data values are assigned to the coveritems
- Excluded-value rules, if any

Metrics define shaped collections of data. A metric explicitly does not define or record the sampling rules for data collection. Metrics may or may not imply a fixed number of coveritems. Conceptually, metrics are similar to pre-defined SystemVerilog covergroup bin collections.

For example, consider an 8-bit integer type. A simple metric can be defined where every integer value is given its own coveritem. This would require 256 separate coveritems to track every value. If during an actual simulation an HDL entity took the two values 17 and 96, the calculated coverage for this metric on the entity is 2/256.

An alternate metric, similar to the SystemVerilog covergroup auto-binning system, might predetermine that a total of 8 coveritems would be sufficient, and that values would be allocated equally. In this case, 32 values are assigned to each coveritem (256/8). The 17 and 96 values would be binned into two separate coveritems and the metric coverage in this case is 2/8. The example demonstrates how different metrics can result in significantly different coverage numbers.

4.9.2 Metric Naming Model

Standardized metric scopes are named with the UCIS: prefix, followed by the specific metric name.

UCIS data generation applications may choose to uses names in this prefix form where the prefix is a tool name, vendor name, or stock ticker symbol associated with the tool or tool vendor, followed by a semi-colon character, followed by a distinguishing metric name. This form of metric name is assumed to be compatible with Universal Object Recognition. See "The Universal Object Recognition Compliance Flags" on page 57).

```
metric-name :== [ UCIS: metric-identifier | vendor-or-tool-identifier : metric-identifier]
```

4.9.3 Metric criteria

Language functional coverage models allow a great deal of control over bin design that is not easily available to automated code coverage collection. Metrics suitable for code coverage collection can be assessed by the following criteria:

- General usefulness – is this criterion meaningful and useful across a wide range of HDL data?
- Practicality – is the number of coveritems necessary to collect the metric practical?
- Universality – is it possible to describe the metric so that multiple tools infer exactly the same structures?

In practice, none of these criteria are binary determinations, there is no perfect metric.

4.9.4 Metric excluded value coveritems

By definition, a metric defines an inclusive list of expected values, and assigns them to coveritems. Therefore it is possible for excluded values to exist for any metric, i.e. values that the variable might take but for which no coveritem has been defined. Excluded values exist in three identified categories: default, ignored, and illegal. Coveritem type definitions exist to identify all three categories and zero or one instance of any of these coveritem types is valid in a metric scope. If present under a metric scope, these coveritems are named for their types:

- Coveritems of type UCIS_ILLEGALBIN are named #illegal_bin#
- Coveritems of type UCIS_IGNOREBIN are named #ignore_bin#
- Coveritems of type UCIS_DEFAULTBIN are named #default_bin#

This naming model is clearly redundant but is necessary to support unique IDs. Note that there are other naming options for these bins in other contexts.

Note: These bin types may be differentiated by coverage aggregation behavior.

4.9.5 Metric Definitions

Three broad categories of metric are currently recognized in this standard. These are input-contribution metrics, expression value coverage metrics and transition coverage metrics.

Note: These metrics apply only to code coverage metrics. SystemVerilog and 'e' covergroups "Covergroups" on page 67 and assertions "Assertion Coverage (SVA/PSL assert)" on page 103 have their own definitions, which are described below.

4.9.5.1 Input-contribution coverage metrics

Input-contribution metrics analyze Boolean-result expressions by their top-level Boolean inputs.

Consider the incomplete code-snippet

```
if (a & (b | c))
```

This is a conditional use of a Boolean expression with three inputs. The three inputs act as Booleans even if they are not, following language rules. Note that the inputs could also be Boolean-result expressions such as the equality test (integer_1 == integer_2). An input-contribution metric defines a collection shape for this expression based on the three input Boolean values and optionally the result as well. The logic above can be expressed as a 3-input truth table and input-contribution metrics can be considered to be metrics related to this truth table.

Input contribution metrics may be completely flattened for a top-level expression, or may be collected for sub-scopes within an expression hierarchy. The ordering of input values used to construct bin names is the lexical order of the operands. The property enum UCIS_STR_EXPR_TERMS may also be attached to a scope to annotate the input order. The syntax of the string value of this property is a list of operands separated by the '#' character.

Metrics that share a binning algorithm but differ in the input list are distinguished by the **_FLAT** and **_HIER** suffixes in the metric name. The _FLAT family of metrics monitors all Boolean inputs at a single level. The _HIER family of metrics monitors that Boolean inputs through the sub expression hierarchy. If the prefixes are common then the underlying binning algorithm is the same. The bins may be different because the inputs may be different due to the differences in breaking down the expression.

4.9.5.1.1 UCIS:FULL

The **UCIS:FULL** input-contribution metric is defined *to require a separate coveritem for every possible combination of input Boolean values*. In the 3-input example case this implies 8 coveritems. Coveritem names are defined to be the binary string representation of the input values, ordered such that the binary msb is the left-most input identifier in the source code, and the other inputs are positioned in source-code order left-to-right.

In the example the input order for UCIS:FULL_FLAT is abc and the actual coveritem names are 000, 001, 010, 011, 100, 101, 110, and 111.

For UCIS:FULL_HIER, the top-level inputs are "a" and "b | c", with coveritem names "00", "01", "10", and "11". UCIS:FULL_HIER monitors the subexpression "b | c" separately, with the same coveritem names.

This metric scores relatively highly on general usefulness and very highly on universality. The number of coveritems is exponential with respect to the number of inputs monitored at a given level, which reduces the practicality of the metric. Metric collection may adversely affect simulation performance, and tools that implement this metric may apply some form of restriction on expressions that are covered by it.

```
metric-name :== UCIS:FULL_FLAT | UCIS:FULL_HIER
full-bin-name :== 0 | 1 { 0 | 1 }
```

4.9.5.1.2 UCIS:BITWISE

The **UCIS:BITWISE** metric is defined *to require a coveritem for each input term seen at 0 and 1*.

This metric causes the construction of exactly 2n coveritems for n inputs. Coveritems are named in the ordered binary string form as for UCIS:FULL, with the addition of the '-' (dash) character for don't care. The form of each name will be a 1 or 0 in one input position with all the remaining inputs as don't care. For example, if there are two inputs, there will be a four bin set "1-", "0-", "-1", and "-0".

Note that this metric's bins can represent sets of overlapping values. For example, both "1-" and "-1" may be satisfied by a single execution of the monitored expression.

The metric scores averagely on usefulness, high on practicality, and high on universality.

```
metric-name :== UCIS:BITWISE_FLAT | UCIS:BITWISE_HIER
bitwise-bin-name :== 0 | 1 | - { 0 | 1 | - }
```

4.9.5.1.3 UCIS:TRUEUDP

The **UCIS:TRUEUDP** metric is defined *to require a coveritem for each of a full set of positive logic expression factors of the entity, with default optimistic assignment*.

This is an example of a metric that depends not only on the expression inputs but knowledge of the expression result. Consider the example (a & (b | c)). The set of (abc) input vectors for which the expression evaluates to a positive result is (110, 101, 111). This 3-coveritem set therefore meets the definition criterion. However alternate sets are also possible, for example (11-, 1-1) where the dash-character indicates the don't-care condition. Similarly (11-, 101) meets the criterion as does (1-1, 110).

This metric has an inherent factorization non-determinism. Different factorizations are available for the same data and observed coveritem-number reduction is a function of the factorization. Nonetheless, the metric scores quite highly on average coveritem reduction and universal recognition. Even if different coveritems are inferred, each coveritem content is universally recognizable by the naming scheme.

Optimistic assignment implies that an input vector matching multiple coveritems increments all of them.

Coveritem names for this metric are in the same form as UCIS:FULL, with the addition of the '-' (dash) character to indicate don't care factorization.

```
metric-name :== UCIS:TRUEUDP_FLAT | UCIS:TRUEUDP_HIER
udp-coveritem-name :== 0 | 1 | - { 0 | 1 | - }
```

4.9.5.1.4 UCIS:UDP

The **UCIS:UDP** metric is defined *to require a coveritem for each of a full set of both positive and negative logic expression factors of the entity, with default optimistic assignment*.

This metric is very similar to the **UCIS:TRUEUDP** except that both the 0 and 1 expression-value factors are collected. This metric scores higher for coverage value, but lower for coveritem-reduction than **UCIS:TRUEUDP.**

```
metric-name :== UCIS:UDP_FLAT | UCIS:UDP_HIER
trueudp-coveritem-name :== 0 | 1 | - { 0 | 1 | - }
```

4.9.5.1.5 UCIS:STD

The **UCIS:STD** metric is defined *to require a coveritem for each sensitized vector of expression values.*

This metric records only combinations (vectors) of expression values that are sensitized. A sensitized vector of values is one for which a single value change (on any input) will change the expression result.

The coveritem naming form is binary string with don't care. Input order is sub-expression lexical order.

```
metric-name :== UCIS:STD_FLAT | UCIS:STD_HIER
std-coveritem-name :== 0 | 1 | - { 0 | 1 | - }
```

4.9.5.1.6 UCIS:CONTROL

The **UCIS:CONTROL** metric is defined *to require a coveritem for those combinations of expression values for which a single value change (on one or more input(s)) will change the expression result. For each combination of expression values, the input(s) whose change in value can change the expression result is/are referred to as controlling input(s). The scoring of a sub-expression table, if any, corresponding to an input is a function of that input being a controlling input, in the combination of expression values, of its parental expression.*

The coveritem naming form is binary string with don't care. Input order is sub-expression lexical order.

```
metric-name :== UCIS:CONTROL_FLAT | UCIS:CONTROL_HIER
control-coveritem-name :== 0 | 1 | - { 0 | 1 | - }
```

4.9.5.1.7 UCIS:VECTOR

The **UCIS:VECTOR** metric is defined *to require a coveritem for each sensitized value of a multi-bit vector used as a Boolean term in a higher-level expression.*

The operands of the SystemVerilog logical operators such as && or || act as single-bit logic values in the logical evaluation. If a multi-bit *reg* variable is an operand of a && or || operator, the sensitized values are the all-0s value and each value with a single 1. For example, the sensitized set of each three-bit operand to a && operator is 000, 001, 010 and 100.

Input order is inferred from the source language definition of expression bit order.

```
metric-name :== UCIS:VECTOR_FLAT | UCIS:VECTOR_HIER
vector-coveritem-name :== 0 | 1 { 0 | 1 }
```

4.9.5.1.8 UCIS:OBS

The **UCIS:OBS** metric is defined *to require a coveritem for each observable sensitized vector of values.*

This metric is identical to UCIS:STD for top-level expressions. For subexpressions, this metric is similar to UCIS:STD except that the bin values are further restricted by the requirement that the other sibling components of the top-level expression also be in the sensitized state.

The coveritem naming form is binary string with don't care. Input order is top-level lexical order of the subexpressions monitored for the given expression.

```
metric-name :== UCIS:OBS_FLAT | UCIS:OBS_HIER
obs-coveritem-name :== 0 | 1 | - { 0 | 1 | - }
```

4.9.5.1.9 UCIS:BITWISE_CONTROL

The **UCIS:BITWISE_CONTROL** metric is a bit-striped version of UCIS:CONTROL. *Each UCIS:CONTROL coveritem is replicated for each bit stripe.*

For example, the three UCIS:CONTROL ab coveritems for a single bit OR function (a | b) are 00, 10 and 01.

If a and b are vectors of length *m*, there will be *m* x 3 coveritems total, because there will be a set of three coveritems for each bit in the vector.

The coveritem naming form is the UCIS:CONTROL form with the normalized bit index appended. Bit index normalization is necessary because of expressions of the form (a[3:1] & b[-17:-19]). Normalization is 0-referenced to the right-hand-side index. Thus in the example the bins for (a[1] & b[-19]) are named 11#0#, 10#0# and 01#0# where #0# is the normalized bit index.

```
metric-name :== UCIS:BITWISE_CONTROL_FLAT | UCIS:BITWISE_CONTROL_HIER
bitwise_control-coveritem-name :== 0 | 1 | - { 0 | 1 | - } # normalized-bit-index #
```

4.9.5.2 Expression-value coverage metrics

Expression-value coverage metrics analyze multi-valued entities by the values they have taken.

4.9.5.2.1 UCIS:AUTO-n

The **UCIS:AUTO-n** metric is defined where *n* is a positive integer. It is a set of auto-binning metrics applied to multi-valued integral entities. It is defined to *require n coveritems, into which the full range of entity values are assigned equally in order starting at value 0, with excess numbers being assigned to the final coveritem*. Signed usage is ignored; value 0 is defined to be the all-bits-zero value.

Implementations may impose a limit on the number of automatic bins based on the value of the SystemVerilog auto_bin_max setting, or an equivalent mechanism, depending on source language.

```
metric-name :== UCIS:AUTO- number-of-coveritems
auto-coveritem-name :== auto[ coveritem-number ]
```

4.9.5.2.2 UCIS:ENUM

The **UCIS:ENUM** metric is defined for enumerated entities. This metric *requires a coveritem for each enumerated value that the entity source HDL type allows*. The coveritems are named for the enumerated names.

```
metric-name :== UCIS:ENUM
enum-coveritem-name :== source-HDL-enum-element-name
```

4.9.5.2.3 UCIS:STATE

The **UCIS:STATE** metric is defined for Finite-State Machine state variable entities. This metric *requires a coveritem for each state value that the state variable can take*. If the states are explicitly named in the HDL, for example when the state variable is an enumerated type, the coveritems are named for the state names. Otherwise the coveritems are named for the state values as integers or binary strings, depending on the type of the state variable.

```
metric-name :== UCIS:STATE
state-coveritem-name :== [source-HDL-state-name | source-enum-name | integer | binary-string ]
```

4.9.5.3 Transition coverage metrics

Transition coverage metrics analyze multi-valued entities by their value transitions.

4.9.5.3.1 UCIS:2STOGGLE

The **UCIS:2STOGGLE** metric is defined for a single-bit entity, including bits from bit-blasted vectors, and *records its transition from the value 0 to the value 1, and from the value 1 to the value 0*. The metric defines exactly two coveritems, one named '0->1' and the other named '1->0'

```
metric-name :== UCIS:2STOGGLE
2stoggle-coveritem-name :== 0->1 | 1->0
```

4.9.5.3.2 UCIS:ZTOGGLE

The **UCIS:ZTOGGLE** metric is defined for a single-bit entity and *records its transitions between the values 0, 1 and Z*. The metric defines exactly six coveritems, named '0->1', '1->0', '0->z', 'z->0', '1->z', 'z->1'

Data into and out of the X state is not collected for this metric.

```
metric-name :== UCIS:ZTOGGLE
ztoggle-coveritem-name :== 0->1 | 1->0 | 0->z | z->0 | 1->z | z->1
```

4.9.5.3.3 UCIS:XTOGGLE

The **UCIS:XTOGGLE** metric is defined for a single-bit entity and *records its transitions between the values 0, 1 and x*. The metric defines exactly six coveritems, named '0->1', '1->0', '0->x', 'x->0', '1->x', 'x->1'

Data into and out of the Z state is not collected for this metric.

```
metric-name :== UCIS:XTOGGLE
xtoggle-coveritem-name :== 0->1 | 1->0 | 0->x | x->0| 1->x | x->1
```

4.9.5.3.4 UCIS:TRANSITION

The **UCIS:TRANSITION** metric is defined as *recording the transitions that a variable can take from one value to another*. It defines the use of the -> separator between the values in the coveritem name. Value-identification syntax may be any recognized form appropriate to the variable type such as integer, state variable, enumerated name, binary string form, and so on. An application may apply a limit to the total number of bins created.

```
metric-name :== UCIS:TRANSITION
transition-coveritem-name :== start-value -> next-value { -> value-n }
```

4.10 Net Aliasing

HDL net types may be multiply represented in an instantiation hierarchy under different names. This is problematic for coverage aggregation because the net contributes to the calculated coverage multiple times.

UCIS_TOGGLE scopes represent objects with value, including nets. The UCIS_IS_TOP_NODE flag is set on the single canonical net in the database for each aliasing set.

The UCIS_IS_TOP_NODE flag and toggle net canonical name are UCISDB data items that transmit external knowledge of the HDL design connectivity. As such these items cannot be inferred or maintained by the UCIS API implementation. It is recommended, but not required, that the application that creates the data in a UCISDB adds this standardized semantic annotation where appropriate.

Any UCIS_TOGGLE scope that does not have this flag set is potentially an alias for one of the canonical nets. The UCIS_STR_TOGGLE_CANON_NAME property may be used to query the target canonical name (for the object on which the UCIS_IS_TOP_NODE flag is set) in Unique ID format, or NULL, depending on whether the toggle scope represents an aliased object. This interface provides sufficient data to eliminate alias versions of the data from coverage aggregation if required.

5 Data Model Schema

This chapter contains the following sections:

5.1 Introduction

This normative chapter describes rules which an API must follow to construct a compliant database.

The coverage hierarchy consists of **scope** and **coveritem** objects.

5.1.1 Scopes or hierarchical nodes

Designs and testbenches are hierarchically organized, i.e., organized like a tree. Design units (modules or VHDL entity/architectures) can be hierarchical, though they are not always.

Verification plans can be hierarchical. Even coverage data – of which the SystemVerilog covergroup is the best example – can be hierarchical.

To store hierarchical structures, which are basically elements of a database that can have children, the UCISDB has **scopes.**

5.1.2 Coveritems or leaf nodes

Coverage data and assertion data are basically *counters* – indicating how many times something happened in the design; for example, how many times a sequence completed, a coveritem incremented, or a statement executed. In the UCISDB, these counters and some associated data are called **coveritems**.

Coveritems are nodes that cannot have children and are *leaf nodes* of the database.

5.2 Primary key design for scopes and coveritems

A database design must clearly specify primary key usage. Primary key choices control the basic API construction and matching behavior for the database objects, irrespective of meaning assigned to the primary key values. For UCIS scopes and coveritems, the primary keys are used to construct the string-form Unique IDs. See “Getting Unique IDs from objects” on page 53.

Applications may use non-primary key data to further refine database selections and algorithms but the essential property that the primary key must be unique, is normative to this database design.

This section describes the database primary key data models for HDL scope objects, coverage scope objects, and coveritem objects. These are collectively referred to as ‘objects’ for the purposes of this section.

5.2.1 Character sets and encodings

Name components stored in the UCISDB shall be a byte sequence of non-zero length.

The byte value 0x00 shall be the name component terminator.

Stored name components may use any byte values from 0x20 to 0xFF.

Stored name components shall use Unicode code points and UTF-8 encoding.

Two stored name components shall be considered the same if and only if all bytes including the terminating 0x00 byte are present in each name component in the same order.

Type components stored in the UCISDB shall be a 64-bit one-hot integer.

5.2.2 Primary key components

The combination of a name and type shall be the **child primary key** for child objects under a parental object and shall therefore be a unique combination for every object under the same scope.

An attempt to create an object with a name and type that already exists under the parental object shall generate an API error.

The ordered list of child primary keys starting at the top of the database and traversing down the scope hierarchy to any intermediate or leaf object in the UCISDB shall uniquely identify that object.

The local object names shall be **name components** in the hierarchical identification scheme. A name component shall not necessarily be unique without a qualifying coverage type.

The local object types shall be **type components** in the hierarchical identification scheme. A type component shall not necessarily be unique without a name.

5.2.3 API interface to stored names and primary keys

The API routines shall be the intermediary access layer between a user application and the stored form of the objects.

The API shall provide access to native **local name** and **local type** components for an object.

In addition there shall be two forms of **API composite identification** derived from the complete ordered list of primary keys for any object. API routines shall recognize one of these two forms as noted for each relevant routine and perform appropriate translation to and from the stored form of the local components.

The two forms of API composite identification shall be called the **hierarchical fullname** and **unique ID** forms.

Both forms of API composite identification shall be a byte sequence of non-zero length. The byte value 0x00 shall be the terminator.

Each of these forms shall also have an equivalent relative form.

The **hierarchical fullname** form shall be constructed from name components only, and therefore shall identify zero, one, or a set of more than one objects.

The **unique ID** form shall identify either a singular object or zero objects. It shall be based on both the type and name components in the child primary key hierarchy.

These two forms shall not be interchangeable in the API routines. Each relevant API routine shall explicitly accept or construct either one form or the other as noted.

The **path separator** character managed by the routines ucis_GetPathSeparator() and ucis_SetPathSeparator() shall be used by the API when mapping child primary key components to API composite identification forms in both the hierarchical fullname and unique ID forms.

The hierarchical fullname of an object shall start with a single path separator character, to indicate that the first point of reference is the entire database. A single path separator character shall be inserted after each succeeding name component, except for the final name component, which is terminated with a 0x00 byte. The name components used in construction to name an object shall be the ordered list of parental name components traversing down the scope hierarchy, where the last name component is the local name of the target object.

The '\' character (backslash) shall be used as a single-character escape in the composite name forms. Any character immediately following the escape character shall be considered part of the name string itself, with no special meaning, with the exception of 0x00, which cannot be escaped. Escape characters shall not be copied into the stored name component, unless escaped themselves.

This requirement explicitly separates the native primary key representation from the constructed forms and supports constructed forms using any path separator character other than 0x00.

A **hierarchical relative name** shall be composed similarly but shall not start with the path separator. It may start with the name component of any object in the database. Relative hierarchical names are not necessarily unique within the database and are meaningful only when combined with a second piece of data to identify the parental scope.

The **unique ID** naming form shall similarly start with a single path separator character and shall similarly separate components with single path separator characters. Each component after a path separator shall be constructed from two pieces of information, the **type component representation** and the **name component.**

The **type** component representation shall be an unambiguous mapping of the 64-bit one-hot types to a string representation as described below.

The type component representation shall have two forms, one form to represent the set of scope types and one form to represent the set of coveritem types.

The **scope type** component (ucisScopeTypeT) shall be transformed to a positive decimal integer representation of the 0-referenced bit position of the scope type, followed by a colon ':' character, for example "23:".

The **coveritem type** component (ucisCoverTypeT) shall be transformed to a positive decimal integer representation of the 0-referenced bit position of the coveritem type, preceded and followed by a colon ‘:’ character, for example “:16:”.

5.2.3.1 Examples

The UCIS_TOGGLE scope type is defined to be 0x0000000000000001, one-hot bit position 0 and therefore the scope type component representation is 0:

The UCIS_FSM scope type is defined to be 0x0000000000400000, 22 and 22:.

The UCIS_INSTANCE scope type is defined to be 0x0000000000000010, 4 and 4:.

Thus two possible unique IDs within a UCISDB for coverage collected on variable my_state_int using ‘/’ as the path separator might be:

```
“/4:top/0:my_state_int” and “/4:top/22:my_state_int”
```

These scopes have the same hierarchical fullname:

```
“/top/my_state_int”.
```

The UCIS_TOGGLE_BIN coveritem type is defined to be 0x0000000000000200, one-hot bit position 9, and coveritem type component representation “:9:”. If the UCIS_TOGGLE scope in the example contains a UCIS_TOGGLEBIN coveritem named “1->0 there is also an object with a unique ID in the database:

```
/4:top/0:my_state_int/:9:1->0
```

Embedded path separator and escape characters in the name component shall be disambiguated in the constructed names as described above.

No character in the unique ID form shall be interpreted as a wildcard character.

The unique ID forms support hierarchy manipulations where a singular object identification is necessary.

The unique ID form shall have an equivalent relative form, the **relative unique ID,** with no preceding path separator, for example:

```
0:my_state_int/:9:1->0
```

In the example, this object is only recognizable relative to the scope /4:top

5.3 HDL language-specific rules

The API composite identification forms and database naming components shall be language-agnostic (however see “Matching and case sensitivity” on page 54 on language case-sensitivity issues). Language-specific escape syntax shall not be recognized by the API. The only escape syntax that shall be recognized by the API shall be in the API composite identification forms and shall be as described above. Thus the only allowable transform of name components when interpreting the API composite identification forms with respect to the stored forms, shall be the removal of backslash characters and unconditional acceptance of the succeeding character. Similarly, when constructing API composite identification forms from stored name components, the API shall insert backslash characters before the current path separator and stored backslash characters only.

5.3.0.1 Examples

The double quote characters in the examples delimit the strings and are not part of them. The terminating 0x00 byte is not shown but is assumed to terminate each string.

All the examples assume a current path separator character of /.

If the hierarchical fullname ‘C’ string is “/top/\\a_SystemVerilog_$name /a \/name“

There are three matching stored database components and they have name components “top”, “\a_SystemVerilog_$name “, and “a /name”.

If the hierarchical fullname ‘C’ string is “/top/\\a_VHDL_$name\\/a+reg”

There are three stored database components and they have name components “top”, “\a_VHDL_$name\” and “a+reg”.

No check is made as to the presence or validity of these escaping schemes within their own language.

5.4 Getting Unique IDs from objects

Unique IDs and hierarchical names may be retrieved from objects using the ucis_GetStringProperty() routine with the appropriate property enum, UCIS_STR_UNIQUE_ID.

A negative coverindex shall return the Unique ID for the scope.

Unique IDs may be constructed, they shall not necessarily be stored. Memory may be allocated by this routine to store the constructed Unique ID. The returned value shall be valid until the next call to ucis_GetStringProperty() for the database *db*, or until the database *db* is closed, whichever occurs first.

5.5 Using Unique IDs to perform database searches

5.5.1 Basic algorithm

By definition, a Unique ID form search shall return 0 or 1 database objects based on matching against the name plus type primary key hierarchy.

5.5.2 Matching and case sensitivity

The basic name matching algorithm shall use exact string-matching, which is inherently a case-sensitive match.

Naming case-sensitivity is assumed to be a function of source language. There is at least one HDL language (VHDL) where identifiers are not case sensitive. In VHDL, name matching is required to ignore case.

It is therefore also possible for design hierarchies to have a mix of case-sensitive and case-insensitive source languages, where for example the design is implemented in a mix of VHDL and SystemVerilog design units. In order to perform a traversing search on such a hierarchy, a case-sensitivity aware search routine needs to be aware of the local case-sensitivity policy.

To address these issues, two forms of unique ID search shall be provided. The basic search form shall perform exact type plus name matching throughout, and for example the name component TOP shall not match top for any language. This is the fastest search form, and sufficient when all input data from case-insensitive languages is known to have consistent case wherever it is found in the input data. For example VHDL names might have been consistently up-cased, or VHDL source case might have been consistently retained.

The case-sensitivity aware form of the search routine shall query each scope for its source language type as it iterates the hierarchy. Case-sensitivity shall be inferred from the language type. Specifically, if the language type is UCIS_VHDL, the match attempt of the name component of the candidate scope shall ignore case. Thus for a VHDL scope, TOP shall match top but for a SystemVerilog scope, it shall not. Coveritem matches shall inherit the naming sensitivity of their parent.

5.5.3 Multiple match possibilities differing only in case

It is possible for an application to create two child objects of the same type whose names differ only in case. This data is legitimate with respect to the UCISDB, but incorrect for VHDL and any language with case-insensitive identifier rules. API applications may generate warnings when data of this form is created.

The case-sensitivity aware match behavior for VHDL objects differing only in case shall be that the search routine may return any of the matching objects, and it shall be undefined which one it will choose. Further, the search routine shall be required only to traverse one matching path in its search and shall not be required to attempt other hierarchical matches on encountering a match failure. Any other matching objects shall in effect be unreachable via the case-sensitivity aware routines.

For example, consider a database containing these objects. Instance typing is shown to reinforce that these routines operate on Unique IDs, however the particular typing is not relevant to the example, except in that the typing is assumed to match. The language associated with the scope is indicated by its name.

- /4:SV_top/4:A_VHDL_OBJECT/4:AN_SV_OBJECT
- /4:SV_top/4:A_VHDL_OBJECT/4:an_sv_object
- /4:SV_top/4:a_vhdl_object/4:AN_SV_OBJECT
- /4: SV_top/4:a_vhdl_object/4:an_sv_object
- /4:SV_top/4:A_vHdL_oBjEcT

It is undefined whether a case-sensitivity aware search for /4:SV_top/4:a_vhdl_object/4:AN_SV_OBJECT returns object 1, object 3, or in fact, no object at all, if it traverses scope 5 first. The same basic search will unambiguously return object 3.

There is further ambiguity when, as in this case, an object in a case-sensitive language is referred to from a case-insensitive source. Whether one or both SystemVerilog variants exist, this usage is not covered by any standard and not easily analyzable. A mixed-language flow should be designed to avoid ambiguities such as these. For the purposes of the case-aware matching routines, case-sensitivity is tied to the object definition, not the calling location, and therefore the final objects in the example shall be treated as distinct.

5.5.4 Other case sensitivity considerations

The behavior of the case-sensitivity aware search routines is defined specifically only to affect the matching algorithm.

The export of name components from the database when the naming forms are constructed does not adjust the case of the names it encounters. The characters in the names are used to construct Unique IDs exactly as they are found. Thus the underlying behavior of the UCIS is case-retention for VHDL object names.

5.5.5 Routine definitions

Each search routine has a basic routine, and an equivalent routine that will perform matches based on the case-sensitivity rules of the intermediate scopes as described above

5.5.6 ucis_MatchScopeByUniqueID

5.5.6.1 ucis_CaseAwareMatchScopeByUniqueID

```
ucisScopeT ucis_MatchScopeByUniqueID( ucisT db, ucisScopeT scope, const char
    *uniqueID)
ucisScopeT ucis_CaseAwareMatchScopeByUniqueID( ucisT db, ucisScopeT scope, const char
    *uniqueID)
```

The uniqueID string may be the full or relative form. If the full form (where the first character is the path separator) is supplied, the supplied scope is ignored as the match is from the top of the database.

If a relative form Unique ID is supplied, matching starts at *scope*, or if this is NULL, at the top of the database.

Returns a scope, or NULL if there is no match.

No wildcards are supported.

5.5.7 ucis_MatchCoverByUniqueID

5.5.7.1 ucis_CaseAwareMatchCoverByUniqueID

```
ucisScopeT ucis_MatchCoverByUniqueID( ucisT db, ucisScopeT scope, const char
    *uniqueID, int *index)
ucisScopeT ucis_CaseAwareMatchCoverByUniqueID( ucisT db, ucisScopeT scope, const char
    *uniqueID, int *index)
```

The uniqueID string may be the full or relative form. If the full form (where the first character is the path separator) is supplied, the supplied scope is ignored as the match is from the top of the database.

If a relative form Unique ID is supplied, matching starts at *scope*, or if this is NULL, at the top of the database.

If the match succeeds the routine returns the parental scope of the cover item and sets the coverindex for the match in the index location. If the match fails the routine returns NULL.

No wildcards are supported.

6 Data Models

This chapter contains the following sections:

6.1 Introduction

The models and naming conventions in this chapter are advisory except where noted, and are not required to be enforced by the API. This advisory chapter describes rules for object construction and naming that, if followed, support Universal Object Recognition across multiple database sources.

The advisory models describe minimum data sets for Universal Object Recognition where such data sets have been identified. Nothing prevents a UCIS application from annotating the structures described with additional non-primary-key information, for example using the UCIS attribute system.

6.1.1 The Universal Object Recognition Compliance Flags

— UCIS_UOR_SAFE_SCOPE
— UCIS_UOR_SAFE_SCOPE_ALLCOVERS
— UCIS_UOR_SAFE_COVERITEM

The compliance flags are provided to identify objects that are compliant with universal object recognition as described in this standard. It is the responsibility of the data generation application to set the value of these flags appropriately.

UCIS_UOR_SAFE_SCOPE and UCIS_UOR_SAFE_SCOPE_ALLCOVERS are scope flags; UCIS_UOR_SAFE_COVERITEM is a coveritem flag.

UCIS_UOR_SAFE_SCOPE applies to the scope name itself and indicates that the local scope name and type combination is compliant with a universal recognition rule.

UCIS_UOR_SAFE_SCOPE_ALLCOVERS may be used to optimize the data model and indicates that all (non-recursive) child coveritems under a scope are compliant with a universal recognition rule. It is possible for a scope to contain a mix of compliant and non-compliant coveritems in which case this flag should not be set. Data generation tools including merge applications are responsible for maintaining these flags when this situation is created or encountered.

UCIS_UOR_SAFE_COVERITEM is set on individual coveritems to indicate that they are compliant with a universal recognition rule.

6.1.2 Coverage Model Overview

The coverage models in this chapter are subdivided into the following categories

— HDL scope data models
— Functional coverage data models
— Code coverage data models
— Miscellaneous coverage item data models.

These scope categorizations are illustrated in the following diagrams as italicized classes. These class names may be used in schema where multiple scope types share similar behavior.

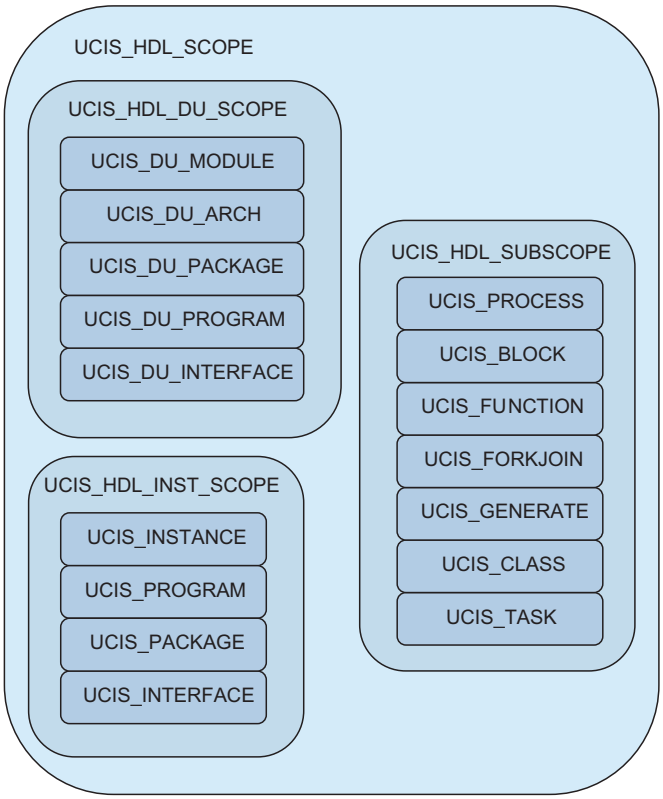

Figure 7—HDL Scopes

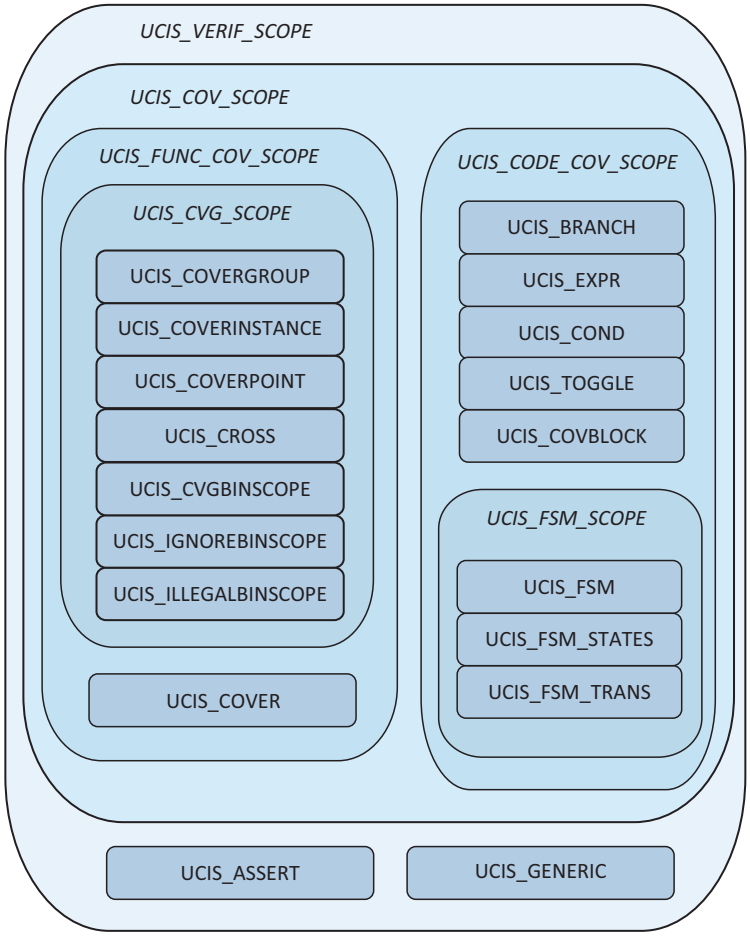

Figure 8—Functional and Code Coverage Scopes

If a scope or coveritem type is not mentioned or listed here, this standard has not defined a universal object recognition model for it. This can be because the structures and names are assumed to be inherited from some other standardized domain, such as a language reference manual. Alternatively, there may be types for which there is no generally-accepted information model.

Neither case prevents a coverage generation tool from creating data with this type, or a consumption tool from acting on the data to the degree to which it can understand it. It may however require extra analysis from tools attempting to reconcile data of this type across tools or vendors.

6.2 General Scope Data Model

The arguments to the scope constructors ucis_CreateScope() and ucis_CreateInstance() indicate the basic data associated with scope objects.

Table 6-4 — Scope data model

Constructor argument	Type	Description
db	ucisT	Identify the UCISDB where the scope is to be constructed.
parent	ucisScopeT	Locate the construction point in the UCISDB hierarchy (DU scopes are created at the top level indicated by NULL).
name	const char*	Primary key local name component. Naming models are covered for different types of scope below.
fileinfo/srcinfo	ucisSourceInfoT*	Composite information identifying source file, line, and token number.
weight	int	Coverage aggregation tuning value to indicate importance of this scope.
source	ucisSourceT	Source language (enumerated choices).
type	ucisScopeTypeT	Primary key local type component.
du_scope	ucisScopeT	For instance scope constructors only, makes the link to the associated design unit scope.
flags	int	General purpose flags.

6.3 HDL Scope Data Models

6.3.1 Introduction

HDL scopes are used to represent HDL and testbench designs, for known and possibly future languages that fall into the broad category of representing electronic entities. As such, very few construction restrictions are placed by the API on the database structures that can be constructed using HDL scopes. This maintains future flexibility, although may also limit the ability of the API as-defined to perform language-based error detection.

HDL scopes are categorized into Design Unit Scopes and other HDL scopes, such as instance scopes and subscopes. The instance scopes and subscopes participate in the construction of the instantiation hierarchy. The design unit scopes are stored as separate top level entities.

6.3.2 Structural Model

It is assumed that UCIS coverage generation applications share a common understanding of how to represent the design structures and that this understanding can be inherited from the languages themselves. There is further an explicit assumption that this common understanding is the basis for universal object recognition of language-based hierarchical structure. Furthermore, as this understanding is based on the language-specific reference manuals it need not and should not be replicated here. The generalization of the HDL structures that may be constructed is shown in Figure 9.

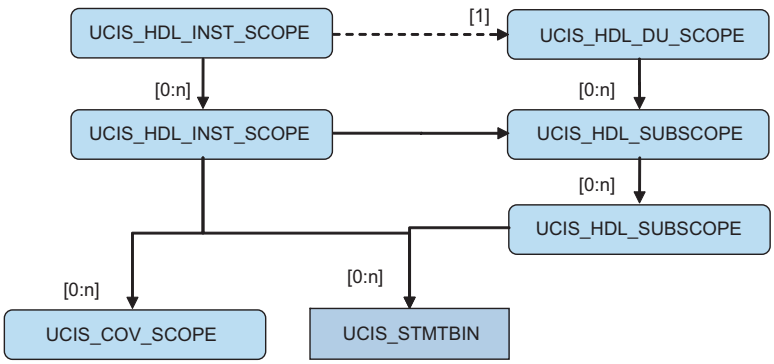

Figure 9—HDL Scope Schema

Figure 10 provides an example of a generic hierarchy data model. This example uses the flattened covergroup bin model. See also “Covergroups” on page 67 for a hierarchical covergroup bin model.

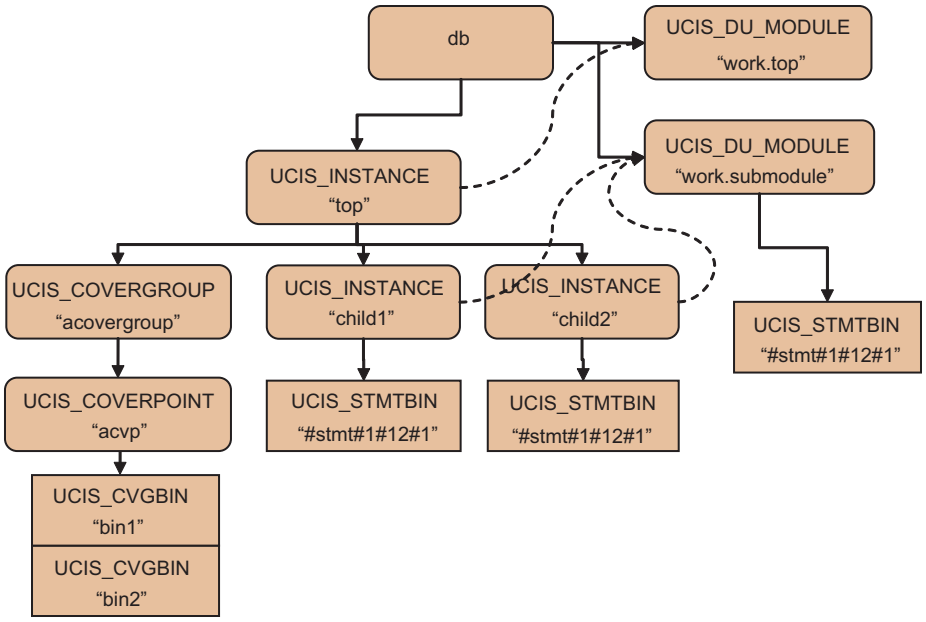

Figure 10—Generic Hierarchy Example

Unique ID list for objects in Figure 10 /24:work.top

```
/4:top
/24:work.submodule
/24:work.submodule/:5:#stmt#1#12#1#
/4:top/4:child1
/4:top/4:child2
/4:top/12:acovergroup
/4:top/12:acovergroup /14:acvp
/4:top/4:child1/:5:#stmt#1#12#1#
/4:top/4:child2/:5:#stmt#1#12#1#
/4:top/12:acovergroup /14:acvp/:0:bin1
/4:top/12:acovergroup /14:acvp/:0:bin2
```

6.3.3 Design Units in the Structural Model - Normative

Design unit scopes in the UCIS serve several purposes:

- a repository of information specific to the design unit
- a record of the link target for instance design unit information
- a repository for rolled-up coverage information (union-of-instances)

From each instance scope, its corresponding design unit may be accessed. The design unit shall exist prior to creating the instance. There are restrictions on the association of design unit types for particular instance types:

- **UCIS_INSTANCE** shall have a corresponding **UCIS_DU_MODULE** or **UCIS_DU_ARCH** scope as its design unit.
- **UCIS_PROGRAM** shall have a corresponding **UCIS_DU_PROGRAM** scope as its design unit.
- **UCIS_PACKAGE** shall have a corresponding **UCIS_DU_PACKAGE** scope as its design unit.

Note: Although VHDL and SystemVerilog do not have actual instances of packages in the language, verification tools can represent a package twice: the UCIS_PACKAGE corresponds to the top-level node in the instance tree, and UCIS_DU_PACKAGE to the definition of the package.

- **UCIS_INTERFACE** shall have a corresponding **UCIS_DU_INTERFACE** scope as its design unit.

All Design Unit callbacks (UCIS_REASON_DU) shall always precede all instantiation scope callbacks.

Each design unit shall have a signature attribute for the purpose of determining whether or not the source code has changed. This signature can be accessed through property enum UCIS_STR_DU_SIGNATURE. Note that individual API implementations may use any signature calculation method of their choice therefore this data is reliable only within the API that calculated it.

6.3.4 Design Unit Naming Model

Design unit scope names are constructed from three components, the library, primary, and secondary names. API utility routines are provided to construct and interpret the name from these components, ucis_ComposeDUName() and ucis_ParseDUName(). The general constructed form for the members of this class is:

```
du-name :== library . primary [ ( secondary ) ]
```

The secondary component is provided to represent VHDL architecture information and may be entirely omitted for SystemVerilog design units. Note that the UCIS design unit represents all parameterizations; separate design unit scopes are not constructed for differently parameterized instances found in the instantiation hierarchy.

6.3.4.1 UCIS_DU_MODULE

Scope type component representation 24:

6.3.4.2 UCIS_DU_ARCH

Scope type component representation 25:

6.3.4.3 UCIS_DU_PACKAGE

Scope type component representation 26:

6.3.4.4 UCIS_DU_PROGRAM

Scope type component representation 27:

6.3.4.5 UCIS_DU_INTERFACE

Scope type component representation 28:

6.3.5 HDL Instance and Subscope Naming Model

Many HDL components inherit user-supplied names from the HDL source. Some HDL component names are constructed from source language naming rules. Both these cases support Universal Object Recognition. There are also some HDL components that are or may be given tool-assigned names that do not support Universal Object Recognition at this time.

6.3.5.1 UCIS_INSTANCE

Scope type component representation 4:

```
instance-name :== hdl-instance-name
```

6.3.5.2 UCIS_PROCESS

Scope type component representation 5:

```
process-name :== [ user-label | simulator-assigned-process-name]
```

6.3.5.3 UCIS_BLOCK

Scope type component representation 6:

```
block-name :== [ user-label | simulator-assigned-block-name]
```

6.3.5.4 UCIS_FUNCTION

Scope type component representation 7:

```
function-name :== user-function-name
```

6.3.5.5 UCIS_FORKJOIN

Scope type component representation 8:

```
forkjoin-name :== [ user-label | simulator-assigned-forkjoin-name]
```

6.3.5.6 UCIS_GENERATE

Scope type component representation 9:

```
generate-name :== [HDL-name | tool-assigned-name]
```

(generate-name may be universally recognizable based on language rules but this is language-specific).

6.3.5.7 UCIS_CLASS

Scope type component representation 11:

```
class-name :== [user-hdl-class-type-name | tool-assigned-parameterized-class-name]
```

6.3.5.8 UCIS_PROGRAM

Scope type component representation 18:

```
program-name :== user-hdl-program-name
```

6.3.5.9 UCIS_PACKAGE

Scope type component representation 19:

```
package-name :== user-hdl-package-name
```

6.3.5.10 UCIS_TASK

Scope type component representation 20:

```
task-name :== user-hdl-task-name
```

6.3.5.11 UCIS_INTERFACE

Scope type component representation 21:

```
interface-name :== user-hdl-interface-name
```

6.4 Functional Coverage Data Models

6.4.1 Introduction

The UCIS_COVERGROUP, UCIS_COVERINSTANCE, UCIS_COVERPOINT and UCIS_CROSS scope types are a defined set of scopes specifically to implement functional coverage models similar to the SystemVerilog model. All these scopes are modeled together to show how they may interact.

SystemVerilog covergroups are defined in the IEEE 1800 standard. IEEE 1800 covergroup behavior is definitive in cases of ambiguity or change and supersedes this standard, in the case of contradiction. IEEE 1800-2009 is the reference version of the SystemVerilog standard on which the UCIS covergroup advisory models are based, however the implementation is general enough to be extensible to other languages such as ‘e’.

6.4.2 Covergroups

6.4.2.1 Structural Model

Covergroup and coverinstance data schema are similar, the covergroup schema is a superset of the coverinstance schema. The difference between these two types of scope is that the covergroup data is the cumulative type data, whereas the coverinstance data is the raw collected instance data. Thus covergroups can own a collection of coverinstances, as well as the consolidated view of the data for the type. A coverinstance can either be owned by its corresponding covergroup or parental instance that instantiated it but not both.

Covergroups may be class members, and in that case, their uniquely identifying UCISDB scope names will fall within a class-named UCISDB hierarchy. If the class is parameterized, the class naming hierarchy may have implementation-supplied names that are not meaningful across different UCISDBs.

There is a relation between a covergroup and its corresponding coverinstance and between a cross and its constituent coverpoints. Also, there is a relationship between a covergroup or a coverinstance and its parental HDL scope.

The first schema diagram below shows the relationship between classes, HDL scopes, covergroup, coverinstance, coverpoint, and cross scopes.

The dotted-line (non-ownership) relationship between the coverinstance and covergroup is an instantiation relationship; each coverinstance has a parental type.

There is also a dotted-line relationship between a cross and its coverpoints. This is a one-to-many relation and the utility routines ucis_GetIntProperty(UCIS_INT_NUM_CROSSED_CVPS), ucis_GetIthCrossedCvp() and ucis_GetStringProperty(UCIS_STR_NAME) are provided to query it.

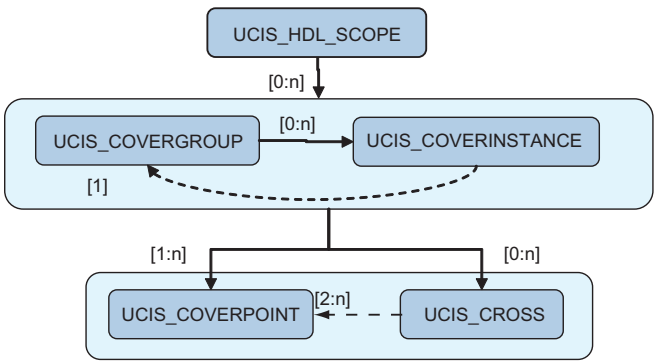

Figure 11—Covergroup relationships

The next two schema diagrams show how coveritems (in this case directly representing SystemVerilog bins) may be stored under coverpoints and crosses. UCISDB coveritem typing correlates directly to the SystemVerilog regular, default, ignore and illegal bin types. Intermediate scopes may be used to collate these bins by type, or bins of all types may collected under a single scope. The schema diagrams in Figure 12 show both possibilities. SystemVerilog language definitions are assumed to be the ultimate determinant of which bin types will actually be found under coverpoint and cross representations.

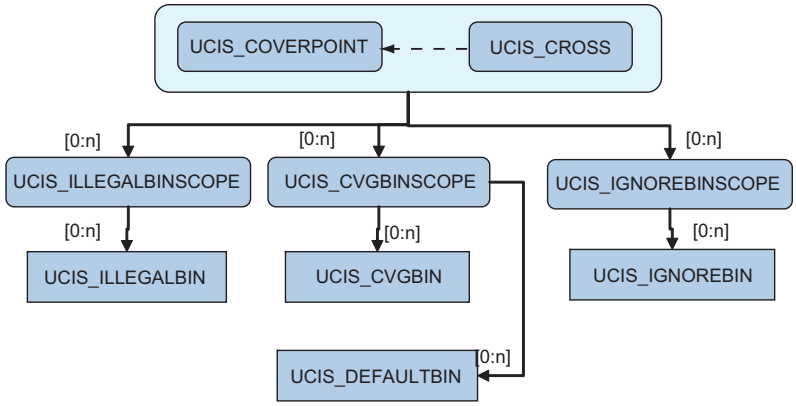

Figure 12—Type-collated Schema

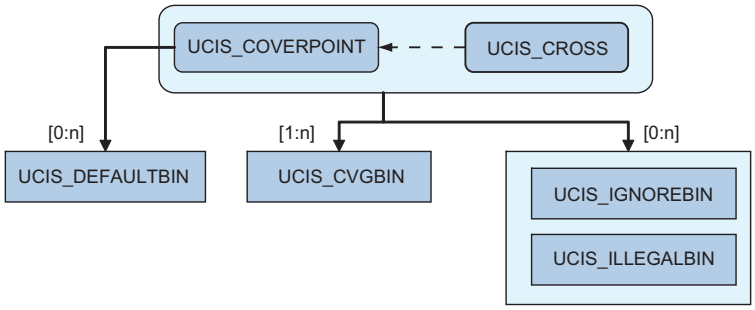

Figure 13—Flattened Schema

6.4.3 Naming model

This section describes the rules for local names of covergroups and related objects. Names that are supplied by an implementation as described here may not be recognizable across multiple UCISDBs.

The identification by name of UCISDB objects such as scopes and coveritems, relies on a combination of user-supplied and implementation-supplied names. In the case of covergroups, there are naming components for which the names may be user-supplied, but if there are no user-supplied names, the implementation is required to provide a default. When a user chooses not to supply a name, there are consequences to downstream UCISDB data consumers that are not always obvious, one of which may be that Universal Object Recognition is no longer possible.

In all cases for covergroup UCISDB hierarchies, users have the option to supply names. A user-supplied name is presupposed to meet the criterion for universal recognition. Any object with a parental hierarchy where all names are user-supplied, and whose own name is user-supplied, also meets this uniqueness criterion.

Names with language-defined name-generation schemes, even if not user-supplied, also meet this criterion, both locally, and as hierarchical elements. An example is the SystemVerilog auto bin names which are assigned to coveritems in the UCISDB.

Names that are specified to be implementation-supplied do not meet this standard. Thus an object with an implementation-supplied name anywhere in its parental hierarchy, or in its own name, shall be uniquely named within the UCISDB, but may not support object recognition across multiple UCISDBs.

UCIS_COVERGROUP scopes shall be named according to the covergroup type identifier from the HDL source covergroup definition. In the special case of embedded covergroups, which are anonymous types, the UCIS_COVERGROUP scopes shall be given the same name as the class covergroup variable.

UCIS_COVERINSTANCE scopes shall be named according to user-supplied names if present. The option.name and set_inst_name() methods to do this are described in IEEE 1800. If there is no user-specified name, the implementation shall supply a name.

UCIS_COVERPOINT and UCIS_CROSS scope names shall use the user-specified coverpoint or cross name if present. If there is no user-specified name, the implementation shall supply a name.

UCIS_CVGBINSCOPE, UCIS_IGNOREBINSCOPE and UCIS_ILLEGALBINSCOPE scope names shall use the user-specified bin identifier or IEEE 1800 defined ‘auto’ keyword if there is no user-specified bin-identifier.

UCIS_CVGBIN, UCIS_ILLEGALBIN, UCIS_DEFAULTBIN and UCIS_IGNOREBIN coveritem names used in SystemVerilog functional coverage context shall be compliant with IEEE 1800 bin naming. UCIS_ILLEGALBIN, UCIS_IGNOREBIN, and UCIS_DEFAULT bins may also be used in code coverage context, in which case they are named #illegal_bin#, #ignore_bin#, and #default_bin# respectively.

6.4.3.1 UCIS_COVERGROUP

Scope type component representation 12:

```
covergroup-name :== [1800covergroup_identifier]
```

Note: The covergroup_identifier is defined in IEEE 1800-2009 as the name of the coverage model, i.e. it names the covergroup type. Its usage changes slightly when the covergroup is an embedded covergroup; in this case it names the single instance variable of the embedded anonymous type. In both cases, the covergroup_identifier is used to name the UCIS_COVERGROUP scope in the UCIS.

Note: See IEEE 1800-2009 section 19.4 for specific use of the SystemVerilog covergroup_identifier in a covergroup embedded in a class.

6.4.3.2 UCIS_COVERINSTANCE

Scope type component representation 13:

```
coverinstance-name :== [user-supplied-name | pseudo-static-handle-path | simulator-assigned-
    name]
```

The pseudo-static-handle-path is an absolute path through the design hierarchy composed of one or more static scope names followed by zero or more pseudo-static class handle variable names, terminating in a pseudo-static covergroup instance handle variable name. A handle variable is pseudo-static in a simulation if there is a one-to-one correspondence between the handle variable and an object; in other words, the handle variable points to one and only one object during a simulation run (it may be null at times), and that object is only ever pointed to by that handle variable. The path separator shall be the natural path separator in the target language. Note that all handles in the path must be pseudo-static for the pseudo-static-handle-path to exist. If it does not exist, a simulator assigned name shall be used.

Note: See IEEE 1800-2009 section 19.7 and 19.8 for specific details on SystemVerilog coverinstance naming.

6.4.3.3 UCIS_COVERPOINT

Scope type component representation 14:

```
coverpoint-name :== [1800cover_point_identifier| tool-assigned-coverpoint-name]
```

Note: See IEEE 1800-2009 section 19.6 for specific details on SystemVerilog coverpoint names. The optional user-defined cover_point_identifier is used to name the UCIS_COVERPOINT scope if supplied.

6.4.3.4 UCIS_CROSS

Scope type component representation 15:

```
cross-name :== [1800cross_identifier | tool-defined-cross-name]
```

Note: See IEEE 1800-2009 section 19.6 for specific details on SystemVerilog cross bin names. The optional user-defined cross_identifier is used to name the UCIS_CROSS scope if supplied.

6.4.3.5 UCIS_CVGBINSCOPE

Scope type component representation 30:

```
cvgbinscope-name :== [1800bin_identifier | auto]
```

Note: See IEEE 1800-2009 section 19.5 and 19.6 for specific details on SystemVerilog bin names under UCIS_COVERPOINT scopes and UCIS_CROSS scopes respectively.

6.4.3.6 UCIS_IGNOREBINSCOPE

Scope type component representation 31:

```
ignorebinscope-name :== [1800bin_identifier]
```

Note: See IEEE 1800-2009 section 19.5 and 19.6 for specific details on SystemVerilog bin names under UCIS_COVERPOINT scopes and UCIS_CROSS scopes respectively.

6.4.3.7 UCIS_ILLEGALBINSCOPE

Scope type component representation 32:

```
illegalbinscope-name :== [1800bin_identifier]
```

Note: See IEEE 1800-2009 section 19.5 and 19.6 for specific details on SystemVerilog bin names under UCIS_COVERPOINT scopes and UCIS_CROSS scopes respectively.

6.4.3.8 UCIS_CVGBIN

Coveritem type component representation :0:

```
cvgbin-name :== [language-defined-name | #bin#]
```

Note: UCIS_CVGBIN names are derived from language rules where applicable. Language-defined names are considered to be universally recognizable.

Note: See IEEE 1800-2009 section 19.5 and 19.6 for specific details on SystemVerilog bin names under UCIS_COVERPOINT scopes and UCIS_CROSS scopes respectively.

6.4.3.9 UCIS_IGNOREBIN

Coveritem type component representation :19:

```
ignorebin-name :== [language-defined-name | #ignore_bin#]
```

6.4.3.10 UCIS_ILLEGALBIN

Coveritem type component representation :20:

```
illegalbin-name :== [language-defined-name | #illegal_bin#]
```

6.4.3.11 UCIS_DEFAULTBIN

Coveritem type component representation :21:

```
defaultbin-name :== [ language-defined-name | #default_bin#]
```

6.4.3.12 Other Information Models for Covergroups

There are other user choices within the IEEE 1800 definitions that have downstream tool side-effects. For example, if a user specifies both type_option.merge_instances=0 and option.per_instance=0, no coveritem data for covergroups is required to be saved in the UCISDB. The per_instance option, when FALSE, explicitly allows the coverinstance scopes to be optimized away, including their coveritems. The merge_instances option, when FALSE, requires only that the covergroup type average coverage (a real, not integral, quantity) be available.

These options therefore directly affect the UCISDB contents. The covergroup type behavior for the merge_instances option when it is FALSE, is that the average coverage of the instances is recorded, not the coveritems within the covergroup. This computed average coverage is a real quantity, and cannot be represented in coveritem counts. There is no meaningful coveritem data to be captured and therefore there are no coveritems under the UCIS_COVERPOINT scopes. The average real-valued coverage values shall be retrievable with the ucis_GetRealProperty() routine for the UCIS_REAL_CVG_INST_AVERAGE property, expressed as a percentage between 0 and 100.

When merge_instances is TRUE, the captured bin data is the merge of all the instance bins under the covergroup type scope. The average coverage values shall be retrievable with the UCIS_REAL_CVG_INST_AVERAGE in this case too. The merged coveritems for the type shall also be available.

6.4.3.13 Examples

Coverpoint and cross example

```
module top;
 int a=0, b=0;
 covergroup cg;
  type_option.comment = Example;
  type_option.merge_instances = 1;
  option.at_least = 2; // becomes the 'goal' in the coveritems
  cvpa: coverpoint a {bins a = {0}; }
  cvpb: coverpoint b {bins b[] = {1,2}; ignore_bins c = {3}; }
  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
```

This simple example causes a lot of data to be stored into an associated UCISDB file. The diagrams below show some of it. The merge_instances=1 type option results in the storage of covergroup bins and the per_instance=0 (defaulted) option results in the optional lack of stored cover instances. The cross bin (coveritem) has been assigned a language-defined name.

The first diagram shows the type-collation model for the collected data. The second diagram is an alternate flattened representation of the same data.

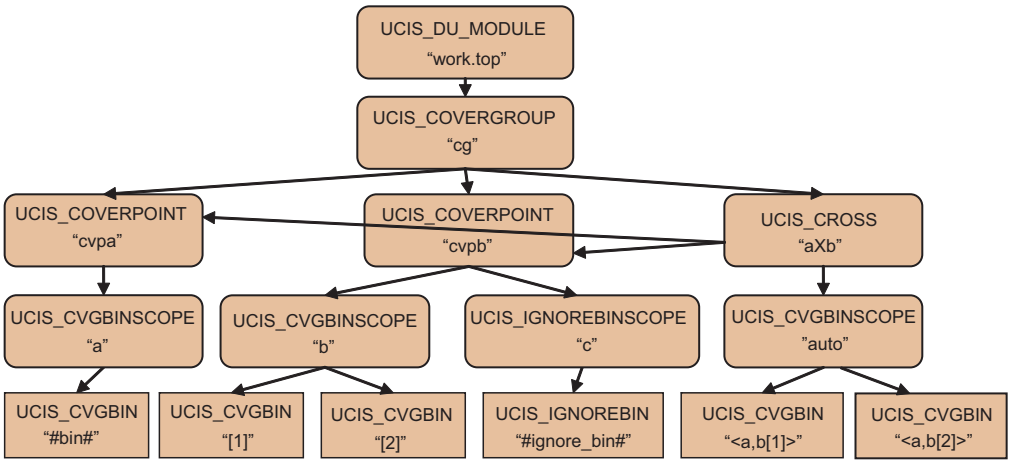

Figure 14— Hierarchical Representation

Unique ID list for hierarchical representation (Figure 14):

```
/24:work.top
/24:work.top/12:cg
/24:work.top/12:cg/14:cvpa
/24:work.top/12:cg/14:cvpb
```

```
/24:work.top/12:cg/15:axb
/24:work.top/12:cg/14:cvpa/30:a
/24:work.top/12:cg/14:cvpb/30:b
/24:work.top/12:cg/14:cvpb/31:c
/24:work.top/12:cg/15:axb/30:auto
/24:work.top/12:cg/14:cvpa/30:a/:0:#bin#
/24:work.top/12:cg/14:cvpb/30:b/:0:[1]
/24:work.top/12:cg/14:cvpb/30:b/:0:[2]
/24:work.top/12:cg/14:cvpb/31:c/:19:#ignore_bin#
/24:work.top/12:cg/15:axb/30:auto/:0:<a,b[1]>
/24:work.top/12:cg/15:axb/30:auto/:0:<a,b[2]>
```

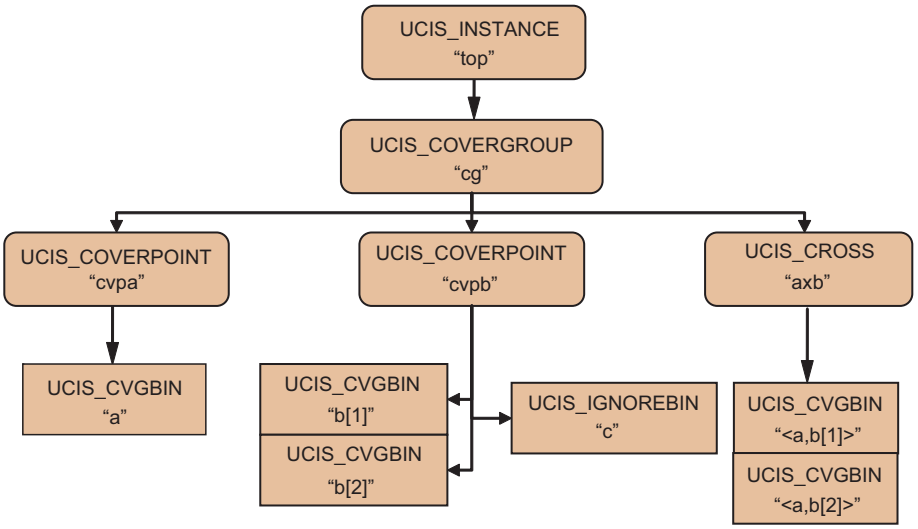

Figure 15—Flattened Representation

Unique ID list for flattened representation: See Figure 15

```
/4:top
/4:top/12:cg
/4:top/12:cg/14:cvpa
/4:top/12:cg/14:cvpa/:0:a
/4:top/12:cg/14:cvpb
/4:top/12:cg/14:cvpb/:19:c
/4:top/12:cg/14:cvpb/:0:b[1]
/4:top/12:cg/14:cvpb/:0:b[2]
/4:top/12:cg/15:axb
/4:top/12:cg/15:axb/:0:<a,b[1]>
/4:top/12:cg/15:axb/:0:<a,b[2]>
```

6.5 Code Coverage Data Models

6.5.1 Introduction

Many of the code coverage models rely on lexical information to construct names for inherently nameless items such as branches or expressions.

Lexical information is derived from the source HDL file or files. The underlying information model is that these items can be identified by a combination of *file-number* plus *line-number* plus *item-number* information.

Definitions for file-numbers and line-numbers can be found in "Source-derived name components" on page 37. Item-number definitions are specialized to the type of coverage being collected and are described in the appropriate sections that follow.

6.5.2 Branch Coverage

6.5.2.1 Structural Model

Branch coverage models whether branches were taken. See also condition coverage (UCIS_COND) for more in-depth analysis of branch causation.

Branching structure representation is designed to be language-agnostic. Universal object recognition is based on syntactic (token-based) rather than semantic (language-rules based) information.

Two primary forms of HDL branching syntax are recognized, the **if-elsif-else** form and the **switch-case** form. Actual keyword syntax is language-dependent. The SystemVerilog ternary form is considered equivalent to an if-else form using alternate tokens.

The logical analysis of both forms is similar and is as follows. A branching structure is introduced by the initiatory token, for example the first **if** or the SystemVerilog **case** token. One or more explicitly true coveritems are defined, i.e. coveritems associated with an explicitly-tested expression or expressions. In the if-style, the **if** token itself defines the first true coveritem. In the case style, the first case item token defines the first true coveritem.

Further true coveritems are optional, introduced either by an **elsif** (or equivalent) token, or by further instances of the specific case item tokening used by the language.

Finally there is an optional differentiated catch-all coveritem. It is differentiated by the absence of an explicit expression. This is the terminating **else** or **default** token. Because this token is (or may be) optional, in its absence, an **all-false** coveritem is inferred, to count the cases where the initiating branching was executed but none of the explicit tests were evaluated true. The all-false coveritem is only inferred for branching structures that are not syntactically full-case. It is not always possible to detect that a branching structure is full-case in the absence of the terminating else or default token. Some tools or languages may use pragma syntaxes to force full-case behavior. Otherwise, it is the presence or absence of the terminating token that determines the inference of full-case behavior or not.

Token identification underlies both scope and coveritem identification mechanisms. As described above, there are three HDL token types in this scheme – the *initiatory* token type, the *true* token type, and the *default* token type.

The *initiatory* token type is the branch-initiation token, for example the original **if** or SystemVerilog **case** token, or the ternary **?** token.

The *true* token type identifies branch counts associated with an explicit expression test. In the if-style, the first **if** is also a true token, as are the subsequent **elsif**s. In the case-style, the first and subsequent true-branch tokens are the case item tokens.

The *default* token-type is the catch-all token, which is the optional terminating **else** or **default** token. This is differentiated from the true-branch tokens by the fact that it may be absent, and the fact that it has no associated explicit expression test.

The basic branch structure is a UCIS_BRANCH_SCOPE with 2 or more UCIS_BRANCHBIN coveritems.

A branch coverage item-number is derived from the generalization of the recognition of the tokens described above.

It is defined as *the 1-referenced count of whatever language-defined tokens are used to implement the branching functions on the source HDL line*. Encountering any of the functional branching tokens (initiatory, true, or default) causes the item-number to increment. Separate counts are not maintained per function.

Only branching-related tokens count; other language tokens do not increment the branching item-number. For example block begin or end tokens, or the VHDL **then**, do not affect the count. The dual-keyword **else if** (for example SystemVerilog) is counted as a single *true*-type token, equivalent to the VHDL **elsif** keyword. Similarly, **default:** counts as a single *default*-type token, the colon is not counted separately.

With these definitions in place, branch structures are created as follows.

An *initiatory* token-type causes the construction of a UCIS_BRANCH branching scope, with naming derived from its file-number, line-number, and item-number.

This scope must contain two or more UCIS_BRANCHBIN coveritems. There is at least one true coveritem (a true coveritem is associated with an explicitly-tested condition) and a catch-all coveritem that is either the explicit **else** or **default** coveritem, or is an inferred all-false coveritem.

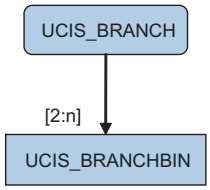

Figure 16—Basic branch structure

See the section, "Statement and Block Coverage" on page 81 for a description of hierarchical branch coverage.

6.5.2.2 Naming model

All the branching scopes and coveritems are named by their file-number plus line-number plus item-number information with the exception of the all-false coveritem (which by its nature has no token). Note that the first coveritem of the **if** structure will have the same line-number and item-number as the branch scope itself, as the first **if** is also the first *true* branch token.

Note that although some forms of branching imply prioritization, the UCISDB coveritem representation, including database ordering, explicitly does not represent any language prioritization rules or HDL order. The only way in which coveritems are identified, is as defined. The prioritization semantic is not represented in the UCISDB. Coveritem order does not carry meaning in the UCISDB.

6.5.2.2.1 UCIS_BRANCH

Scope type component representation 1:

```
branch-scope-name :== #branch#file-number#line-number#item-number#
```

6.5.2.2.2 UCIS_BRANCHBIN

Coveritem type component representation :6:

```
branchbin-name :== [ #true#file-count#line-number#item-number#
| #default# file-count#line-number#item-number#
| all_false_bin ]
```

6.5.2.3 Nested branching

Branch nesting does not map to code coverage UCISDB scope nesting. See the section, "Statement and Block Coverage" on page 81 for a description of nested branch coverage. For example, an **if** branch initiated on a case item line, causes construction of a code coverage scope that is a sibling to the case branching scope, under the same parental HDL scope. This is shown below.

```
case (a) begin // line 10
0: if (b==0) $do_this();
default: $do_that();
endcase
```

The case statement on line 10 is an *initiatory* token and causes the inference of a branch coverage scope. Two branching tokens are present on line 11, the first is a *true* token type, indicating a case item associated with the initiatory token on line 10. The second is an *initiatory* **if** token which causes the construction of a new code coverage branch scope for the design unit. This new scope will be a sibling of the case branch from line 10.

6.5.2.3.1 Examples

Note the similarity of the inferred models across two languages and both branching forms based on the abstract tokenization analysis.

SystemVerilog ‘if’ with no default

```
module top;                 // line 1
 bit x = 0;                 // line 2
 bit y = 0;                 // line 3
 always @ (x or y) begin    // line 4
  if (x)                    // line 5
   $display("x is true");   // line 6
  else if (y)               // line 7
   $display("y is true");   // line 8
  end                       // line 9
 initial begin              // line 10
  #1; x = 1;                // line 11
  #1; x = 0;                // line 12
  #1; y = 1;                // line 13
 end                        // line 14
endmodule                   // line 15
```

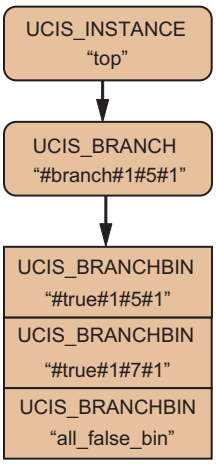

Figure 17—SystemVerilog ‘if’ with no default

Unique ID list for the diagram above:

```
/4:top
/4:top/1:#branch#1#5#1#
/4:top/1:#branch#1#5#1#/:6:#true#1#5#1#
/4:top/1:#branch#1#5#1#/:6:#true#1#7#1#
/4:top/1:#branch#1#5#1#/:6:all_false_bin
```

VHDL ‘if’ with default

```
library IEEE;                                  -- line 1
use IEEE.STD_LOGIC_1164.all;                   -- line 2
use std.textio.all;                            -- line 3
entity top is                                  -- line 4
end top;                                       -- line 5
architecture arch of top is                    -- line 6
 signal x : std_logic := '0';                  -- line 7
 signal y : std_logic := '0';                  -- line 8
 begin                                         -- line 9
  branch: process                              -- line 10
   variable myoutput : line;                   -- line 11
   begin                                       -- line 12
    wait until x'event or y'event;             -- line 13
    if (x = '1') then                          -- line 14
     write(myoutput,string'("x is true"));     -- line 15
     writeline(output, myoutput);              -- line 16
    elsif (y = '1') then                       -- line 17
     write(myoutput,string'("y is true"));     -- line 18
     writeline(output, myoutput);              -- line 19
    else                                       -- line 20
     write(myoutput,string'("x,y false"));     -- line 21
     writeline(output, myoutput);              -- line 22
    end if;                                    -- line 23
   end process branch;                         -- line 24
  drive: process                               -- line 25
   begin                                       -- line 26
    wait for 10 ns;                            -- line 27
    x <= '1';                                  -- line 28
    wait for 10 ns;                            -- line 29
    x <= '0';                                  -- line 30
    wait for 10 ns;                            -- line 31
    y <= '1';                                  -- line 32
    wait;                                      -- line 33
   end process drive;                          -- line 34
end arch;                                      -- line 35
```

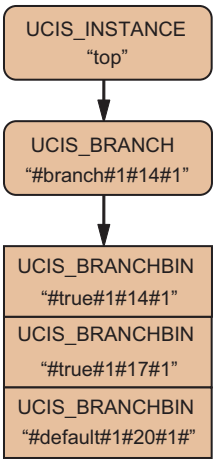

Figure 18—VHDL ‘if’ with default

SystemVerilog ‘case’ with default:

```
module top;                                 // line 1
 int x = 0;                                 // line 2
 always @ (x)                               // line 3
  case (x)                                  // line 4
   1: $display("x is 1");                   // line 5
   2: $display("x is 2");                   // line 6
   default: $display("x is not 1 or 2");    // line 7
  endcase                                   // line 8
 initial begin                              // line 9
  #1; x = 1;                                // line 10
  #1; x = 2;                                // line 11
  #1; x = 3;                                // line 12
 end                                        // line 13
endmodule                                   // line 14
```

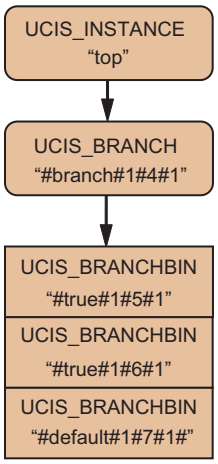

Figure 19—SystemVerilog ‘case’ with default

SystemVerilog nested branches. Note how the ‘case’ and both ‘if’ tokens cause inferred sibling branch scopes.

```
module top;                                    // line 1
 int x = 0;                                    // line 2
 int y = 0;                                    // line 3
 always @ (x)                                  // line 4
  case (x)                                     // line 5
   0: if (y==0) $display("both 0");            // line 6
   1: if (y==1) $display("both 1");            // line 7
      else $display("x is 1 but y is not");    // line 8
  endcase                                      // line 9
 initial begin                                 // line 10
  #1; y = 1;                                   // line 11
  #1; x = 1;                                   // line 12
  #1; y = 0;                                   // line 13
 end                                           // line 14
endmodule                                     // line 15
```

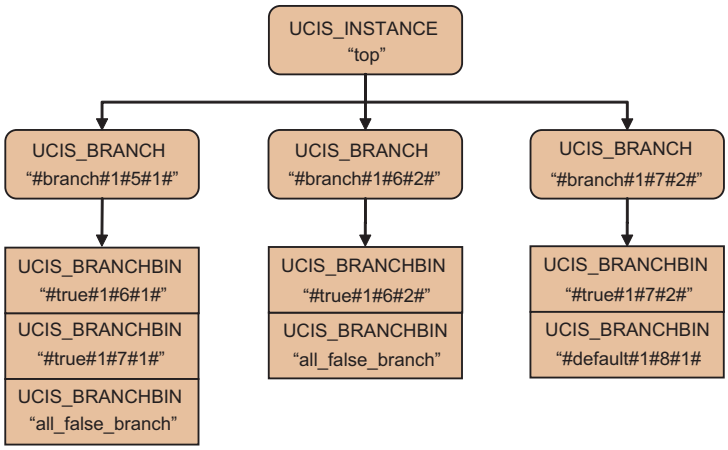

Figure 20—SystemVerilog nested branches

6.5.3 Statement and Block Coverage

6.5.3.1 Structural Model

The data model for Statement coverage is based on UCIS_STMTBIN coveritems that can be constructed under any HDL scope. The data model for block coverage is based on basic blocks (UCIS_BBLOCK scopes) that can be identified under any HDL scope. A UCIS_BLOCKBIN coveritem underneath a UCIS_BBLOCK scope shall contain the coverage for its parental UCIS_BBLOCK.

In case of a combination of block and statement coverage, each statement can be represented through its corresponding UCIS_STMTBIN coveritem underneath its constituent block scope. I.e., UCIS-based applications can choose to model only statement coverage, only block coverage, or both block and statement coverage. For branch coverage, the pre-defined scope flag UCIS_SCOPE_BLOCK_ISBRANCH on the block scope shall indicate that the block scope can also denote a branch scope.

The naming model described below allows unique identification of each individual statement or block or branch (true/false/case/default) even if there are differences in representation of blocks and/or statements across UCISDB implementations. Also, irrespective of the representation, coverage statistics should be computed assuming a flat representation so as to have consistent coverage statistics across representations.

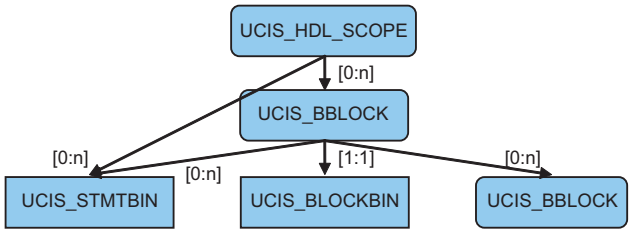

Figure 21—Statement and block coverage naming

6.5.3.2 Naming Model

The coveritems are named according to file-number, line-number, and item-number. The statement item-number is derived from the number of statements that start on the line and is 1-referenced. Statement start tokens anchor statement counting; identification of statement start tokens is derived from the language-dependent definition of a statement. Statements may be compound and contain other statements; for each token that starts any statement on a line, including statements embedded in other statements, the statement item-number is incremented. Statements that do not begin in a line are not considered for that line.

A block' file-number, line-number and item-number are determined by the file-number, line-number and item-number respectively of the first constituent statement of that block.

6.5.3.2.1 UCIS_STMTBIN

Coveritem type component representation :5:

```
stmtbin-name :== #stmt# file-number # line-number # item-number #
```

6.5.3.2.2 UCIS_BBLOCK

Scope type component representation 29:

```
block-scope-name :== #[block | true | false | case | default]#file-number#line-number#item-
    number#
```

6.5.3.2.3 UCIS_BLOCKBIN

Coveritem type component representation :22:

```
block-bin-name :== #block#
```

6.5.3.3 Examples

SystemVerilog abstract tokenizing:

```
always @ (a) begin // line 1
b=c; d=e; f=       // line 2
g;                 // line 3
end                // line 4
```

Statement bins may be constructed for the statements b=c d=e and f=g. The file/line/item triads to identify the statements would be, respectively (for a file number of 1), 1/2/1, 1/2/2, and 1/2/3. The statement f=g is named for the line on which the statement starts. This definition is designed to support unambiguous universal object recognition. Applications may choose to store additional information to identify statement span lines.

6.5.3.3.1 SystemVerilog with generate blocks example

Note that the UCIS_GENERATE blocks have a user-assigned name and a language-standard name respectively.

```
module top;                            // line 1
 bottom #0 inst0();                    // line 2
 bottom #1 inst1();                    // line 3
endmodule                              // line 4
module bottom;                         // line 5
 parameter clause = 0;                 // line 6
 if (clause == 0)                      // line 7
  begin: clause0                       // line 8
   initial $display("hello from %m"); // line 9
  end                                  // line 10
 else                                  // line 11
  begin                                // line 12
   initial $display("hello from %m"); // line 13
  end                                  // line 14
endmodule                              // line 15
```

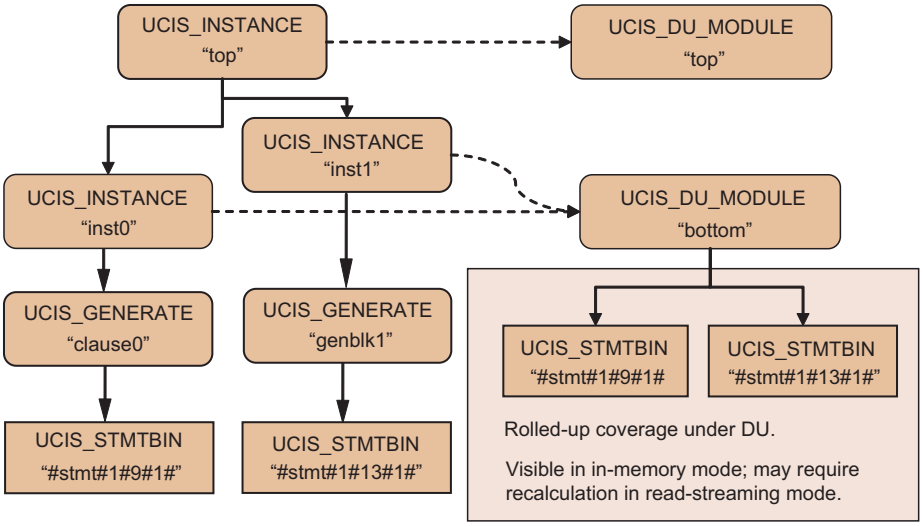

Figure 22—SystemVerilog with generate blocks

Unique ID list for the diagram above:

```
/4:top
/24:work.top
/4:top/4:inst0
/24:work.bottom
/4:top/4:inst1
/4:top/4:inst0/9:clause0
/4:top/4:inst1/9:genblk1
/24:work.bottom/:5:#stmt#1#13#1#
/4:top/4:inst0/9:clause0/:5:#stmt#1#9#1#
/4:top/4:inst1/9:genblk1/:5:#stmt#1#13#1#
/24:work.bottom/:5:#stmt#1#9#1#
```

SystemVerilog nested branches

```
module top;                                  // line 1
 int x = 0;                                  // line 2
 int y = 0;                                  // line 3
 always @ (x)                                // line 4
  case (x)                                   // line 5
   0: if (y==0) $display("both 0");          // line 6
   1: if (y==1) $display("both 1");          // line 7
      else $display("x is 1 but y is not"); // line 8
  endcase                                    // line 9
 initial begin                               // line 10
  #1; y = 1;                                 // line 11
  #1; x = 1;                                 // line 12
  #1; y = 0;                                 // line 13
 end                                         // line 14
endmodule                                    // line 15
```

The following diagram illustrates block coverage representation. Although, it does not show rolled up coverage for the design unit, the representation shall be similar for the design unit as well. UCIS_BBLOCK* indicates that the UCIS_BBLOCK scope also denotes a branch scope based on the pre-defined scope flag UCIS_SCOPE_BLOCK_ISBRANCH.

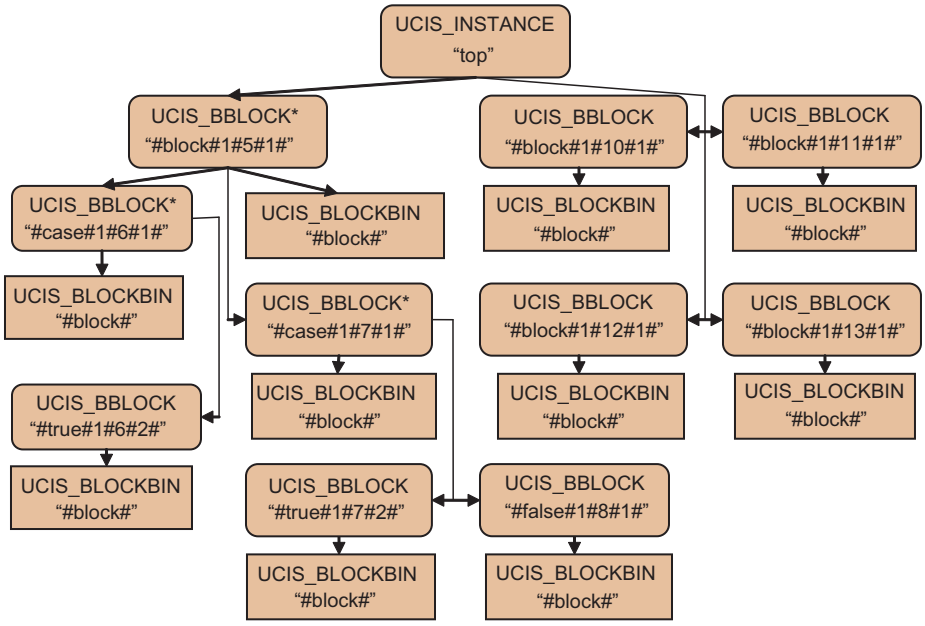

Figure 23—Example of block coverage

Another block coverage example is shown below. In this example, the basic block (UCIS_BBLOCK) scopes are flattened, and contain UCIS_STMTBIN coveritems for each statement in each block, rather than a single UCIS_BLOCKBIN.

```
module top;                                    // line 1
 int x = 0;                                    // line 2
 int y = 0;                                    // line 3
 always @ (x)                                  // line 4
  case (x)                                     // line 5
   0: if (y==0) $display("both 0");            // line 6
   1: if (y==1) $display("both 1");            // line 7
      else $display("x is 1 but y is not");    // line 8
  endcase                                      // line 9
 initial begin                                 // line 10
  #1; y = 1;                                   // line 11
  #1; x = 1;                                   // line 12
  #1; y = 0;                                   // line 13
 end                                           // line 14
endmodule                                      // line 15
```

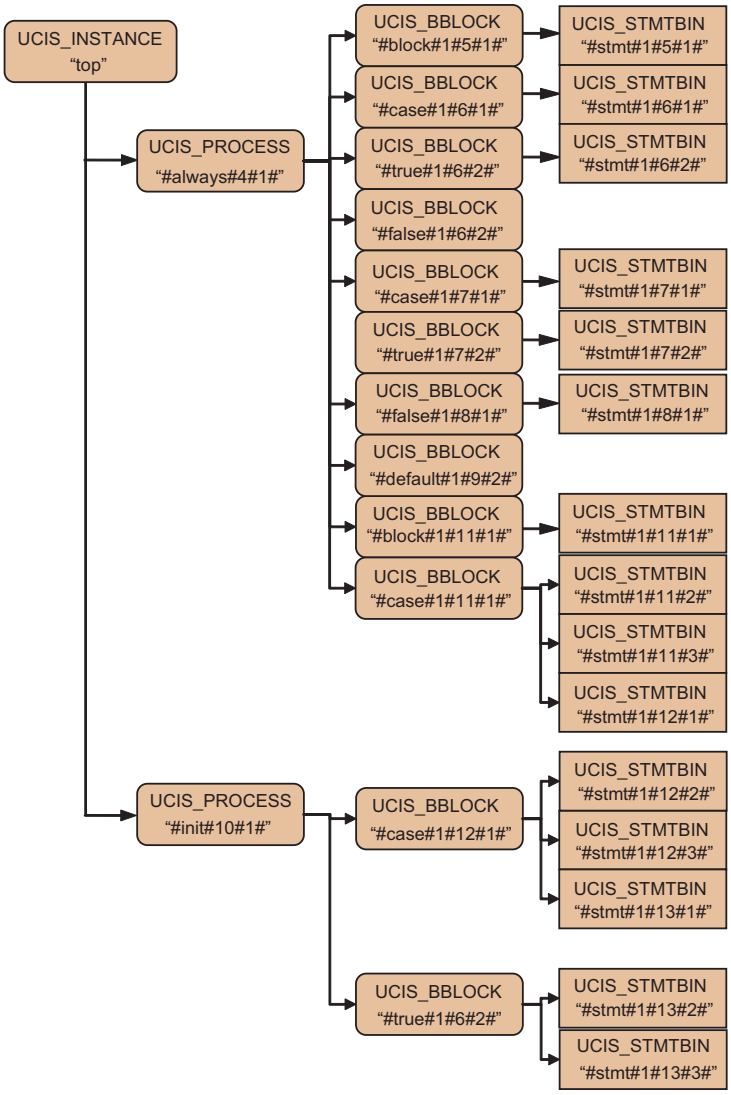

Figure 24—Example of block coverage with UCIS_BBLOCK flattened

6.5.4 Condition and Expression Coverage

6.5.4.1 Structural Model

Condition and expression coverage models coverage on Boolean expression objects with value when they are used in conditional and assignment contexts respectively.

The flat structural model is a two-scope hierarchy where the top UCIS_EXPR or UCIS_COND scope identifies the HDL condition or expression and contains one or more input-contribution metric scopes. All the Boolean inputs are analyzed together in each metric scope. See "Metric Definitions" on page 41 for standardized metric scope definitions. The COND and EXPR hierarchies are identical except for the names. The intent is that the COND hierarchy be used in contexts where the expression result is used in a condition control statement, and that the EXPR hierarchy be used for other input-value contexts.

A hierarchical structural model is based on the expression hierarchy as defined by respective language precedence and associativity. It has a multi-scope hierarchy where the top UCIS_EXPR or UCIS_COND scope identifies the top-level HDL condition or expression that is being represented for coverage. The second (or subsequent) UCIS_EXPR or UCIS_COND scope identifies operands of the expression modeled by the respective parental scope. Additionally, each of these UCIS_EXPR or UCIS_COND scopes may contain zero or more UCIS_EXPR or UCIS_COND input-contribution metric scopes. Also, irrespective of the representation, coverage statistics should be computed assuming a flat representation so as to have consistent coverage statistics across representations.

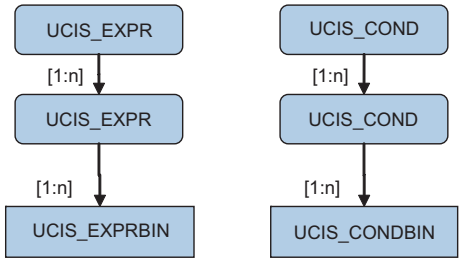

Figure 25—Example of condition and expression structural model

6.5.4.2 Naming Model

The top-level condition and expression coverage scopes are both identified by names constructed from file-number, line-number, and item-number. The item-number of a top UCIS_EXPR or UCIS_COND scope is derived from the number of top expression scopes (operators) on the line and is 1-referenced. The intermediate and/or leaf UCIS_EXPR or UCIS_COND scopes are identified by an item-number which is derived from level-order traversal of the expression hierarchy, starting from the corresponding parental top UCIS_EXPR/UCIS_COND scope, and is 1-referenced.

The condition and expression coverage scopes that represent input-contribution metrics are named for the metric that defines the coveritem collection within them. Coveritem naming is a function of the metric.

File-number and line-number values and anchors have been previously defined in "Code Coverage Data Models" on page 74.

For multi-bit assignments in a single expression, there may be one or more optional final bit-assignment naming items that identify the LHS bit indexes.

Multi-indexed LHS entities may repeat the **bit-index**# syntax.

Packed structures, or similar, with no natural bit-indexes, are flattened to a 0-referenced indexing scheme where language definitions exist to do this, for example, SystemVerilog.

6.5.4.2.1 UCIS_EXPR

Scope type component representation 2:

```
expr-scope-name :== [#expr#file-number#line-number#item-number# { bit-index# } | #expr#item-
    number# { bit-index# } | metric-name]
```

6.5.4.2.2 UCIS_EXPRBIN

Coveritem type component representation :7:

```
exprbin-name :== [metric-coveritem-name]
```

6.5.4.2.3 UCIS_COND

Scope type component representation 3:

```
cond-scope-name :== [ #cond#file-number#line-number#item-number#{bit-index#} | #cond#item-
    number# | metric-name]
```

6.5.4.2.4 UCIS_CONDBIN

Coveritem type component representation :8:

```
condbin-name :== [metric-coveritem-name]
```

6.5.4.2.5 Examples

SystemVerilog Expressions and Conditions

```
module top;                                          // line 1
 bit a,b,c,d,e,f,z;                                  // line 2
 always @ (b or c or d or e or f) begin              // line 3
  a=(b&((c&d) ? (e|f) : (e&f))); z=(b|(f?c:e));      // line 4
 end                                                 // line 5
 initial begin                                       // line 6
  #1; f = 1; c=1;                                    // line 7
  #1; d=1;                                           // line 8
  #1; b=1;                                           // line 9
  #1; e=1;                                           // line 10
  #1; f=0;                                           // line 11
 end                                                 // line 12
endmodule                                            // line 13
```

Expression coverage is inferred based on the presence of the top-level expression. Expression coverage naming is based on file, line, and item-numbers. Assuming a file-number of 1, the file/line/item triads from which the names will be derived are 1/4/1 and 1/4/2 for the expressions **(b&((c&d)?(e|f):(e&f)))** and **(b|(f?c:e))** with top-level operators **&** and **|** respectively.

6.5.4.3 Flat Model Examples

The flat structural model example shows collection of the UCIS:BITWISE_FLAT metric for the first expression and UCIS:TRUEUDP_FLAT for the second expression. This is purely illustrative; actual metric choice is application-determined and although it is not shown, each top-level expression could have multiple metric sub-scopes. Note that the first expression has 5 inputs, therefore the UCIS:FULL_FLAT metric collection would require 32 bins. The second expression has 4 inputs, UCIS:FULL_FLAT collection would require 16 bins. The actual metrics shown use fewer bins in both cases, but at the cost of unrecoverable information loss.

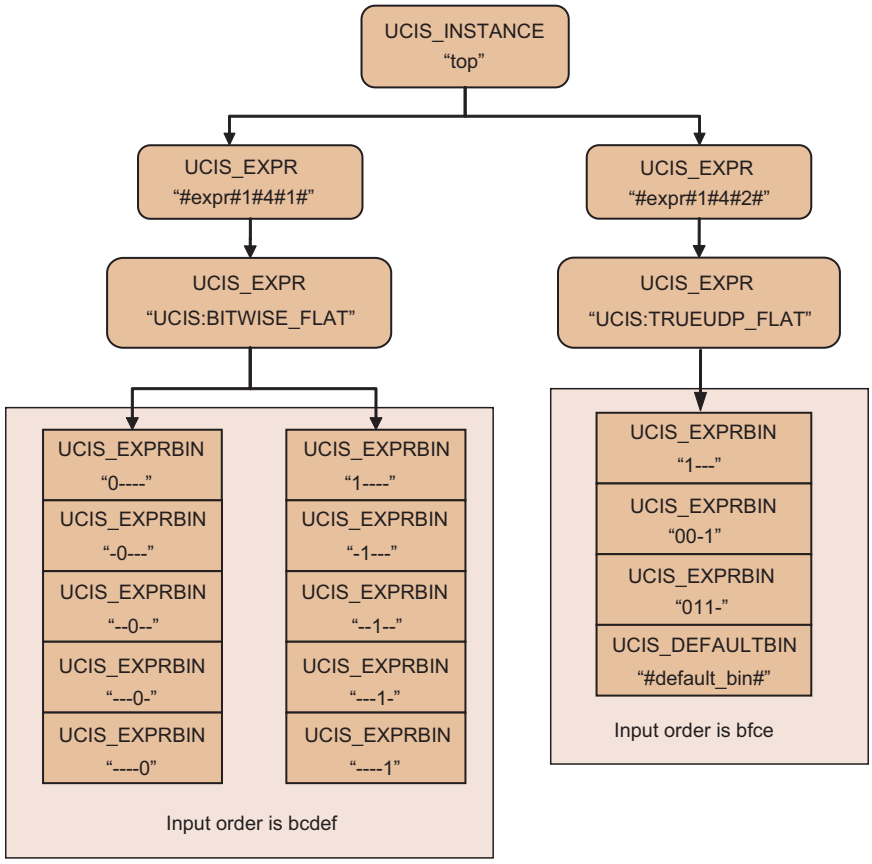

Figure 26—Example of condition and expression flat model

6.5.4.4 Hierarchical Model Examples

The hierarchical structural model, for the above example, shall be as follows. Note that naming components of top-level expressions is consistent across these flat and hierarchical structural models.

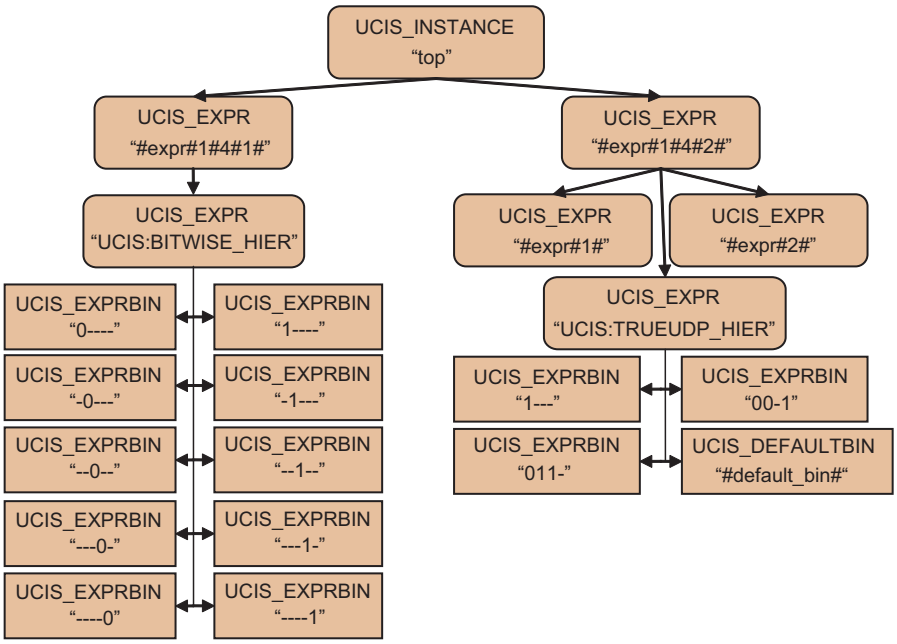

Figure 27—Example of condition and expression hierarchical model

The next example shows UCIS:STD_HIER and UCIS:OBS_HIER metrics within a condition hierarchy.

```
if (a || (b && c) || (d && e) )
//  1      ---2--       --3---
```

In this example, the operands are numbered as shown in the comment and the non-metric sub-scopes are named for the operand number. Although the diagrams show UCIS:STD_HIER and UCIS:OBS_HIER, it is valid for any of the input contribution metrics to be collected under the subscopes.

The UCIS:STD_HIER metric contains the sensitized vector bins as shown in the diagram below.

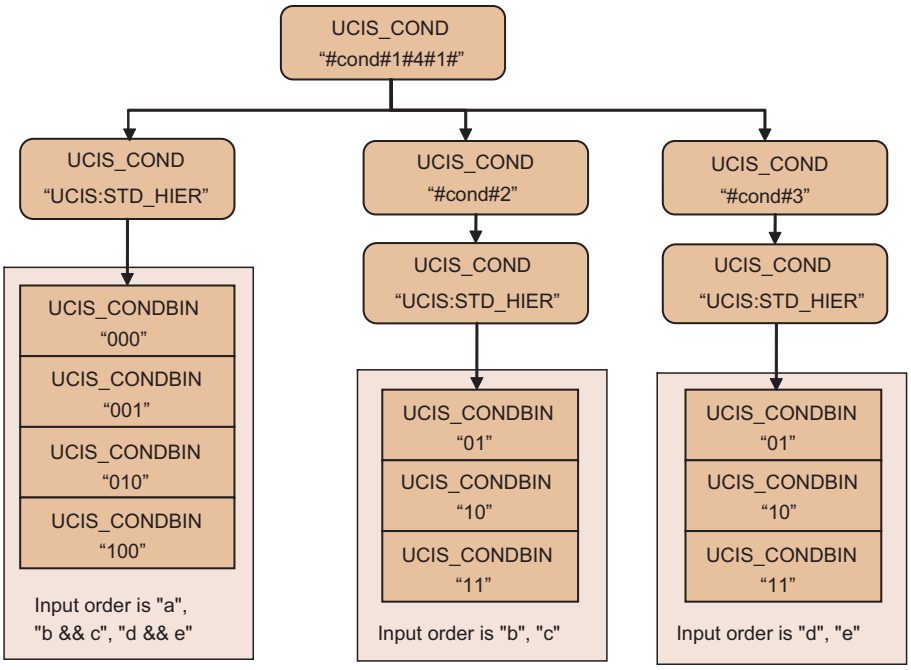

Figure 28—Example of condition with sensitized vector bins

Assuming the top UCIS_COND scope is in the top-level instance "top", the UIDs of the scopes and items in the figure above are:

```
"/4:top/3:#cond#1#4#1#
"/4:top/3:#cond#1#4#1#/3:UCIS:STD_HIER
"/4:top/3:#cond#1#4#1#/3:#cond#2
"/4:top/3:#cond#1#4#1#/3:#cond#3
"/4:top/3:#cond#1#4#1#/3:UCIS:STD_HIER/:8:000
"/4:top/3:#cond#1#4#1#/3:UCIS:STD_HIER/:8:001
"/4:top/3:#cond#1#4#1#/3:UCIS:STD_HIER/:8:010
"/4:top/3:#cond#1#4#1#/3:UCIS:STD_HIER/:8:100
"/4:top/3:#cond#1#4#1#/3:#cond#2/3:UCIS:STD_HIER/:8:01
"/4:top/3:#cond#1#4#1#/3:#cond#2/3:UCIS:STD_HIER/:8:10
"/4:top/3:#cond#1#4#1#/3:#cond#2/3:UCIS:STD_HIER/:8:11
"/4:top/3:#cond#1#4#1#/3:#cond#3/3:UCIS:STD_HIER/:8:01
"/4:top/3:#cond#1#4#1#/3:#cond#3/3:UCIS:STD_HIER/:8:10
"/4:top/3:#cond#1#4#1#/3:#cond#3/3:UCIS:STD_HIER/:8:11
```

The UCIS:OBS_HIER metric contains the observable vector bins as shown in the diagram below. The third string shown in the figure (e.g., "a#b && c#d && e") is the value of the optional property enum UCIS_STR_EXPR_TERMS that records the operand strings, separated by the '#' character.

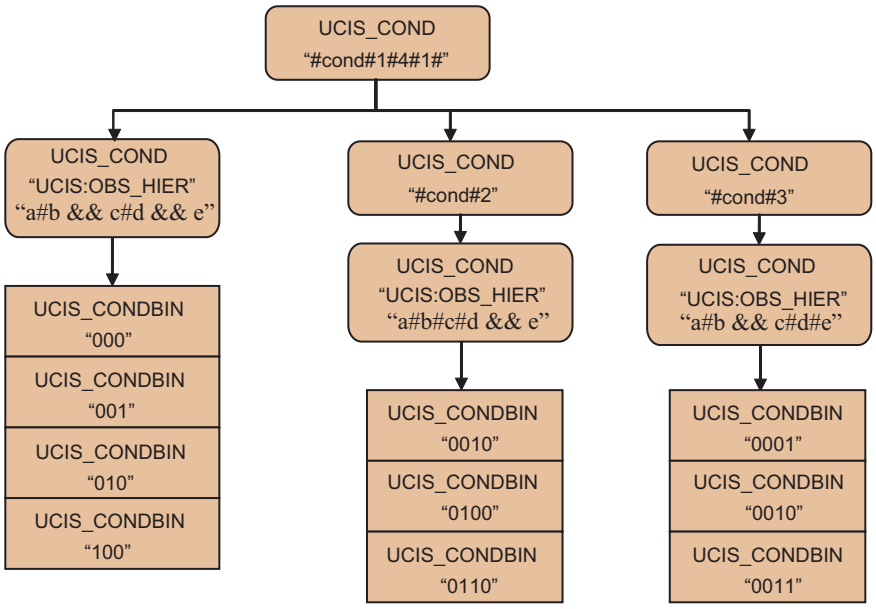

Figure 29—Vector bins

The next example shows the UCIS_CONTROL metric for the following assignment statement:

```
a = b || ((c | d) && e); // line 4
```

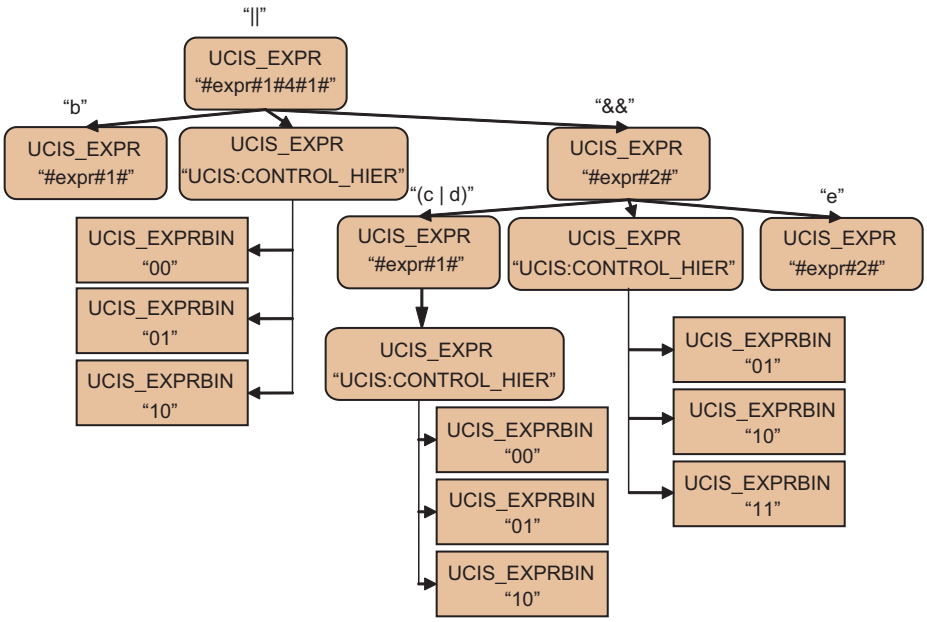

Figure 30—UCIS_CONTROL metric for an assignment statement

Assuming the top UCIS_EXPR scope is in the top-level instance "top", the UIDs of the scopes and items in the diagram above are:

```
"/4:top/2:#expr#1#4#1#
"/4:top/2:#expr#1#4#1#/2:#expr#1#
"/4:top/2:#expr#1#4#1#/2UCIS:CONTROL_HIER
"/4:top/2:#expr#1#4#1#/2:UCIS:CONTROL_HIER/:7:00
"/4:top/2:#expr#1#4#1#/2:UCIS:CONTROL_HIER/:7:01
"/4:top/2:#expr#1#4#1#/2:UCIS:CONTROL_HIER/:7:10
"/4:top/2:#expr#1#4#1#/2:#expr#2#
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:#expr#1#
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:#expr#1#/2:UCIS:CONTROL_HIER
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:#expr#1#/2:UCIS:CONTROL_HIER/:7:00
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:#expr#1#/2:UCIS:CONTROL_HIER/:7:01
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:#expr#1#/2:UCIS:CONTROL_HIER/:7:10
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:UCIS:CONTROL_HIER
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:UCIS:CONTROL_HIER/:7:01
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:UCIS:CONTROL_HIER/:7:10
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:UCIS:CONTROL_HIER/:7:11
"/4:top/2:#expr#1#4#1#/2:#expr#2#/2:#expr#2#
```

The next example shows the usage of the UCIS:BITWISE_CONTROL and UCIS:VECTOR metrics. Each bit of the multi-bit signal is scored separately.

```
a = b || ((c | d) && e); // line 4
         // b, c, d and e are all 2-bits wide
```

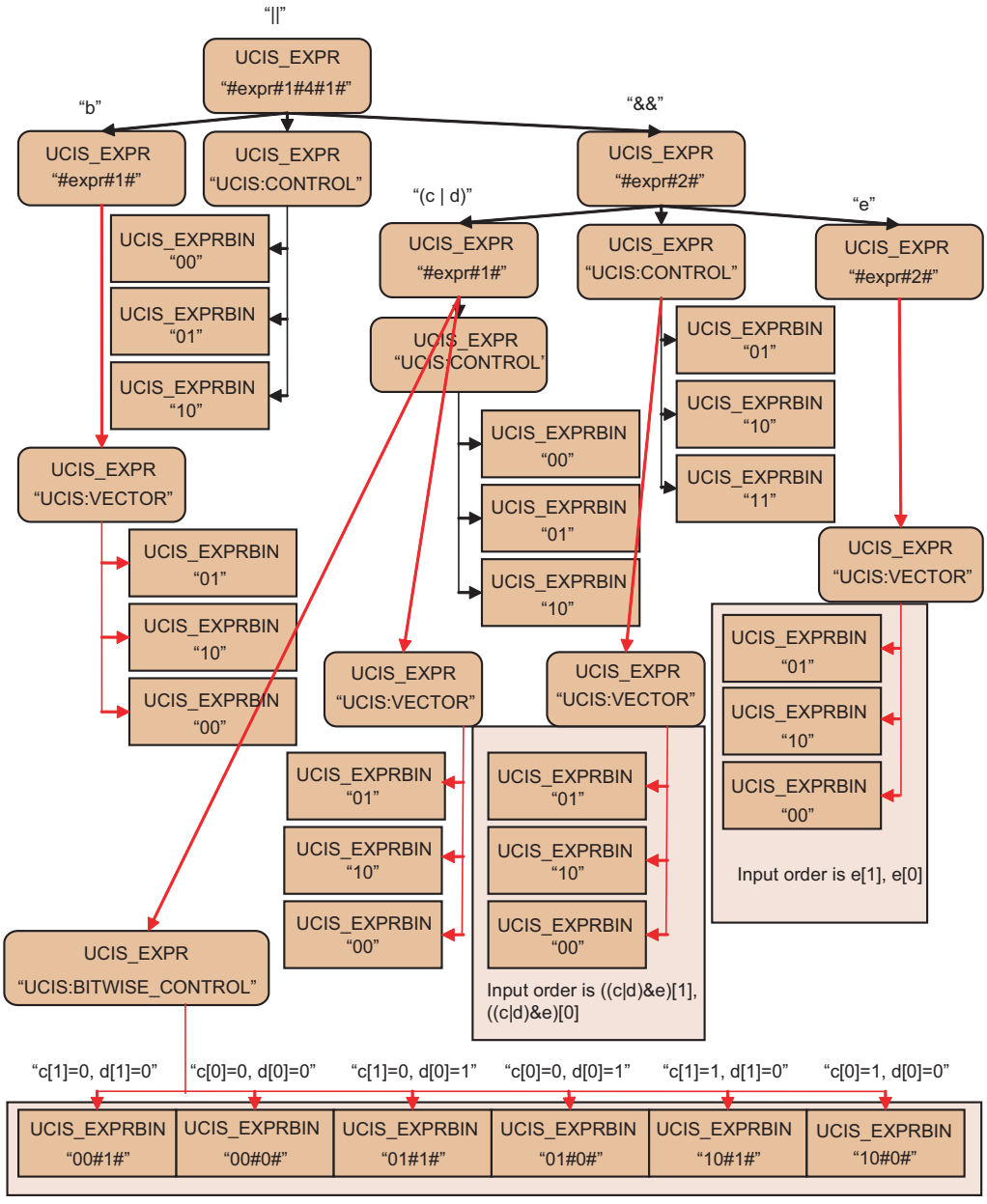

Figure 31—UCIS:BITWISE_CONTROL and UCIS:VECTOR metrics

6.5.4.5 A SystemVerilog multi-bit example

```
int ia, ib;
bit b1, b2;
reg [7:4] r1;

r1 = {r2[7:6]|r3[1:0], 0x01,(ia == ib) ? b1 : b2}; // file 1 line 97
```

Three expression scopes may be generated here, for r1[7], r1[6], and r1[4]. r1[5] is assigned to a literal constant and there is nothing to cover. The corresponding expression scope names would be:

```
#expr#1#97#1#7#, #expr#1#97#1#6# and #expr#1#97#1#4#
```

Coveritem name components depend on the metrics but may be derived from indexed bits in r2 or r3, from (ia == ib), or from b1 and b2. Examples would be r2[7], r2[6], r3[1], r3[0], ia == ib, b1 and b2 following the rules defined in “Source-derived name components” on page 37.

6.5.5 Toggle Coverage

6.5.5.1 Structural Model

Toggle coverage is applied to HDL objects with value such as variables and nets and tracks the changes in those values. The structural model is a two-scope hierarchy where the top UCIS_TOGGLE scope is named for the variable or net. UCIS_TOGGLE scopes may contain one or more expression-value metric scopes, or may contain bins directly. See "Metrics" on page 40 for details on metrics.

A variety of metrics are provided for TOGGLE coverage, depending on the variable type.

6.5.5.1.1 Single-bit variable toggle coverage metrics

Toggle coverage monitors changes in value in signal bits. A signal bit is fully covered under toggle coverage if it is observed to transition from the value 0 to the value 1, and from the value 1 to the value 0. Transitions to and from other bit values are ignored for purposes of toggle coverage. This is the UCIS:2STOGGLE metric.

Two alternate models of toggle coverage monitors are also defined. In the first alternate model, a signal bit is fully covered only if all six possible value transitions between the values Z, 0 and 1 are observed. Transitions to and from other values are ignored under this model of toggle coverage. This is the UCIS:ZTOGGLE metric.

In the second alternate model, a signal bit is fully covered only if all six possible value transitions between the values X, 0 and 1 are observed. Transitions to and from other values are ignored under this model of toggle coverage. This is the UCIS:XTOGGLE metric.

6.5.5.1.2 Integral and real toggle coverage metrics

Any integral- or real-valued variables can be modeled with the UCIS:VALCHANGE metric. Enumerated variables can be modeled with the UCIS:ENUM metric.

Empirically, a high proportion of UCISDB data is toggle coverage data in many UCISDBs, and this drives a need for optimized storage forms. There is an optimized form of toggle coverage where the metric association to the variable is implicit, and the intermediate scope is not present. The metric bins are stored directly under the top scope. For example, if the only metric of interest for an enumerated variable is the UCIS:ENUM metric, the intermediate scope may be stripped out and the enumerated element bins collected directly under the variable name. If the optimized form is used, the default metric is based on the value type as in the following table.

Table 6-5 — Toggle coverage metrics

Value type	Default toggle metric
Enumerated	UCIS:ENUM
Two-state single bit	UCIS:2STOGGLE
Four-state single bit	UCIS:ZTOGGLE
Integral variable	UCIS:VALCHANGE
Real variable	UCIS:VALCHANGE

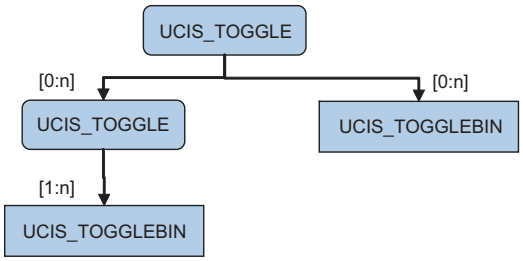

Figure 32—Toggle coverage

6.5.5.2 Naming Model

The top level UCIS_TOGGLE scopes in this context are named for the HDL variable, or variable components such as indexed bits or slices, or structure members. See "Source-derived name components" on page 37 for the rules as to how variable names are used as naming components.

The lowest UCIS_TOGGLE scope is named for the metric that defines the coveritem collection within it. Coveritem naming is a function of the metric, whether it is implicit or explicit.

6.5.5.2.1 UCIS_TOGGLE

Scope type component representation 0:

```
toggle-name :== [hdl-valued-object-name | index | metric-name ]
```

6.5.5.2.2 UCIS_TOGGLEBIN

Coveritem type component representation :9:

```
togglebinname :== metric-coveritem-name
```

The example below shows the UCIS:2STOGGLE coverage structure for simple one-bit signal, "my_state_int":

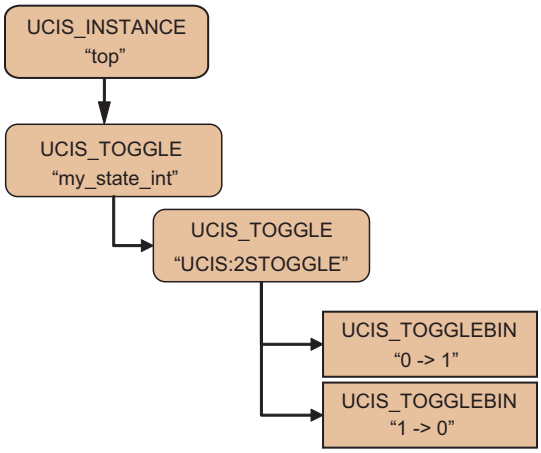

Figure 33—UCIS:2STOGGLE coverage structure

The UIDs associated with the above diagram are:

```
"/4:top"
"/4:top/0:my_state_int"
"/4:top/0:my_state_int/0:UCIS:2STOGGLE"
"/4:top/0:my_state_int/0:UCIS:2STOGGLE/:9:0->1"
"/4:top/0:my_state_int/0:UCIS:2STOGGLE/:9:1->0"
```

For UCIS:2STOGGLE, it is allowed to omit the metric type scope:

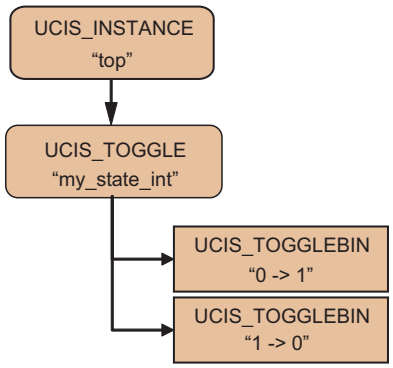

Figure 34—UCIS:2STOGGLE without metric tye scope

In this style, the UIDs would be:

```
"/4:top"
"/4:top/0:my_state_int"
"/4:top/0:my_state_int/:9:0->1"
"/4:top/0:my_state_int/:9:1->0"
```

An example of a multi-bit UCIS:2STOGGLE toggle coverage object like wire [1:0] w; in SystemVerilog.

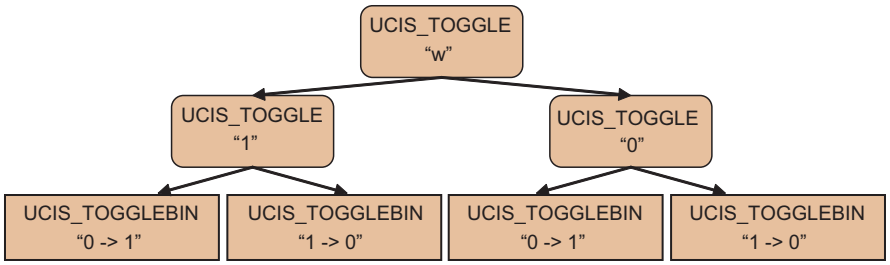

Figure 35—UCIS_TOGGLE in SystemVerilog

Assuming the top UCIS_TOGGLE scope is in the top-level instance "top", the UIDs of the scopes and items in the diagram above are:

```
"/4:top/0:w"
"/4:top/0:w/0:1"
"/4:top/0:w/0:1/:9:0->1"
"/4:top/0:w/0:1/:9:1->0"
"/4:top/0:w/0:0"
"/4:top/0:w/0:0/:9:0->1"
"/4:top/0:w/0:0/:9:1->0"
```

6.5.6 FSM Coverage

6.5.6.1 Structural Model

Finite-State Machine (FSM) coverage applies to FSM state variables and is modeled as a two-scope hierarchy. The top scope identifies the FSM variable and contains two metric scopes, one scope to collect state data and one scope to collect transition data.

There is a relationship between the states the FSM can have and the transitions it can make.

This relationship is recognized by the usage of specific, restricted, metric scope typing, and the provision of two utility API routines, ucis_ScopeIterate(UCIS_FSM_STATES) and ucis_CreateNextTransition(). A UCIS_FSM scope shall have a single UCIS_FSM_STATES scope and a single UCIS_FSM_TRANS scope.

The UCIS:STATE and UCIS:TRANSITION metrics are provided specifically for the FSM data model.

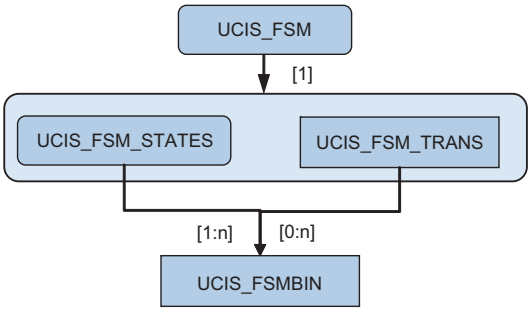

Figure 36—FSM coverage structuraal model

6.5.6.2 Naming Model

The top level UCIS_FSM scope is named for the HDL state variable. The UCIS_FSM_STATES and UCIS_FSM_TRANS metric sub-scopes are named for the metric that defines the coveritem collection within them. Coveritem naming is a function of the metric, see “Metrics” on page 40.

6.5.6.2.1 UCIS_FSM

Scope type component representation 22:

```
fsm-name :== user-hdl-variable-name
```

6.5.6.2.2 UCIS_FSM_STATES

Scope type component representation 29:

```
fsm-name :== UCIS:STATE
```

6.5.6.2.3 UCIS_FSM_TRANS

Scope type component representation 30:

```
fsm-name :== UCIS:TRANSITION
```

6.5.6.2.4 UCIS_FSMBIN

Coveritem type component representation :11:

```
fsmbin-name :== metric-coveritem-name
```

The property enum UCIS_INT_FSM_STATEVAL may be used to annotate each UCIS_FSMBIN in the UCIS_FSM_STATES scope. If used, this property records the value of the FSM state variable associated with the coveritem independently of the coveritem name used for universal object recognition.

6.5.6.3 Examples

SystemVerilog FSM

```
module top;                                 // line 1
 bit clk = 0;                               // line 2
 bit i = 0;                                 // line 3
 bit reset = 1;                             // line 4
 enum {stR, st0} state;                     // line 5
 always @ (posedge clk or posedge reset)    // line 6
  begin                                     // line 7
  if (reset)                                // line 8
   state = stR;                             // line 9
  else                                      // line 10
   case (state)                             // line 11
    stR: if (i==0) state = st0;             // line 12
   endcase                                  // line 13
  end                                       // line 14
 always #10 clk = ~clk;                     // line 15
 always @ (state) $display(state);          // line 16
 initial begin                              // line 17
  $display(state);                          // line 18
  @(negedge clk);                           // line 19
  @(negedge clk) reset = 0;                 // line 20
  @(negedge clk);                           // line 21
  $stop;                                    // line 22
 end                                        // line 23
endmodule                                   // line 24
```

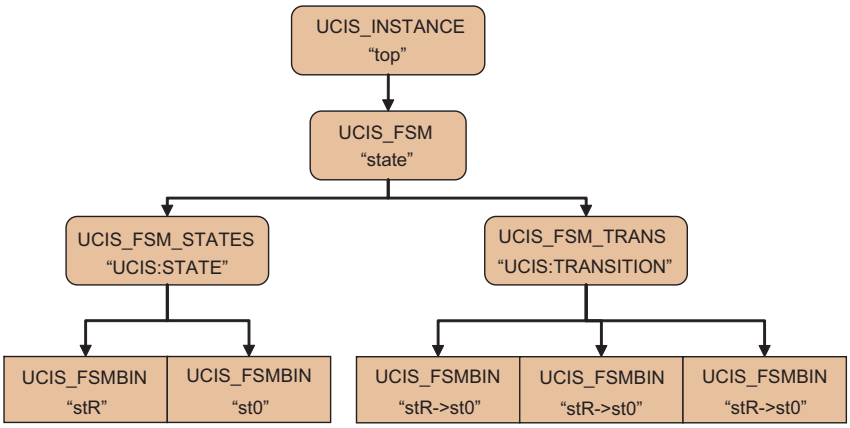

Figure 37—FSM coverage

Unique ID list for the diagram above:

```
/4:top
/4:top/22:state
/4:top/22:state/29:UCIS:STATE
/4:top/22:state/30:UCIS:TRANSITION
/4:top/22:state/29:UCIS:STATE/:11:stR
/4:top/22:state/29:UCIS:STATE/:11:st0
/4:top/22:state/30:UCIS:TRANSITION/:11:stR->st0
/4:top/22:state/30:UCIS:TRANSITION/:11:stR->stR
/4:top/22:state/30:UCIS:TRANSITION/:11:st0->stR
```

6.6 Other Models

6.6.1 SVA/PSL cover

Cover directives in PSL and cover statements in SystemVerilog are considered exactly the same. The cover data model also shares some features with the assertion data model.

6.6.1.1 Structural Model

Procedural cover statements are assigned exactly one UCIS_COVERBIN coveritem.

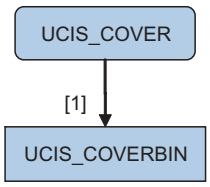

Figure 38—SVA/PSL cover model

6.6.1.2 Naming Model

6.6.1.2.1 UCIS_COVER

Scope type component representation 16:

```
cover-name :== [user-hdl-cover-label | tool-assigned-cover-name]
```

6.6.1.2.2 UCIS_COVERBIN

Coveritem type component representation :1:

```
coverbin-name :== coverbin
```

6.6.1.3 Example

SystemVerilog and PSL example

```
module top;                              // line 1
 bit a=0, b=0, clk=0;                    // line 2
 always #10 clk = ~clk;                  // line 3
 initial begin                           // line 4
  @(negedge clk);      b=1;              // line 5
  @(negedge clk); a=1; b=0;              // line 6
  @(negedge clk); a=0;                   // line 7
  @(negedge clk); $stop();               // line 8
 end                                     // line 9
 // psl default clock = rose(clk);       // line 10
 // psl pslcover: cover {b;a};           // line 11
 sequence a_after_b;                     // line 12
  @ (posedge clk) b ##1 a;               // line 13
 endsequence                             // line 14
 svacover: cover property(a_after_b);    // line 15
endmodule                                // line 16
```

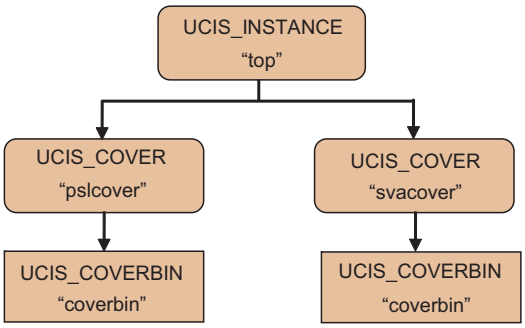

Figure 39—SystemVerilog and PSL

Unique ID list for the diagram above:

```
/4:top
/4:top/16:pslcover"
/4:top/16:svacover"
/4:top/16:pslcover/:1:coverbin
```

6.6.2 Assertion Coverage (SVA/PSL assert)

6.6.2.1 Structural Model

The UCIS_ASSERT scope models assertion data. Assertion data is different from coverage data in that the counts in the basic assertion coveritem have a negative rather than a positive connotation and therefore behave differently for coverage aggregation than coverage coveritem counts. A UCIS_ASSERT scope must have a UCIS_ASSERTBIN to record assertion failures, but other assertion behavior may also be counted. These additional coveritems are optional and are designed to track assertion passes, vacuous attempts, and so on. They are primarily intended for debugging the assertion itself. However, if present, the UCIS_PASSBIN counts towards assertion coverage. The UCIS_PEAKACTIVEBIN records the highest number of assertion threads active during the simulation run at any one time; in some sense it records the cost of the assertion.

Coveritem typing is used to indicate the meaning of the count; coveritem-names are redundantly equivalent to the coveritem types for this class of coverage, but are nonetheless required to be as specified. For example, the name of UCIS_PASSBIN is "passbin".

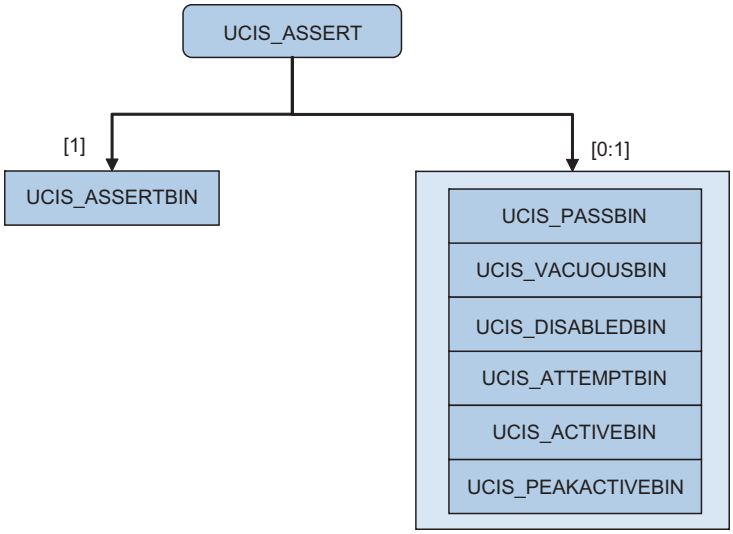

Figure 40—Assertion coverage structural model

6.6.2.2 Naming Model

6.6.2.2.1 UCIS_ASSERT

Scope type component representation 17:

```
assertion-name :== [user-hdl-assertion-label | tool-assigned-assertion-name]
```

6.6.2.2.2 UCIS_ASSERTBIN

Coveritem type component representation :2:

```
assertbin-name :== failbin
```

6.6.2.2.3 UCIS_PASSBIN

Coveritem type component representation :10:

```
passbin-name :== passbin
```

6.6.2.2.4 UCIS_VACUOUSBIN

Coveritem type component representation :15:

```
vacuousbin-name :== vacuousbin
```

6.6.2.2.5 UCIS_DISABLEDBIN

Coveritem type component representation :16:

```
disabledbin-name :== disabledbin
```

6.6.2.2.6 UCIS_ATTEMPTBIN

Coveritem type component representation :17:

```
attemptbin-name :== attemptbin
```

6.6.2.2.7 UCIS_ACTIVEBIN

Coveritem type component representation :18:

```
activebin-name :== activebin
```

6.6.2.2.8 UCIS_PEAKACTIVEBIN

Coveritem type component representation :22:

```
peakactivebin-name :== peakactivebin
```

6.6.2.3 Example

SystemVerilog and PSL example. The example shows only assertion failures for "pslassert", and failures, attempts and passes for "svaassert". The tool interface determines whether any of the other bins are also collected.

```
module top;                                      // line 1
 bit a=0, b=0, clk=0;                            // line 2
 always #10 clk = ~clk;                          // line 3
 initial begin                                   // line 4
  @(negedge clk);      b=1;                      // line 5
  @(negedge clk); a=1; b=0;                      // line 6
  @(negedge clk); a=0;                           // line 7
  @(negedge clk); $stop();                       // line 8
 end                                             // line 9
 // psl default clock = rose(clk);               // line 10
 // psl pslassert: assert always {b} |=> {a};    // line 11
 property a_after_b;                             // line 12
```

```
  @ (posedge clk) b |-> a;                    // line 13
 endproperty                                  // line 14
 svaassert: assert property(a_after_b);       // line 15
endmodule                                     // line 16
```

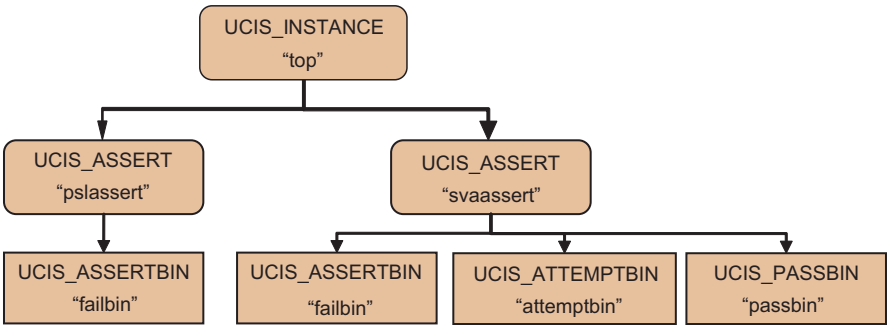

Figure 41—Example of SVA and PSL assertion coverage

Unique ID list for the diagram above:

```
/4:top
/4:top/17:pslassert"
/4:top/17:svaassert"
/4:top/17:pslassert/:2:failbin"
/4:top/17:svaassert/:2:failbin"
/4:top/17:svaassert/:2:attemptbin"
/4:top/17:svaassert/:2:passbin"
```

6.6.3 User Defined

The UCIS_USER scope and associated UCIS_USERBIN and UCIS_COUNT are provided for extensibility. The coverage aggregation rules for these types are that UCIS_USER and UCIS_USERBIN count towards basic coverage but that UCIS_COUNT coveritems do not, hence UCIS_COUNT coveritems can record integral count data for any non-coverage purpose. Otherwise no limits are placed on these objects except that they shall not be named with any string starting UCIS:. There is no universal object recognition available for the extensibility types.

6.6.3.0.1 UCIS_USER

Scope type component representation 10:

```
user-name :== extended-name
```

6.6.3.0.2 UCIS_USERBIN

Coveritem type component representation :12:

```
userbin-name :== user-assigned-name
```

6.6.3.0.3 UCIS_COUNT

Coveritem type component representation :13:

```
count-name :== user-assigned-name
```

7 Versioning

UCIS version information can be obtained from UCIS API implementation tool (simulator, formal tool etc.) and following objects:

- UCIS database in memory
- UCIS database in non-volatile form like file/directory
- UCIS history nodes

UCIS defines an abstract version handle, ucisVersionHandleT, to contain version information so that UCIS implementations have the flexibility to define their own internal implementation of version handle. Only one version handle per target shall be allowed. Versioning shall be applied and stored as a background task by API routines when a database or history node is created or stored. Access to the version information via API shall be read-only.

The following APIs have been defined to obtain version handle from, UCIS API implementation tool, UCIS database in-memory, UCIS database in non-volatile form and UCIS history node, respectively.

```
ucisVersionHandleT apiVersion = ucis_GetAPIVersion();
ucisVersionHandleT dbVersion = ucis_GetDBVersion(ucisT db);
ucisVersionHandleT fileVersion = ucis_GetFileVersion(char*
   filename_or_directory_name);
ucisVersionHandleT hVersion = ucis_GetHistoryNodeVersion(ucisT db,ucisHistoryNodeT
   hnode);
```

UCIS pre-defines certain version properties that can be extracted from version handle through ucis_GetVersionStringProperty API.

```
const char* ucis_GetVersionStringProperty (ucisVersionHandleT versionH,
   ucisStringPropertyEnumT property);
```

These version properties are defined through following enum constants of ucisStringPropertyEnumT.

Character-based data in the version strings shall be taken from the UTF-8 printable character set, excluding the backslash (‘\’) character. The strings shall be NULL terminated and NULL padded.

Table 7-6 defines versioning enums.

Table 7-6 — Versioning enums

Enumeration constant	Purpose
UCIS_STR_VER_STANDARD	Fixed string that identifies this standard: — UCIS
UCIS_STR_VER_STANDARD_VERSION	Choice of definitive version strings from the following list: — 2012
UCIS_STR_VER_VENDOR_ID	Vendor’ stock exchange symbol or any unique string. No absolute guarantee of uniqueness shall be implied or enforced by this standard. e.g. “MENT”, “CDNS”, “SNPS”, etc.
UCIS_STR_VER_VENDOR_TOOL	Vendor’ tool. e.g. “VCS”, “Incisive”, “Questa”, etc.
UCIS_STR_VER_VENDOR_TOOL_VERSION	Vendor’ tool version. e.g. “6.5c”, “Jun-12-2012”, etc.

8 UCIS API Functions

This chapter contains the following sections:

For an alphabetized list of functions, see "Function Index" on page 349.

8.1 Database creation and file management functions

A UCIS database (UCISDB) 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 once 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: ucis_Open(), which always creates a valid database handle. If a filename is given to ucis_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, if and only if no other calls have been made since ucis_Open() that require accessing the in-memory image.

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

  This operation can be performed at any time with the ucis_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 ucis_Close() function. After this call, the database handle is no longer valid.

8.1.1 Using write streaming mode.

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

— Create UCISDB attributes first. Creating UCISDB attributes at the beginning of the file is not enforced to allow the case of UCISDB attributes 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).

— Create History Nodes.

— Create scopes. Creating 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 OK, but accessing siblings is not (nor is it OK to access an ancestor’s siblings).

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

8.1.2 Using read streaming mode

The read streaming mode operates with callbacks. The persistent database is opened with a ucis_OpenReadStream() call that passes control to the UCIS system which 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. Note the following information on read streaming mode callback order:

— INITDB is always the first callback.

— UCISDB attributes created first in write streaming mode are available, as are UCISDB attributes created with in-memory mode.
— All TEST callbacks follow; after the next non-TEST callback there will be no more TEST callbacks.
— DU callbacks must precede their first associated instance SCOPE callbacks, but need not immediately precede them.
— 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.
— ENDSCOPE callbacks correspond to SCOPE and DU callbacks and imply a "pop" in the nesting of scopes and design units.
— ENDDB callbacks can be used to access UCISDB attributes written at the end of the file, if created in write streaming modes.

8.1.3 Opening a UCISDB in stream mode

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

The read streaming function uses these data types which are defined in the header file (see Annex B, ’ Header file - normative reference” on page 317)

— ucisCBReasonT - the enumeration of callback reasons
— ucisCBReturnT - the enumeration of callback routine return statuses to control the callback scan
— ucisCBDataT - the data type of the object passed to the callback routine

8.1.4 ucis_Close

Purpose	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 ucis_OpenWriteStream(), this functional call has the side-effect of closing the output file.
Syntax	`int ucis_Close(` `  ucisT  db);`
Arguments	db Database.
Return value	Returns 0 if successful, or -1 if error.

8.1.5 ucis_Open

Purpose	Creates an in-memory database, optionally populating it from the specified file.
Syntax	`ucisT ucis_Open(` `  const char*  name);`
Arguments	*name* File system path.
Return value	Returns a database handle if successful, or NULL if error.

8.1.6 ucis_OpenFromInterchangeFormat

Purpose	Creates an in-memory database populating it from the specified interchange format file.
Syntax	`ucisT ucis_OpenFromInterchangeFormat(` `  const char*  name);`
Arguments	*name* File system path.
Return value	Returns a database handle if successful, or NULL if error.

8.1.7 ucis_GetPathSeparator

Purpose	Returns the path separator for the specified database, or -1 if error.
Syntax	`char ucis_GetPathSeparator(` `  ucisT  db);`
Arguments	*db* Database.
Return value	Returns the path separator character for the database, or '\0' for error.

8.1.8 ucis_SetPathSeparator

Purpose	Sets the path separator for the specified database. The path separator is stored with the persistent form of the database.

Syntax	`int ucis_SetPathSeparator(`
	`  ucisT  db,`
	`  char  separator);`

Arguments	*db* Database.
	separator Path separator.
Return value	Returns 0 if successful, or -1 if error.

8.1.9 ucis_OpenReadStream

Purpose	Opens a database for streaming read mode from the specified file.

Syntax	`int ucis_OpenReadStream(`
	`  const char*  name,`
	`  ucis_CBFuncT  cbfunc,`
	`  void*  userdata);`

Arguments	*name* File system path.
	cbfunc User-supplied callback function.
	userdata User-supplied function data.
Return value	Returns 0 if successful, or -1 if error.

8.1.9.1 Typedef for ucis_OpenReadStream()

```
typedef ucisCBReturnT (*ucis_CBFuncT) \
   (void* userdata, ucisCBDataT* cbdata);
```

8.1.10 ucis_OpenWriteStream

Purpose	Opens data in write streaming mode, overwriting the specified file.
Syntax	`ucisT ucis_OpenWriteStream(` `  const char*  name);`
Arguments	*name* File system path (write permission must exist for the file).
Return value	Returns a restricted database handle if successful, or NULL if error.

8.1.11 ucis_WriteStream

Purpose	Finishes a write of current object to the persistent database file in write streaming mode. This operation is like a *flush*, which completes the write of whatever was most recently created in write streaming mode. Multiple ucis_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 ucis_OpenWriteStream().
Syntax	`int ucis_WriteStream(` `  ucisT  db);`
Arguments	*db* Database.
Return value	Returns 0 if successful, or -1 if error.

8.1.12 ucis_WriteStreamScope

Purpose	Finishes a write of the current scope (similar to the *flush* operation of ucis_WriteStream) and *pops* the stream to the parent scope. (i.e., terminates the current scope and reverts to its parent). Objects created after this belong to the parent scope of the scope just ended. Unlike ucis_WriteStream, this function cannot be called benignly multiple times as it always causes a reversion to the parent scope. This process is the write streaming analogue of the UCIS_REASON_ENDSCOPE callback in read streaming mode. The specified database handle must have been previously opened with ucis_OpenWriteStream().
Syntax	`int ucis_WriteStreamScope(` `  ucisT  db);`
Arguments	*db* Database.
Return value	Returns 0 if successful, or -1 if error.

8.1.13 ucis_Write

Purpose	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.	

Syntax	`int ucis_Write(`
	`  ucisT  db,`
	`  const char*  file,`
	`  ucisScopeT  scope,`
	`  int  recurse,`
	`  int  covertype);`	

Arguments	*db*	Database. The database handle *db* cannot have been opened for one of the streaming modes.
	file	File name (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 "API interface to stored names and primary keys" on page 50) to save or -1 for everything.
Return value	Returns 0 if successful, or -1 if error.	

8.1.14 ucis_WriteToInterchangeFormat

Purpose	Copies the entire in-memory database or the specified subset of the in-memory database to a persistent form stored in the specified file in the interchange format, overwriting the specified file.	

Syntax	`int ucis_WriteToInterchangeFormat(`
	`  ucisT  *db*,`
	`  const char*  *file*,`
	`  ucisScopeT  *scope*,`
	`  int  *recurse*,`
	`  int  *covertype*);`	

Arguments	*db*	Database. The database handle *db* cannot have been opened for one of the streaming modes.
	file	File name (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 "API interface to stored names and primary keys" on page 50) to save or -1 for everything.
Return value	Returns 0 if successful, or -1 if error.	

8.2 Error handler functions

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

Alternatively, you can check function return values. In general, functions that return a handle return a NULL or invalid handle on error. Otherwise, they return the handle. Functions that return an int value return non-zero on error, 0 otherwise.

The function prototype for the ucis_ErrorHandler routine is defined in the header file, see Annex B, ' Header file - normative reference" on page 317. Supporting data types ucisMsgSeverityT, which enumerates the message severities, and ucisErrorT, which collates all the available message information, are also defined.

8.2.1 ucis_RegisterErrorHandler

Purpose	Registers an error handler that is called whenever an API error occurs.	

Syntax	`void ucis_RegisterErrorHandler(`
	`  ucis_ErrorHandler  errHandle,`
	`  void*  userdata);`	

Arguments	*errHandle*	Error handler handle.
	userdata	User-specified data for the error handler.
Return value	None	

8.3 Property functions

This section identifies property functions for integer, string, real, and object handler properties. The property values are enumerated in the types ucisIntPropertyEnumT, ucisStringPropertyEnumT, and ucisRealPropertyEnumT. See the header file (Annex B, ’ Header file - normative reference” on page 317) for these definitions.

8.3.1 Integer properties

Table 8-1 defines the integer properties.

Table 8-1 — Integer Properties

Enumeration Constant	Purpose	Object
UCIS_INT_IS_MODIFIED	Modified since opening stored UCISDB (In-memory and read only)	ucisT (ucisScopeT = NULL, coveritem = -1)
UCIS_INT_MODIFIED_SINCE_SIM	Modified since end of simulation run (In-memory and read only)	ucisT (ucisScopeT = NULL, coveritem = -1)
UCIS_INT_NUM_TESTS	Number of test history nodes (UCIS_HISTORYNODE_TEST) in UCISDB	ucisT (ucisScopeT = NULL, coveritem = -1)
UCIS_INT_SCOPE_WEIGHT	Scope weight	ucisScopeT
UCIS_INT_SCOPE_GOAL	Scope goal	ucisScopeT
UCIS_INT_SCOPE_SOURCE_TYPE	Scope source type (ucisSourceT)	ucisScopeT
UCIS_INT_NUM_CROSSED_CVPS	Number of coverpoints in a cross (read only)	ucisScopeT (UCIS_CROSS)
UCIS_INT_SCOPE_IS_UNDER_DU	Scope is underneath design unit scope (read only)	ucisScopeT
UCIS_INT_SCOPE_IS_UNDER_COVERINSTANCE	Scope is underneath covergroup instance (read only)	ucisScopeT (UCIS_CVG_SCOPE)
UCIS_INT_SCOPE_NUM_COVERITEMS	Number of coveritems underneath scope (read only)	ucisScopeT
UCIS_INT_SCOPE_NUM_EXPR_TERMS	Number of input ordered expr term strings delimited by '#'	ucisScopeT (UCIS_EXPR, UCIS_COND)
UCIS_INT_TOGGLE_TYPE	Toggle type (ucisToggleTypeT)	ucisScopeT (UCIS_TOGGLE)
UCIS_INT_TOGGLE_DIR	Toggle direction (ucisToggleDirT)	ucisScopeT (UCIS_TOGGLE)
UCIS_INT_TOGGLE_COVERED	Toggle object is covered	ucisScopeT (UCIS_TOGGLE)
UCIS_INT_BRANCH_HAS_ELSE	Branch has an 'else' coveritem	ucisScopeT (UCIS_BRANCH, UCIS_BLOCK)
UCIS_INT_BRANCH_ISCASE	Branch represents 'case' statement	ucisScopeT (UCIS_BRANCH, UCIS_BLOCK)
UCIS_INT_COVER_GOAL	Coveritem goal	ucisCoverTypeT
UCIS_INT_COVER_LIMIT	Coverage count limit for coveritem	ucisCoverTypeT
UCIS_INT_COVER_WEIGHT	Coveritem weight	ucisCoverTypeT

Table 8-1 — Integer Properties (Continued)

Enumeration Constant	Purpose	Object
UCIS_INT_TEST_STATUS	Test run status (ucisTestStatusT)	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_INT_TEST_COMPULSORY	Test run is compulsory	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_INT_STMT_INDEX	Index or number of statement on a line	ucisCoverTypeT
UCIS_INT_BRANCH_COUNT	Total branch execution count	ucisScopeT (UCIS_BRANCH, UCIS_BLOCK)
UCIS_INT_FSM_STATEVAL	FSM state value	ucisCoverTypeT (UCIS_FSMBIN) underneath UCIS_FSM_STATES
UCIS_INT_CVG_ATLEAST	Covergroup at_least option	ucisScopeT (UCIS_CVG_SCOPE)
UCIS_INT_CVG_AUTOBINMAX	Covergroup auto_bin_max option	ucisScopeT (UCIS_CVG_SCOPE)
UCIS_INT_CVG_DETECTOVERLAP	Covergroup detect_overlap option	ucisScopeT (UCIS_CVG_SCOPE)
UCIS_INT_CVG_NUMPRINTMISSING	Covergroup cross_num_print_missing option	ucisScopeT (UCIS_CROSS)
UCIS_INT_CVG_STROBE	Covergroup strobe option	ucisScopeT (UCIS_CVG_SCOPE)
UCIS_INT_CVG_PERINSTANCE	Covergroup per_instance option	ucisScopeT (UCIS_COVERINSTANCE, UCIS_COVERGROUP)
UCIS_INT_CVG_GETINSTCOV	Covergroup get_inst_coverage option	ucisScopeT (UCIS_COVERINSTANCE, UCIS_COVERGROUP)
UCIS_INT_CVG_MERGEINSTANCES	Covergroup merge_instances option	ucisScopeT (UCIS_COVERINSTANCE, UCIS_COVERGROUP)

8.3.2 String properties

Table 8-2 defines the string properties.

Table 8-2 — String Properties

Enumeration Constant	Purpose	Object
UCIS_STR_DU_SIGNATURE	Signature of design unit scope (read only).	ucisScopeT
UCIS_STR_FILE_NAME	UCISDB file/directory name (read only).	ucisT (ucisScopeT = NULL, coveritem = -1)
UCIS_STR_SCOPE_NAME	Scope name.	ucisScopeT
UCIS_STR_SCOPE_HIER_NAME	Hierarchical scope name.	ucisScopeT
UCIS_STR_INSTANCE_DU_NAME	Instance' design unit name.	ucisScopeT (UCIS_INSTANCE, UCIS_COVERINSTANCE)
UCIS_STR_UNIQUE_ID	Scope or coveritem unique-id (read only).	ucisScopeT, ucisCoverTypeT
UCIS_STR_VER_STANDARD	Standard (Currently fixed to be "UCIS").	ucisVersionHandleT
UCIS_STR_VER_STANDARD_VERSION	Version of standard (e.g. "2012", etc).	ucisVersionHandleT
UCIS_STR_VER_VENDOR_ID	Vendor id (e.g. "CDNS", "MENT", "SNPS", etc).	ucisVersionHandleT
UCIS_STR_VER_VENDOR_TOOL	Vendor tool (e.g. "Incisive", "Questa", "VCS", etc).	ucisVersionHandleT
UCIS_STR_VER_VENDOR_VERSION	Vendor tool version (e.g. "6.5c", "Jun-12-2012", etc).	ucisVersionHandleT
UCIS_STR_GENERIC	Miscellaneous string data.	ucisScopeT, ucisCoverTypeT, ucisVersionHandleT
UCIS_STR_ITH_CROSSED_CVP_NAME	I^{th} coverpoint name of a cross.	ucisScopeT (UCIS_CROSS)
UCIS_STR_HIST_CMDLINE	Test run command line.	ucisHistoryNodeT
UCIS_STR_HIST_LOG_NAME	Logical name of history node	ucisHistoryNodeT
UCIS_STR_HIST_PHYS_NAME	Physical file or directory name of history node	ucisHistoryNodeT
UCIS_STR_HIST_RUNCWD	Test run working directory.	ucisHistoryNodeT
UCIS_STR_HIST_TOOLCATEGORY	Set of pre-defined values (UCIS:Simulator, UCIS:Formal, UCIS:Analog, UCIS:Emulator, UCIS:Merge, UCIS:Comparison)	ucisHistoryNodeT
UCIS_STR_COMMENT	Comment.	ucisScopeT (UCIS_CVG_SCOPE), ucisHistoryNodeT
UCIS_STR_TEST_TIMEUNIT	Test run simulation time unit.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_STR_TEST_DATE	Test run date.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)

Table 8-2 — String Properties (Continued)

Enumeration Constant	Purpose	Object
UCIS_STR_TEST_SIMARGS	Test run simulator arguments.	ucisHistoryNodeT
UCIS_STR_TEST_USERNAME	Test run user name.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_STR_TEST_NAME	Test run name.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_STR_TEST_SEED	Test run seed.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_STR_TEST_HOSTNAME	Test run hostname.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_STR_TEST_HOSTOS	Test run hostOS.	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)
UCIS_STR_EXPR_TERMS	Input ordered expr term strings delimited by '#'.	ucisScopeT (UCIS_EXPR, UCIS_COND)
UCIS_STR_TOGGLE_CANON_NAME	Toggle object canonical name.	ucisScopeT (UCIS_TOGGLE)
UCIS_STR_UNIQUE_ID_ALIAS	Scope or coveritem unique-id alias.	ucisScopeT, ucisCoverTypeT
UCIS_STR_DESIGN_VERSION_ID	Version of the design or elaboration-id.	ucisT (ucisScopeT = NULL, coveritem = -1)

8.3.3 Real properties

Table 8-3 defines the real properties.

Table 8-3 — Real Properties

Enumeration Constant	Purpose	Object
UCIS_REAL_CVG_INST_AVERAGE	Average-of-instances coverage percentage for a covergroup type	ucisScopeNodeT
UCIS_REAL_HIST_CPUTIME	Test run CPU time	ucisHistoryNodeT
UCIS_REAL_TEST_COST	Implementation-defined	ucisHistoryNodeT(UCIS_HISTORYNODE_TEST)
UCIS_REAL_TEST_SIMTIME	Test run simulation time	ucisHistoryNodeT (UCIS_HISTORYNODE_TEST)

8.3.4 Object Handle properties

Table 8-4 defines the object handle properties.

Table 8-4 — Object Handle Properties

Enumeration Constant	Purpose	Object
UCIS_HANDLE_SCOPE_PARENT	Parent scope	ucisScopeT
UCIS_HANDLE_SCOPE_TOP	Top (root) scope	ucisScopeT
UCIS_HANDLE_INSTANCE_DU	Instance' design unit scope	ucisScopeT (UCIS_INSTANCE, UCIS_COVERINSTANCE)
UCIS_HANDLE_HIST_NODE_PARENT	Parent history node	ucisHistoryNodeT
UCIS_HANDLE_HIST_NODE_ROOT	Top (root) history node	ucisHistoryNodeT

8.3.5 ucis_GetIntProperty

Purpose	Get integer properties from a UCISDB.	

Syntax	`int ucis_GetIntProperty (`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  ucisIntPropertyEnumT  *property*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem in its parent scope. If *coverindex* is negative, apply integer property on the scope.
	property	Integer property identification, taken from ucisIntPropertyEnumT definition. See the header file at Annex B, ’ Header file - normative reference” on page 317.
Return value	Returns value of the integer property, or -1 if error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on a specific scope. However, if the return value is -1 because an integer property’s value is indeed -1, the error handler, if any, shall not be called.	
Example	`int ret_val = ucis_GetIntProperty (db, NULL, -1, UCIS_INT_IS_MODIFIED);`	

8.3.6 ucis_SetIntProperty

Purpose	Set integer properties in a UCISDB.	

Syntax	`int ucis_SetIntProperty (`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  ucisIntPropertyEnumT  *property*,`
	`  int  *value*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem in its parent scope. If *coverindex* is negative, apply integer property on the scope.
	property	Integer property enum whose value is to be set.
	value	Integer value of the property.
Return value	Returns 0 if successful, or -1 if error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on a specific object.	
Example	`int ret_val = ucis_SetIntProperty (db, NULL, -1, UCIS_INT_SUPPRESS_MODIFIED, 1);`	

8.3.7 ucis_GetRealProperty

Purpose	Get real properties from a UCISDB.	

Syntax	`double ucis_GetRealProperty(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  ucisRealPropertyEnumT  *property*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem in its parent scope. If *coverindex* is negative, apply the real property on the scope.
	property	Real property enum (defined above) whose value is to be set.
Return value	Returns value of the real property, or -1 if error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on a specific scope. However, if the return value is -1 because a real property's value is indeed -1, the error handler, if any, shall not be called.	

8.3.8 ucis_SetRealProperty

Purpose	Set real properties in a UCISDB.	

Syntax	`int ucis_SetRealProperty(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  ucisRealPropertyEnumT  *property*,`
	`  double  *value*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem in its parent scope. If *coverindex* is negative, apply the real property on the scope.
	property	Real property enum whose value is to be set.
	value	Real value of the property.
Return value	Returns 0 if successful, or -1 if error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on a specific object.	

8.3.9 ucis_GetStringProperty

Purpose	Get string (const char*) properties from a UCISDB.	

Syntax	`const char* ucis_GetStringProperty (`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  ucisStringPropertyEnumT  *property*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem in its parent scope. If *coverindex* is negative, apply the string property on the scope.
	property	String property identification, taken from ucisStringPropertyEnumT definition. See the header file at Annex B, ’ Header file - normative reference” on page 317.
Return value	Returns a NULL-terminated value of the string property, or a NULL if none or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on a specific object. However, if the return value is NULL because a string property’ value was not implicitly or explicitly set, the error handler, if any, shall not be called.	
Example	`const char *fname = ucis_GetStringProperty (db, NULL, -1, UCIS_STR_FILENAME);`	

8.3.10 ucis_SetStringProperty

Purpose	Set string (const char*) properties into a UCISDB.

Syntax	`int ucis_SetStringProperty (`
	`  ucisT  *db*,`
	`  ucisObjT  obj,`
	`  int  *coverindex*,`
	`  ucisStringPropertyEnumT  *property*,`
	`  const char*  *value*);`

Arguments	*db* — Database. *obj* — Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute). *coverindex* — Index of coveritem in its parent scope. If *coverindex* is negative, apply the string property on the scope. *property* — String property enum whose value is to be set. *value* — NULL-terminated string value of the property.
Return value	Returns 0 if successful, or -1 if error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on a specific object.
Example	`int ret_val = ucis_SetStringProperty (db, scope, -1, UCIS_STR_NAME, "top");`

8.3.11 ucis_GetHandleProperty

Purpose	Get scope handle (ucisScopeT) associated with another scope from a UCISDB.

Syntax	`ucisScopeT ucis_GetHandleProperty (`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  ucisHandleEnumT  *property*);`

Arguments	*db* — Database. *obj* — Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute). *property* — Handle property identification, taken from ucisHandleEnumT definition. See the header file at Annex B, ’ Header file - normative reference” on page 317.
Return value	Returns associated scope handle, or NULL if none or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific value on a specific scope. However, if the return value is NULL because a scope was not implicitly or explicitly set, the error handler, if any, shall not be called.
Example	`ucisScopeT parent = ucis_GetHandleProperty (db, scope,` `  UCIS_HANDLE_SCOPE_PARENT);`

8.3.12 ucis_SetHandleProperty

Purpose	Associate scope handle (ucisScopeT) with another scope into a UCISDB.	

Syntax	`int ucis_SetHandleProperty (`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  ucisHandleEnumT  *property*,`
	`  ucisScopeT  *value*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	property	Handle property identification, taken from ucisHandleEnumT definition. See the header file at Annex B, ’ Header file - normative reference” on page 317.
	value	Value (associated scope handle).
Return value	Returns 0 if successful, -1 if error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific value on a specific scope.	
Example	`int ret_val = ucis_SetHandleProperty (db, scope, UCIS_HANDLE_SCOPE_DU, duscope);`	

8.4 User-defined attribute functions

This is a simple set of facilities for providing user-defined attributes associated with objects in the database -- scopes, coveritems, or tests -- or with the database itself (global attributes). User-defined attributes are key-value pairs that may be traversed or looked up by key.

The data types used to manage attributes are ucisAttrTypeT and ucisAttrValueT and are defined in the header file. See Annex B, ’ Header file - normative reference” on page 317 for these definitions.

Note: For attributes of coveritems: Coveritems are identified by a combination of the parent scope handle and an integer index for the coveritem. To use the attribute functions for a scope only, the integer index must be set to -1.

Note: Memory management: Key/value string storage is maintained by the API. With "set" routines, which add key/value pairs, the strings passed in by the user are copied to storage maintained by the API. The user must not de-allocate individual strings returned by the API.

8.4.1 ucis_AttrAdd

Purpose	Adds the specified attribute (key/value) to the specified database object or global attribute list. The attribute value is copied to the system.	

Syntax	`int ucis_AttrAdd(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  const char*  *key*,`
	`  ucisAttrValueT*  *value*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem. If *obj* is ucisScopeT, specify -1 for scope.
	key	Attribute key.
	value	Attribute value.
Return value	Returns 0 if successful, or -1 if error.	

8.4.2 ucis_AttrMatch

Purpose	Returns the value of the attribute of the specified object (or global attribute) that matches the specified key, or NULL if error.	

Syntax	`ucisAttrValueT* ucis_AttrMatch(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  const char*  *key*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem. If *obj* is ucisScopeT, specify -1 for scope.
	key	Key or NULL to remove the first attribute.
Return value	Returns the value of the attribute of the specified object (or global attribute) that matches the specified key, or NULL if error.	

8.4.3 ucis_AttrNext

Purpose	Returns the next attribute key and gets the corresponding attribute value from the specified database object, or returns NULL when done traversing attributes.	

Syntax	`const char* ucis_AttrNext(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  const char*  *key*,`
	`  ucisAttrValueT**  *value*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem. If *obj* is ucisScopeT, specify -1 for scope.
	key	Previous key or NULL to get the first attribute.
	value	Attribute value returned.
Return value	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. 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;
ucisAttrValueT* value;
while (key = ucis_AttrNext(db,obj,-1,key,&value)) {
   printf("Attribute '%s' is ", key);
   print_attrvalue(value);
}
```

8.4.4 ucis_AttrRemove

Purpose	Removes the attribute that has the specified key from the specified database object or global attribute list.	

Syntax	`int ucis_AttrRemove(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  const char*  *key*);`	

Arguments	*db*	Database.
	obj	Object type: ucisScopeT, ucisHistoryNodeT, or NULL (for global attribute).
	coverindex	Index of coveritem. If *obj* is ucisScopeT, specify -1 for scope.
	key	Key or NULL to remove the first attribute.
Return value	Returns 0 if successful, or -1 if error.	

8.5 Scope management functions

8.5.1 Hierarchical object APIs

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

The scope typing system is based on a 64-bit one-hot type system. The scope type definitions are in the header file, as also are some related masking and multiple type selection definitions.

The bitwise ucisFlagsT and enumerated ucisSourceT data types are also used within the scope management routines. They define the flags and source language related to the scope respectively. Note that the source type value may affect how identifiers are processed, for example with respect to escaping syntax or case sensitivity.

See Annex B, ’ Header file - normative reference” on page 317 for definitions of these items.

Note: For hierarchical identifiers: If a scope type is Verilog or SystemVerilog, Verilog escaped identifiers syntax is assumed for a path within that scope.

Note: For hierarchical identifiers: If a scope type is VHDL, VHDL extended identifiers may be used. The escaped identifier syntax is sensitive to the scope type so that escaped identifiers may appear in the user's accustomed syntax.

Note: char* attributes can be omitted with a NULL value.

Note: int attributes can be omitted with a negative value.

Note: If a scope type is VHDL, entity, architecture, and library may be encoded in the name.

8.5.2 ucis_CreateScope

Purpose	Creates a specified scope beneath a specified parent scope. Use ucis_CreateInstance for UCIS_INSTANCE or UCIS_COVERINSTANCE scopes.

Syntax	`ucisScopeT ucis_CreateScope(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  ucisSourceInfoT*  *srcinfo*,`
	`  int  *weight*,`
	`  ucisSourceT  *source*,`
	`  ucisScopeTypeT  *type*,`
	`  ucisFlagsT  *flags*);`

Arguments	*db* — Database. *parent* — Parent scope. If NULL, creates the root scope. *name* — Name to assign to scope. *srcinfo* — Source location appropriate to the UCISDB scope (encodes file, line, and item). *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.
Return value	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 ucis_WriteStream* call or the next ucis_Create* call.

8.5.3 ucis_RemoveScope

Purpose	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. Has no effect when db is in streaming mode.

Syntax	`int ucis_RemoveScope(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*);`

Arguments	*db* — Database. *scope* — Scope to remove.
Return value	Returns 0 for success, or -1 for error.

8.5.4 ucis_CallBack

Purpose	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.	

Syntax	`int ucis_CallBack(`
	`  ucisT  *db*,`
	`  ucisScopeT  *start*,`
	`  ucis_CBFuncT  *cbfunc*,`
	`  void*  *userdata*);`	

Arguments	*db*	Database.
	start	Starting scope or NULL to traverse entire database.
	cbfunc	User-supplied callback function.
	userdata	User-supplied function data.
Return value	Returns 0 if successful, or -1 with error.	

8.5.5 ucis_ComposeDUName

Purpose	Composes as design unit scope name for specified design unit. The ucis_ComposeDUName and ucis_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, the user must copy it.

Syntax	`const char*`
	`ucis_ComposeDUName(`
	`  const char*  library_name,`
	`  const char*  primary_name,`
	`  const char*  secondary_name);`

Arguments	*library_name* Library name.
	primary_name Primary name.
	secondary_name Secondary name.
Return value	Returns handle to the parsed design unit scope name for the specified component names, or -1 for error.

8.5.6 ucis_ParseDUName

Purpose	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 ucis_ComposeDUName and ucis_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, the user must copy it.

Syntax	`void ucis_ParseDUName(`
	`  const char*  du_name,`
	`  const char**  library_name,`
	`  const char**  primary_name,`
	`  const char**  secondary_name);`

Arguments	*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.
Return value	None

8.5.7 ucis_CreateCross

Purpose	Creates the specified cross scope under the specified parent (covergroup or cover instance) scope.	

Syntax	`ucisScopeT ucis_CreateCross(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  ucisSourceInfoT*  *fileinfo*,`
	`  int  *weight*,`
	`  ucisSourceT  *source*,`
	`  int  *num_points*,`
	`  ucisScopeT*  *points*);`	

Arguments	*db*	Database.
	parent	Parent scope: UCIS_COVERGROUP or UCIS_COVERINSTANCE.
	name	Name to assign to cross scope.
	fileinfo	Source location appropriate to the UCISDB scope (encodes file, line, and item).
	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.
Return value	Returns a scope handle for the cross, or NULL if error.	

8.5.8 ucis_CreateCrossByName

Purpose	Creates the specified cross scope under the specified parent (covergroup or cover instance) scope.	

Syntax	`ucisScopeT ucis_CreateCrossByName(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  ucisSourceInfoT*  *fileinfo*,`
	`  int  *weight*,`
	`  ucisSourceT  *source*,`
	`  int  *num_points*,`
	`  char**  *point_names*);`	

Arguments	*db*	Database.
	parent	Parent scope: UCIS_COVERGROUP or UCIS_COVERINSTANCE.
	name	Name to assign to cross scope.
	fileinfo	Source location appropriate to the UCISDB scope (encodes file, line, and item).
	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.
Return value	Returns a scope handle for the cross, or NULL if error.	

8.5.9 ucis_CreateInstance

Purpose	Creates an instance scope of the specified design unit type under the specified parent. Not supported in streaming modes; use ucis_CreateInstanceByName() in write streaming mode.	

Syntax	`ucisScopeT ucis_CreateInstance(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  ucisSourceInfoT*  *fileinfo*,`
	`  int  *weight*,`
	`  ucisSourceT  *source*,`
	`  ucisScopeTypeT  *type*,`
	`  ucisScopeT  *du_scope*,`
	`  ucisFlagsT  *flags*);`	

Arguments	*db*	Database.
	parent	Parent of instance scope. If NULL, creates a new root scope.
	name	Name to assign to scope.
	fileinfo	Source location appropriate to the UCISDB scope (encodes file, line, and item).
	weight	Weight to assign to the scope. Negative indicates no weight.
	source	Source of instance.
	type	Type of scope to create: UCIS_INSTANCE or UCIS_COVERINSTANCE.
	du_scope	Previously-created scope that is usually the design unit. If type is UCIS_INSTANCE, then du_scope has type UCIS_DU_*. If type is UCIS_COVERINSTANCE, then du_scope has type UCIS_COVERGROUP to capture the instance -> type of the instance relationship for the covergroup instance.
	flags	Flags for the scope.
Return value	Returns a scope handle, or NULL if error.	

8.5.10 ucis_CreateInstanceByName

Purpose	Creates an instance of the specified named design unit under the specified parent scope.	

Syntax	`ucisScopeT ucis_CreateInstanceByName(`
	`  ucisT  db,`
	`  ucisScopeT  parent,`
	`  const char*  name,`
	`  ucisSourceInfoT*  fileinfo,`
	`  int  weight,`
	`  ucisSourceT  source,`
	`  ucisScopeTypeT  type,`
	`  char*  du_name,`
	`  ucisFlagsT  flags);`	

Arguments	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	Source location appropriate to the UCISDB scope (encodes file, line, and item).
	weight	Weight to assign to the scope. Negative indicates no weight.
	source	Source of instance.
	type	Type of scope to create: UCIS_INSTANCE or UCIS_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.
Return value	Returns a scope handle, or NULL if error.	

8.5.11 ucis_CreateNextTransition

Purpose	This is a specialized version of ucis_CreateNextCover() to create a transition coveritem under an existing parental scope of type UCIS_FSM_TRANS. It records the source and destination state coveritem indices along with the transition. The parent of the state coveritems is assumed to be a sibling of the parent and to have type UCIS_FSM_STATES; the referenced state coveritems are assumed to already exist.	

Syntax	`int ucis_CreateNextTransition(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  ucisCoverDataT*  data,`
	`  ucisSourceInfoT*  *sourceinfo*,`
	`  int*  *transition_index_list*);`	

Arguments	*db*	Database.
	parent	Scope of type UCIS_FSM_TRANS scope.
	name	Coveritem name.
	data	Associated coverage data
	sourceinfo	Source information. This may be NULL.
	transition_index_list	Input array of int terminated with -1. The values are the coverindexes of the state coveritems participating in the transition list.
Return value	Returns the index number of the created coveritem on success, -1 on error.	

8.5.12 ucis_GetFSMTransitionStates

Purpose	Given a UCIS_FSM_TRANS coveritem, return the corresponding state coveritem indices and corresponding UCIS_FSM_STATES scope. This API removes the need to parse transition names in order to access the states.

Syntax	`ucisScopeT ucis_GetFSMTransitionStates(`
	`  ucisT  *db*,`
	`  ucisScopeT  *trans_scope*,  /* input handle for UCIS_FSM_TRANS scope */`
	`  int  *trans_index*,  /* input coverindex for transition */`
	`  int *  *transition_index_list*); /* output array of int, -1 termination */`

Arguments	*db* — Database. *trans_scope* — Input handle to identify the UCIS_FSM_TRANS scope *trans_index* — Input coverindex to identify the transition coveritem *transition_index_list* — Will point to an array of state coveritem indexes on success.
Return value	Returns the related scope of type UCIS_FSM_STATES on success, NULL on failure On success, the transition_index_list points to an integer array of UCIS_FSM_STATES coverindexes in the transition order. The array is terminated with -1 and the array memory is valid until a subsequent call to this routine or closure of the database, whichever occurs first. Callers should not attempt to free this list.

8.5.13 ucis_MatchScopeByUniqueID

Purpose	Find a scope in an in-memory database by its Unique ID. There will be at most one matching scope, no wildcards are supported. The search may be performed on the whole database (when scope is NULL) or under a parental scope, in which case a relative form Unique ID must be used. All name matching is case-sensitive.

Syntax	`ucisScopeT ucis_MatchScopeByUniqueID(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	` const char **uniqueID*)

Arguments	*db* — In-memory database. *scope* — Scope to search under. If NULL, the entire database is searched. *uniqueID* — Pointer to NULL-terminated Unique ID, which may be in full or relative form.
Return value	Returns a pointer to the matching scope, or NULL if there is no match or an error is encountered.

8.5.14 ucis_CaseAwareMatchScopeByUniqueID

Purpose	This routine performs the same function as ucis_MatchScopeByUniqueID except that the name matching is not consistently case-sensitive. Instead the case-sensitivity of the match for each name component is determined by the language setting of the component as traversal occurs. See Section 5.5 — "Using Unique IDs to perform database searches" on page 54 for more details on this algorithm. Because the database rules permit names differing only in case, even where the source language does not, it is possible (though ill-advised) for there to be multiple possible matches. The behavior in this case is undefined.	

Syntax	`ucisScopeT  ucis_CaseAwareMatchScopeByUniqueID(`
	`  ucisT  db,`
	`  ucisScopeT  scope,`
	`  const char  *uniqueID)`	

Arguments	*db*	In-memory database.
	scope	Scope to search under. If NULL, the entire database is searched.
	uniqueID	Pointer to NULL-terminated Unique ID, which may be in full or relative form.
Return value	Returns a pointer to the matching scope, or NULL if there is no match or an error is encountered.	

8.5.15 ucis_MatchCoverByUniqueID

Purpose	Find a coveritem in an in-memory database by its Unique ID. There will be at most one matching coveritem, no wildcards are supported. The search may be performed on the whole database (when scope is NULL) or under a parental scope, in which case a relative form Unique ID must be used. All name matching is case-sensitive.	

Syntax	`ucisScopeT ucis_MatchCoverByUniqueID(`
	`  ucisT  db,`
	`  ucisScopeT  scope,`
	`  const char  *uniqueID,`
	`  int  *index)`	

Arguments	*db*	In-memory database.
	scope	Scope to search under. If NULL, the entire database is searched.
	uniqueID	Pointer to NULL-terminated Unique ID, which may be in full or relative form.
	index	Returned coverindex (the combination of the return value, which is a scope, and this coverindex, identifies the matched coveritem).
Return value	Returns a pointer to the matching scope, or NULL if there is no match or an error is encountered. The index pointer will point to the associated coverindex on success.	

8.5.16 ucis_CaseAwareMatchCoverByUniqueID

Purpose	This routine performs the same function as ucis_MatchCoverByUniqueID except that the name matching is not consistently case-sensitive. Instead the case-sensitivity of the match for each name component is determined by the language setting of the component as traversal occurs. See Section 5.5 — "Using Unique IDs to perform database searches" on page 54 for more details on this algorithm. Because the database rules permit names differing only in case, even where the source language does not, it is possible (though ill-advised) for there to be multiple possible matches. The behavior in this case is undefined.

Syntax	`ucisScopeT ucis_MatchCoverByUniqueID(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  const char  **uniqueID*,`
	`  int  **index*)`

Arguments	*db* — In-memory database.
	scope — Scope to search under. If NULL, the entire database is searched.
	uniqueID — Pointer to NULL-terminated Unique ID, which may be in full or relative form.
	index — Returned coverindex (the combination of the return value, which is a scope, and this coverindex, identifies the matched coveritem).
Return value	Returns a pointer to the matching scope, or NULL if there is no match or an error is encountered. The index pointer will point to the associated coverindex on success.

8.5.17 ucis_MatchDU

Purpose	Returns the design unit scope with the specified name, or NULL if no match is found.

Syntax	`ucisScopeT ucis_MatchDU(`
	`  ucisT  *db*,`
	`  const char*  *name*);`

Arguments	*db* — Database.
	name — Design unit name to match.
Return value	Returns the design unit scope with the specified name, or NULL if no match is found.

8.5.18 ucis_GetIthCrossedCvp

Purpose	Gets the crossed coverpoint of the scope specified by the coverpoint index in the specified cross scope.	

Syntax	`int ucis_GetIthCrossedCvp(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  int  *index*,`
	`  ucisScopeT*  *point_scope*);`	

Arguments	*db*	Database.
	scope	Cross scope.
	index	Coverpoint index in the cross scope.
	point_scope	Crossed coverpoint scope returned.
Return value	Returns 0 if successful, or non-zero if error.	

8.5.19 ucis_SetScopeSourceInfo

Purpose	Sets source information for given scope.	

Syntax	`int`
	`ucis_SetScopeSourceInfo(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  ucisSourceInfoT*  *sourceinfo*);`	

Arguments	*db*	Database.
	scope	Scope.
	sourceinfo	Opaque handle that was previously created and is associated with a filename in the filename table.
Return value	Returns 0 on success, non-zero on failure.	

8.5.20 ucis_GetScopeFlag

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

Syntax	`int ucis_GetScopeFlag(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  ucisFlagsT  *mask*);`	

Arguments	*db*	Database.
	scope	Scope.
	mask	Flag bit to match with scope flags.
Return value	Returns 1 if the scope's flag bit matches the specified mask, otherwise, no match.	

8.5.21 ucis_SetScopeFlag

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

Syntax	`void ucis_SetScopeFlag(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  ucisFlagsT  *mask*,`
	`  int  *bitvalue*);`	

Arguments	*db*	Database.
	scope	Scope.
	mask	Flag bits to set.
	bitvalue	Value (0 or 1) to set mask bits.
Return value	None	

8.5.22 ucis_GetScopeFlags

Purpose	Returns the scope flags of the specified scope.	

Syntax	`ucisFlagsT ucis_GetScopeFlags(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*);`	

Arguments	*db*	Database.
	scope	Scope.
Return value	Returns the scope flags of the specified scope, or -1 if error.	

8.5.23 ucis_SetScopeFlags

Purpose	Sets the flags of the specified scope.	

Syntax	`void ucis_SetScopeFlags(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  ucisFlagsT  *flags*);`	

Arguments	*db*	Database.
	scope	Scope.
	flags	Flags to assign to scope.
Return value	None	

8.5.24 ucis_GetScopeSourceInfo

Purpose	Gets the source information (file/line/token) for the specified scope. Does not apply to toggle nodes.

Syntax	`int ucis_GetScopeSourceInfo(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  ucisSourceInfoT*  *sourceinfo*);`

Arguments	*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.
Return value	Returns 0 if successful, non-zero if error.

8.5.25 ucis_GetScopeType

Purpose	Returns the scope type of the specified scope, or UCIS_SCOPE_ERROR if error.

Syntax	`ucisScopeTypeT ucis_GetScopeType(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*)`

Arguments	*db* — Database.
	scope — Scope.
Return value	Returns the scope type of the specified scope, or UCIS_SCOPE_ERROR if error.

8.5.26 ucis_GetObjType

Purpose	Polymorphic function (aliased to ucis_GetHistoryKind) for acquiring an object type. Return value *must not be used* as a mask.

Syntax	`ucisObjTypeT ucis_GetObjType(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*);`

Arguments	*db* — Database.
	obj — Object.
Return value	Returns: — UCIS_HISTORYNODE_TEST (object is a test data record). — UCIS_HISTORYNODE_MERGE (object is a merge record), scope type ucisScopeTypeT (object is not of these). — UCIS_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.

8.6 Scope traversal functions

This section defines functions to traverse scopes in the UCISDB.

8.6.1 ucis_ScopeIterate

Purpose	Construct and initialize internal structures for iteration of scopes of given type(s) in UCISDB.	

Syntax	`ucisIteratorT ucis_ScopeIterate (`
	`  ucisT  db,`
	`  ucisScopeT  scope,`
	`  ucisScopeMaskTypeT scopemask);`	

Arguments	*db*	Database.
	scope	Scope whose associated (e.g. children) scopes are to be iterated in the database.. If the scope argument is NULL, apply iterator on the (global) database.
	scopemask	Scope type mask
Return value	Returns iterator handle, or NULL if error.	

8.6.2 ucis_ScopeScan

Purpose	Serially scan through the scopes of the type(s) selected by the iterator.	

Syntax	`ucisScopeT ucis_ScopeScan (`
	`  ucisT  *db*,`
	`  ucisIteratorT  *iterator*);`	

Arguments	*db*	Database.
	iterator	Iterator handle returned by ucis_ScopeIterate function.
Return value	Returns next scope handle, or NULL if done or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios. However, if the return value is NULL because all of the required scope nodes have been traversed, the error handler, if any, shall not be called.	

Note: In case of a NULL scope argument to ucis_ScopeIterate, ucis_ScopeScan shall iterate over all of the root scope nodes in the database filtered by the scopemask argument. In case of a non-NULL scope argument to ucis_ScopeIterate, ucis_ScopeScan shall iterate over all of its immediate children scope nodes filtered by the scopemask argument.

8.6.3 ucis_FreeIterator

Purpose	Free the iterator handle. `typedef void* ucisIteratorT;`

Syntax	`void ucis_FreeIterator (`
	`  ucisT  *db*,`
	`  ucisIteratorT  *iterator*);`

Arguments	*db* Database.
	iterator Iterator handle.
Return value	None.

Example	`ucisIteratorT iterator = ucis_ScopeIterate (db, NULL, UCIS_PACKAGE);`
	`while (pscope = ucis_ScopeScan (db, iterator)) {`
	`  …`
	`}`
	`ucis_FreeIterator (db, iterator);`

8.7 Coveritem traversal functions

This section defines functions to traverse coveritems in the UCISDB.

8.7.1 ucis_CoverIterate

Purpose	Construct and initialize internal structures for iteration of coveritems of given type(s) in UCISDB.	

Syntax	`ucisIteratorT ucis_CoverIterate (`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  ucisCoverMaskTypeT *covermask*);`	

Arguments	*db*	Database.
	scope	Scope whose associated (e.g. children) coveritems are to be iterated in the database. If the scope argument is NULL, apply iterator on the (global) database.
	covermask	Mask for type of coveritem.
Return value	Returns iterator handle, or NULL if error.	

8.7.2 ucis_CoverScan

Purpose	Serially scan through the coveritems of the type(s) selected by the iterator.	

Syntax	`int ucis_CoverScan (`
	`  ucisT  *db*,`
	`  ucisIteratorT  *iterator*);`	

Arguments	*db*	Database.
	iterator	Iterator handle returned by ucis_CoverIterate function.
Return value	Returns next coveritem, or -1 if done or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios. However, if the return value is -1 because all of the required coveritems have been traversed, the error handler, if any, shall not be called.	

Example	`ucisIteratorT iterator = ucis_CoverIterate (db, scope, UCIS_ALL_BINS);`
	`while ((bin = ucis_CoverScan (db, iterator)) != -1) {`
	`  …`
	`}`
	`ucis_FreeIterator (db, iterator);`	

8.8 History node traversal functions

This section defines functions to traverse history nodes in the UCISDB.

8.8.1 ucis_HistoryIterate

Purpose	Construct and initialize internal structures for iteration of history nodes of given type in UCISDB.	

Syntax	`ucisIteratorT ucis_HistoryIterate (`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *historynode*,`
	`  ucisHistoryNodeKindEnumT  *kind*);`	

Arguments	*db*	Database.
	historynode	History node whose associated (e.g. children) history nodes are to be iterated in the database. If the *historynode* argument is NULL, apply iterator on the (global) database.
	kind	History node kind.
Return value	Returns iterator handle, or NULL if error.	

8.8.2 ucis_HistoryScan

Purpose	Serially scan through the history nodes of the type selected by the iterator.	

Syntax	`ucisHistoryT ucis_HistoryScan (`
	`  ucisT  *db*,`
	`  ucisIteratorT  *iterator*);`	

Arguments	*db*	Database.
	iterator	Iterator handle returned by ucis_HistoryIterate function.
Return value	Returns next history node handle, or NULL if done or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios. However, if the return value is NULL because all of the required history nodes have been traversed, the error handler, if any, shall not be called.	

Note: In case of a NULL historynode argument to ucis_HistoryIterate, ucis_HistoryScan shall iterate over all of the root history nodes in the database filtered by the *kind* argument. In case of a non-NULL historynode argument to ucis_HistoryIterate, ucis_HistoryScan shall iterate over all of its immediate children history nodes filtered by the *kind* argument.

8.8.2.1 Example

```
ucisIteratorT iterator = ucis_HistoryIterate (db, NULL,
                                              UCIS_HISTORYNODE_MERGE);
while (test = ucis_HistoryScan (db, iterator)) {
      …
}
ucis_FreeIterator (db, iterator);
```

8.9 Tagged object traversal functions

This section defines functions to traverse tagged objects in the UCISDB.

8.9.1 ucis_TaggedObjIterate

Purpose	Construct and initialize internal structures for iteration of all objects in the UCISDB having the given tag.

Syntax	`ucisIteratorT ucis_TaggedObjIterate (`
	`  ucisT  *db*,`
	`  const char*  *tagname*);`

Arguments	*db* — Database.
	tagname — Tag whose associated objects are to be iterated in the database.
Return value	Returns iterator handle, or NULL if error.

8.9.2 ucis_TaggedObjScan

Purpose	Serially scan through the objects with the tag selected by the iterator.

Syntax	`ucisObjT ucis_TaggedObjScan (`
	`  ucisT  *db*,`
	`  ucisIteratorT  *iterator*,`
	`  int*  *coverindex_p*);`

Arguments	*db* — Database.
	iterator — Iterator handle returned by ucis_TaggedObjIterate function.
	coverindex_p — Returns either 0 or a positive number when the tagged object is a coveritem, otherwise coverindex_p returns -1.
Return value	Returns the next associated node handle, or NULL if done or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios. However, if the return value is NULL because all of the required history nodes have been traversed, the error handler, if any, shall not be called. If the tagged item is a coveritem, coverindex_p returns either 0 or a positive number when the tagged object is a coveritem, otherwise coverindex_p returns -1.

Example	`ucisIteratorT iterator = ucis_TaggedObjIterate (db, "mytag");`
	`while (scope = ucis_TaggedObjScan (db, iterator, coverindex_p)) {`
	`  …`
	`}`
	`ucis_FreeIterator (db, iterator);`

8.10 Tag traversal functions

This section defines functions to traverse tags in the UCISDB.

8.10.1 ucis_ObjectTagsIterate

Purpose	Initialize an iterator for all the tag names present on a specified object or UCISDB.	

Syntax	`ucisIteratorT ucis_ObjectTagsIterate (`
	`  ucisT  db,`
	`  ucisObjT  obj,`
	`  int  coverindex);`	

Arguments	*db*	Database.
	obj	Scope, coveritem, or history node object target for tag iteration. If the obj argument is NULL, apply iterator on the (global) database.
	coverindex	-1 for a scope or history node, else valid coveritem index.
Return value	Returns iterator handle. Returns NULL if the object has no tags, or for error.	

8.10.2 ucis_ObjectTagsScan

Purpose	Scan through the tag names present on a specified object using the previously initialized iterator. The order in which the tags are returned is undefined.	

Syntax	`const char* ucis_ObjectTagsScan (`
	`  ucisT  db,`
	`  ucisIteratorT  iterator );`	

Arguments	*db*	Database.
	iterator	Iterator handle returned by ucis_ObjectTagsIterate function.
Return value	Returns the tags owned by the object. A NULL return terminates the scan indicating there are no more tags. The callback based error handling mechanism in UCIS can continue for catching error scenarios. However, if the return value is NULL because all of the required tags have been traversed, the error handler, if any, shall not be called.	

8.11 Coveritem creation and manipulation functions

The coveritem typing system is based on a 64-bit one-hot type system. The coveritem type definitions are in the header file, as also are some related masking and multiple type selection definitions. These definitions are distinct for coveritems although the values overlap and therefore the typing alone cannot distinguish a scope from a coveritem.

The coveritem management routines share the ucisFlagsT data type definition with the scope routines, but coveritems use a separate set of flag meanings from scopes. Both the type and the coveritem flag definitions are in the header file.

The ucisCoverDataValueT and ucisCoverDataT data types are also defined in the header file. They are used for specifying the count precision, and for coveritem composite definition respectively. See Annex B, ’ Header file - normative reference” on page 317 for definitions of these items.

8.11.1 ucis_CreateNextCover

Purpose	Creates the next coveritem in the given scope.	

Syntax	`int ucis_CreateNextCover(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  ucisCoverDataT*  *data*,`
	`  ucisSourceInfoT*  *sourceinfo*);`	

Arguments	*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.
Return value	Returns the index number of the created coveritem, -1 if error.	

8.11.2 ucis_RemoveCover

Purpose	Has no effect when db is in streaming mode. Removes the specified coveritem from its parent. Coveritems cannot be removed from scopes of type UCIS_ASSERT (instead, remove the whole scope). Similarly, coveritems from scopes of type UCIS_TOGGLE with toggle kind UCIS_TOGGLE_SCALAR, UCIS_TOGGLE_-SCALAR_EXT, UCIS_TOGGLE_REG_SCALAR, or UCIS_TOGGLE_REG_-SCALAR_EXT cannot be removed (instead, remove the whole scope).

Syntax	`int ucis_RemoveCover(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  int  *coverindex*);`

Arguments	*db* Database.
	parent Parent scope of coveritem.
	coverindex Coverindex of coveritem to remove.
Return value	Returns 0 if successful, or -1 for error.

8.11.3 ucis_GetCoverData

Purpose	Gets name, data and source information for the specified coveritem. The user must save the returned data as the next call to this function can invalidate the returned data. **Note:** Any of the data arguments can be NULL (i.e., that data is not retrieved).

Syntax	`int ucis_GetCoverData(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  int  *coverindex*,`
	`  char**  *name*,`
	`  ucisCoverDataT*  *data*,`
	`  ucisSourceInfoT*  *sourceinfo*);`

Arguments	*db* Database.
	parent Parent scope of coveritem.
	coverindex Coverindex of coveritem in parent scope.
	name Name returned.
	data Data returned.
	sourceinfo Source information returned.
Return value	Returns 0 for success, and non-zero for error.

8.11.4 ucis_SetCoverData

Purpose	Sets data for the specified coveritem.	

Syntax	`int ucis_SetCoverData(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  int  *coverindex*,`
	`  ucisCoverDataT*  *data*);`	

Arguments	*db*	Database.
	parent	Parent scope of coveritem.
	coverindex	Coverindex of coveritem in parent scope.
	data	Data to set.
Return value	Returns 0 for success, or non-zero for error. The user must ensure the data fields are valid.	

8.11.5 ucis_IncrementCover

Purpose	Increment a coveritem data count by a specified amount	

Syntax	`int ucis_IncrementCover(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,  /* parent scope of coveritem */`
	`  int  *coverindex*, /* index of coveritem in parent */`
	`  int64_t  *increment*); /* added to current count */`	

Arguments	*db*	Database.
	parent	Parent scope of coveritem.
	coverindex	Coverindex of coveritem in parent scope.
	increment	Amount by which to increment the coveritem count.
Return value	Returns 0 for success, or -1 for failure.	

8.11.6 ucis_GetCoverFlag

Purpose	Get the Boolean value of the specified flag associated with the specified coveritem. Assumes that 'mask' is a one-hot value, otherwise the return value is undefined.	

Syntax	`int ucis_GetCoverFlag(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  int  *coverindex*,`
	`  ucisFlagsT  *mask*);`	

Arguments	*db*	Database.
	parent	Parent scope of coveritem.
	coverindex	Coverindex of coveritem in parent scope.
	mask	Flag mask to match.
Return value	Return 1 if the mask bit flag in the target coveritem is set, 0 otherwise.	

8.11.7 ucis_SetCoverFlag

Purpose	Sets bits in the coveritem's flag field with respect to the given mask.	

Syntax	`void ucis_SetCoverFlag(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  int  *coverindex*,`
	`  ucisFlagsT  *mask*,`
	`  int  *bitvalue*);`	

Arguments	*db*	Database.
	parent	Parent scope of coveritem.
	coverindex	Coverindex of coveritem in parent scope.
	mask	Flag mask.
	bitvalue	Value to set: 0 or 1.
Return value	None	

8.11.8 ucis_GetCoverFlags

Purpose	Get a coveritem flag field, i.e. all the flag settings.	

Syntax	`ucisFlagsT ucis_GetCoverFlags(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  int  *coverindex)*;`	

Arguments	*db*	Database.
	parent	Parent scope of coveritem.
	coverindex	Coverindex of coveritem in parent scope.
Return value	Returns a variable of type ucisFlagsT that contains all the coveritem flag settings.	

8.12 Coverage source file functions

Every UCIS object can potentially have a source file name stored with it. Different applications have different requirements for how these are stored. Consequently, the UCIS contains an object called a "file handle", which provides a way of storing indirect references to file names.

8.12.1 Simple use models

If you don’t care about file names and line numbers at all, you can create objects with NULL for the ucisSourceInfoT argument, e.g.:

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

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

```
ucisSourceInfoT sourceinfo;
sourceinfo file = ucisCreateFileHandle(db, filehandle, fileworkdir);
sourceinfo.line = myline;
sourceinfo.token = mytoken;

(void) ucis_CreateNextCover(db,parent,name,&coverdata,&sourceinfo);
```

This method creates a single global look-up table for file names within the UCISDB. File names are stored efficiently for each object within the UCISDB, and each unique file name string is stored only once. Whichever means are used to store file names, you can always access the file name, for example:

```
ucisSourceInfoT sourceinfo;
ucis_GetCoverData(db,parent,i,&name,&coverdata,&sourceinfo);
if (sourceinfo.filehandle != NULL) {
   printf("File name is %s\n",
      ucis_GetFileName(db,&sourceinfo.filehandle));
```

This interface uses the following types, the type definitions are in the header file. See Annex B, ’ Header file - normative reference” on page 317.

- ucisScopeT
- ucisObjT
- ucisFileHandleT
- ucisSourceInfoT

8.12.2 ucis_CreateFileHandle

Purpose	Creates a file handle for the specified file.

Syntax	`ucisFileHandleT ucis_CreateFileHandle(`
	`  ucisT  *db*,`
	`  const char*  *filename*,`
	`  const char*  *fileworkdir*);`

Arguments	*db* — Database.
	filename — Absolute or relative file name.
	fileworkdir — Work directory for the file when *filename* is a path relative to *fileworkdir*. Ignored if *filename* is an absolute path.
Return value	Returns the file handle if successful, or NULL if error.

8.12.3 ucis_GetFileName

Purpose	This function tries to reconstruct a valid file path from the file handle and the directory stored with it and the UCISDB.

Syntax	`const char* ucis_GetFileName(`
	`  ucisT  *db*,`
	`  ucisFileHandleT  *filehandle*);`

Arguments	*db* — Database.
	filehandle — File handle.
Return value	Returns the file name of the file specified by *filehandle*, or NULL if error.

In the following algorithm, *filename* and *fileworkdir* refer to the corresponding arguments of ucis_CreateFileHandle():

```
if (filename is an absolute path)
   return the path name
else if (filename exists at the relative path)
   return the path to filename
else if (filename exists relative to fileworkdir)
   return the path to filename in fileworkdir
else if (filename exists relative to the directory from which the
         UCISDB file was opened -- ie. the directory extracted from the file
         given to ucis_Open() or equivalent)
   return the path to filename relative to that directory
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 UCISDB was originally saved)
   return the path to filename relative to that directory
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 for lack of anything better.

8.13 History node functions

For efficiency, history nodes (ucisHistoryNodeT) 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 (ucisTestStatusT) are a subset of history nodes.

This interface uses the following types which are defined in the header file. See Annex B, ' Header file - normative reference" on page 317.

- ucisHistoryNodeT
- ucisHistoryNodeKindEnumT
- ucisTestStatusT

See also "History Nodes and Test Records" on page 26.

8.13.1 ucis_CreateHistoryNode

Purpose	Creates a history node of the specified kind in the specified database. History node has default values of FILENAME (path) and RUNCWD (current directory).	

Syntax	`ucisHistoryNodeT ucis_CreateHistoryNode(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *parent*,`
	`  char*  *logicalname*,`
	`  char*  *physicalname*,`
	`  ucisHistoryNodeKindT  *kind*);`	

Arguments	*db*	Database.
	parent	Parental history node.
	logicalname	Identifier of the UCISDB component - a string which may not be NULL and must be unique across all the UCISDB history nodes.
	physicalname	A path to the physical storage of the UCISDB, may be NULL.
	kind	A history node kind value from an enumerated set.
Return value	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.	

8.13.2 ucis_RemoveHistoryNode

Purpose	Removes the specified history node and all its descendents. Removing a node immediately invalidates the handle and invalidates the current traversal (if any).

Syntax	`int ucis_RemoveHistoryNode(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *historynode*);`

Arguments	*db* Database.
	child History node.
Return value	Returns 0 if successful, or -1 if error.

8.13.3 ucis_GetHistoryKind

Purpose	Polymorphic function (aliased to ucis_GetObjType) for acquiring an object type.

Syntax	`ucisHistoryNodeKindT ucis_GetHistoryKind(`
	`  ucisT  *db*,`
	`  ucisScopeT  *object*);`

Arguments	*db* Database.
	object Object.
Return value	Returns values from the ucisHistoryNodeKindT enumerated type.

8.14 Coverage test management functions

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.

8.14.1 Test status typedef

```
typedef enum {
    UCIS_TESTSTATUS_OK,
    UCIS_TESTSTATUS_WARNING,      // test warning ($warning called)
    UCIS_TESTSTATUS_ERROR,        // test error ($error called)
    UCIS_TESTSTATUS_FATAL         // fatal test error ($fatal called)
    UCIS_TESTSTATUS_MISSING,      // test not run yet
    UCIS_TESTSTATUS_MERGE_ERROR   /* testdata record was merged with
                                     inconsistent data values */
} ucisTestStatusT;
```

8.14.2 ucis_GetTestData

Purpose	Gets the data for the specified test in the specified database. Allocated values (strings, date and attributes) must be copied if the user wants them to persist.	

Syntax	`int ucis_GetTestData(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *testhistorynode*,`
	`  ucisTestDataT*  *testdata*);`	

Arguments	*db*	Database.
	testhistorynode	History node of type UCIS_HISTORYNODE_TEST.
	testdata	Pointer to results of query, in memory owned by the API.
Return value	Returns 0 if successful, or non-zero if error.	

8.14.3 ucis_SetTestData

Purpose	Sets the data for the specified test in the specified database.	

Syntax	`int ucis_SetTestData(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *testhistorynode*,`
	`  ucisTestDataT*  *testdata*);`	

Arguments	*db*	Database.
	testhistorynode	History node of type UCIS_HISTORYNODE_TEST.
	testdata	Pointer to data to be associated with test history node. Data will be copied to the database and the user copy can be freed or re-used after the call if appropriate.
Return value	Returns 0 if successful, or non-zero if error.	

8.15 Toggle functions

Toggles are the most common type of object in a typical code coverage database. Therefore, they have a specific interface in the API which 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).

Toggle scopes are associated with three properties from enumerated sets. The enumerated types to support this are ucisToggleMetricT, ucisToggleTypeT, and ucisToggleDirT. The type definitions are in the header file. See Annex B, ’ Header file - normative reference” on page 317.

8.15.1 ucis_CreateToggle

Purpose	Creates the specified toggle scope beneath the given parent scope.	

Syntax	`ucisScopeT ucis_CreateToggle(`
	`  ucisT  *db*,`
	`  ucisScopeT  *parent*,`
	`  const char*  *name*,`
	`  const char*  *canonical_name*,`
	`  ucisFlagsT  *flags*,`
	`  ucisToggleMetricT  *toggle_metric*,`
	`  ucisToggleTypeT  *toggle_type*,`
	`  ucisToggleDirT  *toggle_dir*);`	

Arguments	*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_metric	Toggle metric. The metric identifies the bin shape of the bins owned by the scope.
	toggle_type	Toggle type.
	toggle_dir	Toggle direction.
Return value	Returns a handle to created scope (type UCIS_TOGGLE).	

8.16 Tag functions

Tags may be placed on scopes, coveritems, or history nodes. The ucisObjMaskT one-hot enumerated type (defined in the header file, see Annex B, ’ Header file - normative reference” on page 317) may be used either to filter for objects from the required set, or to identify the set membership of an object using ucis_ObjKind().

8.16.1 ucis_ObjKind

Purpose	Returns object type, either ucisScopeT or ucisHistoryNodeT, for the specified object.	

Syntax	`ucisObjMaskT ucis_ObjKind(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*);`	

Arguments	*db*	Database.
	obj	Obj.
Return value	Returns the object type from the enumerated set defined by ucisObjMaskT.	

8.16.2 ucis_AddObjTag

Purpose	Adds a tag to a given object.	

Syntax	`int ucis_AddObjTag(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  const char*  *tag*);`	

Arguments	*db*	Database.
	obj	Object (ucisScopeT or ucisHistoryNodeT).
	coverindex	Coveritem index, -1 for scope or history node target object.
	tag	Tag.
Return value	Returns 0 with success, non-zero with error. Error includes null tag or tag with ’\n’ character.	

8.16.3 ucis_RemoveObjTag

Purpose	Removes the given tag from the object.	

Syntax	`int ucis_RemoveObjTag(`
	`  ucisT  *db*,`
	`  ucisObjT  *obj*,`
	`  int  *coverindex*,`
	`  const char*  *tag*);`	

Arguments	*db*	Database.
	obj	Object (ucisScopeT or ucisHistoryNodeT).
	coverindex	Coveritem index, -1 for scope or history node target object.
	tag	Tag.
Return value	Returns 0 with success, non-zero with error.	

8.17 Associating Tests and Coveritems

The primary reason for associating a test history node with a coveritem is to retain the information that the test incremented that particular coveritem, without necessarily retaining the incremental count. This potentially reduces the size of the associated datum from the number of bits in the count integer (for example, 32 or 64 bits) to 1 bit.

The information model for this association is a Boolean value associated with every combination of history node and coveritem. The test association is a many-to-many relation, i.e. each test history node may increment multiple coveritems, and each coveritem may be incremented by a number of test history nodes.

These routines generalize this many-to-many Boolean relation by adding a property to indicate the semantic meaning of the relation. For the test hit relation, this property should be set to the UCIS_ASSOC_TESTHIT definition.

The relation is modeled as a list of history nodes which may be associated with a coveritem. Presence on the list maps to the Boolean 'true' sense of the specified property. That is, a UCIS_ASSOC_TESTHIT list contains the list of history nodes that incremented the coveritem, omitting those that did not.

8.17.1 History Node Lists

A history node list is accessed through opaque handles of type ucisHistoryNodeListT, defined in the header file. History node lists are not meaningfully ordered and do not contain duplicates. An attempt to add a history node to a list that already contains it is not an error, but has no effect. An attempt to remove a history node from a list that does not contain the node is also not an error, but has no effect.

An empty list and a NULL history node list handle are both representations of zero history nodes associated with a coveritem. These are equivalent when presented to ucis_SetHistoryNodeListAssoc() in that both will delete any existing association. However ucis_GetHistoryNodeListAssoc() returns a NULL list pointer in both cases. That is, the database does not distinguish between an empty list and the absence of a list, and treats both cases as the absence of a list.

History node lists may be created, managed, associated with individual coveritems, queried from coveritems, and iterated for the history nodes on them. The iteration scan routine ucis_HistoryScan is defined in "History node traversal functions" on page 147.

History Node lists created by the user use internal resources and must be freed when no longer needed. Lists returned from the API implementation should not be freed; the resources will be reclaimed on the next call to the routine, or when the database is closed.

8.17.2 ucis_CreateHistoryNodeList

Purpose	This routine creates an empty list of history nodes prior to population.
Syntax	`ucisHistoryNodeListT ucis_CreateHistoryNodeList(` `  ucisT  db);`
Arguments	*db* Database.
Return value	A handle to a history node list that can be modified by the user.

8.17.3 ucis_FreeHistoryNodeList

Purpose	Free a history node list. User-constructed history node lists must be freed when no longer needed. This function does not free the history nodes on the list, only the list itself.

Syntax	`int ucis_FreeHistoryNodeList(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeListT  *historynode_list*);`

Arguments	*db* Database.
	historynode_list A history node list returned from ucis_CreateHistoryNodeList().
Return value	0 for success, -1 for failure.

8.17.4 ucis_AddToHistoryNodeList

Purpose	Add a history node to a history node list.

Syntax	`int ucis_AddToHistoryNodeList(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeListT  *historynode_list*,`
	`  ucisHistoryNodeT  *historynode*);`

Arguments	*db* Database.
	historynode_list A valid history node list previously obtained from ucis_CreateHistoryNodeList().
	historynode A valid history node.
Return value	0 for success, -1 for failure.

8.17.5 ucis_RemoveFromHistoryNodeList

Purpose	Remove a history node from a history node list.

Syntax	`int ucis_RemoveFromHistoryNodeList(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeListT  *historynode_list*,`
	`  ucisHistoryNodeT  *historynode*);`

Arguments	*db* Database.
	historynode_list A valid history node list previously obtained from ucis_CreateHistoryNodeList().
	historynode A valid history node.
Return value	0 for success, -1 for failure

8.17.6 ucis_HistoryNodeListIterate

Purpose	Create an iterator object from a history node list to allow the history nodes on the list to be iterated.

Syntax	`ucisIteratorT ucis_HistoryNodeListIterate(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeListT  *historynode_list*);`

Arguments	*db* Database.
	historynode_list A valid history node list.
Return value	A handle to an iterator object suitable for using with ucis_HistoryScan().

8.17.7 ucis_SetHistoryNodeListAssoc

Purpose	Associate a previously constructed history node list with a coveritem.	

Syntax	`int ucis_SetHistoryNodeListAssoc(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  int  *coverindex*,`
	`  ucisHistoryNodeListT  *historynode_list*,`
	`  ucisAssociationTypeT  *assoc_type*);`	

Arguments	*db*	Database.
	scope	Scope parent of coveritem.
	coverindex	Coverindex of coveritem, may not be -1.
	historynode_list	Previously constructed history node list.
	assoc_type	Defines the purpose of the association, for example: UCIS_ASSOC_TESTHIT
Return value	0 for success, -1 for failure.	

8.17.8 ucis_GetHistoryNodeListAssoc

Purpose	Return the history node list associated with the specified coveritem.	

Syntax	`ucisHistoryNodeListT ucis_GetHistoryNodeListAssoc(`
	`  ucisT  *db*,`
	`  ucisScopeT  *scope*,`
	`  int  *coverindex*,`
	`  ucisAssociationTypeT  *assoc_type*);`	

Arguments	*db*	Database.
	scope	Scope parent of coveritem.
	coverindex	Coverindex of coveritem, may not be -1.
	historynode_list	Previously constructed history node list.
	assoc_type	Defines the purpose of the association, for example: UCIS_ASSOC_TESTHIT
Return value	NULL for error conditions or if no association has been made, which is not considered an error. Otherwise a handle to the history node list is returned.	

8.18 Version functions

This section defines functions to obtain version information of UCIS API implementation tool, UCIS database in-memory, UCIS database in non-volatile form and UCIS history node.

8.18.1 ucis_GetAPIVersion

Purpose	Get the version handle of API implementation tool.
Syntax	`ucisVersionHandleT ucis_GetAPIVersion ();`
Arguments	None
Return value	A version handle, or NULL if error.

8.18.2 ucis_GetDBVersion

Purpose	Get the version handle of database in-memory.
Syntax	`ucisVersionHandleT ucis_GetDBVersion (` `  ucisT  db);`
Arguments	*db* Database.
Return value	A version handle, or NULL if error.

8.18.3 ucis_GetFileVersion

Purpose	Get the version handle of database in non-volatile form.
Syntax	`ucisVersionHandleT ucis_GetFileVersion (` `  char*  filename_or_directory_name);`
Arguments	*filename_or_directory_name* Absolute or relative path of database.
Return value	A version handle, or NULL if error.

8.18.4 ucis_GetHistoryNodeVersion

Purpose	Get the version handle of history node in the database.

Syntax	`ucisVersionHandleT ucis_GetHistoryNodeVersion (`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *historynode*);`

Arguments	*db* — Database.
	historynode — History Node whose version is to be obtained.
Return value	A version handle, or NULL if error.

8.18.5 ucis_GetVersionStringProperty

Purpose	Get string (const char*) properties from version handle.

Syntax	`const char* ucis_GetVersionStringProperty (`
	`  ucisVersionHandleT  *versionhandle*,`
	`  ucisStringPropertyEnumT  *property*);`

Arguments	*versionhandle* — Version handle.
	property — String property identification, taken from ucisStringPropertyEnumT definition.
Return value	Returns a NULL-terminated value of the string property, or a NULL if none or error. The callback based error handling mechanism in UCIS can continue for catching error scenarios like non-applicability of a specific property on version handle. However, if the return value is NULL because a string property's value was not implicitly or explicitly set, the error handler, if any, shall not be called.
Example	`const char* vendor_id = ucis_GetVersionStringProperty (versionhandle,` `  UCIS_STR_VER_VENDOR_ID);`

8.19 Formal data functions

8.19.1 Overview

UCISDB tests can be from simulation or from formal verification (UCISDB tests are described in the section, "Coverage test management functions" on page 159 with respect to simulation). A **formal test** is a ucisHistoryNodeT object, associated with formal information, that describes a formal verification run. The information associated with the formal test describes:

- how, when, where the formal test was run,
- the scope of formal analysis,
- the location of detailed results, and
- the description of the environment (assumptions used).

8.19.1.1 Formal verification results

Formal verification results include:

- **Assertion results** - proofs, failures, etc. This is described by formal status results for assertion scopes. There are APIs for setting and getting this information.
- **Coverage results** - cover statements, assertions exercised/witnessed, statement coverage, stimulus coverage etc. This is described primarily by the same scopes and coverage items as is done for simulation with some additional facilities for formal verification.

How formal coverage results, if written to the UCISDB, are to be interpreted is indicated by an attribute in the associated formal environment describing the context for interpreting the coverage data obtained from the formal verification run. **Coverage reachability** could be the primary objective of a formal verification run or it could be an ancillary byproduct of a formal verification run. The **coverage** could describe the controllability of the design based on the constraints used in the formal verification or it could indicate the parts of the design which are observed by assertions. The **coverage context** helps support various formal coverage use models. **Coverage type** means the different types of coverage information: statement coverage, FSM coverage, assertion coverage, covergroups, etc. **Formal coverage context** means how these different types of coverage were obtained or how they are interpreted. Also refer to "Introduction to UCIS" on page 8.

8.19.2 Formal Results API

These functions are used to set and get formal results for assertions.

This API consists of the following functions:

8.19.3 Formal Results enum

The following enum is used in the API functions to indicate a formal result for an assertion:

```
typedef enum {
     UCIS_FORMAL_NONE,           // No formal info (default)
     UCIS_FORMAL_FAILURE,        // Fails
     UCIS_FORMAL_PROOF,          // Proven to never fail
     UCIS_FORMAL_VACUOUS,        // Assertion is Vacuous as defined by the
                                 //   assertion language
     UCIS_FORMAL_INCONCLUSIVE,   // Proof failed to complete
     UCIS_FORMAL_ASSUMPTION,     // Assertion is an assume
     UCIS_FORMAL_CONFLICT        // Data merge conflict
} ucisFormalStatusT;
```

8.19.4 ucis_SetFormalStatus

Purpose	Sets the formal status on an assertion with respect to a test.	

Syntax	`int ucis_SetFormalStatus(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *assertscope*,`
	`  ucisFormalStatusT  *formal_status*);`	

Arguments	*db*	Database.
	test	A UCISDB test object.
	assertscope	Scope of assertion.

	formal_status	Assert formal status, where *status* may be one of:
		`UCIS_FORMAL_NONE`
		`UCIS_FORMAL_FAILURE`
		`UCIS_FORMAL_PROOF`
		`UCIS_FORMAL_VACUOUS`
		`UCIS_FORMAL_INCONCLUSIVE`
		`UCIS_FORMAL_ASSUMPTION`
		`UCIS_FORMAL_CONFLICT`

Return value	Returns 0 if successful, or non-zero if error. If there is an error, the Formal Status is not set.	

8.19.5 ucis_GetFormalStatus

Purpose	Get the formal status of an assertion. This function works with respect to a test.	

Syntax	`int ucis_GetFormalStatus(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *assertscope*,`
	`  ucisFormalStatusT*  *formal_status*);`	

Arguments	*db*	Database.
	test	A UCISDB test object.
	assertscope	Scope of assertion.
	formal_status	Returned formal status. Formal status is one of: UCIS_FORMAL_NONE UCIS_FORMAL_FAILURE UCIS_FORMAL_PROOF UCIS_FORMAL_VACUOUS UCIS_FORMAL_INCONCLUSIVE UCIS_FORMAL_ASSUMPTION UCIS_FORMAL_CONFLICT
Return value	Returns 0 if successful, or non-zero if error. On error, formal_status is not returned.	

8.19.6 ucis_SetFormalRadius

Purpose	Sets a formal radius with the assertion with respect to a test. The radius is: — a bounded proof result, or — a failure trace length.	

Syntax	`int ucis_SetFormalRadius(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *assertscope*,`
	`  int  *radius*,`
	`  char*  *clock_name*);`	

Arguments	*db*	Database.
	test	A UCISDB test object.
	assertscope	Scope. Set a radius to assertions with status: — UCIS_FORMAL_INCONCLUSIVE - a proof radius, if available. The formal radius for an inconclusive result with no formal proof radius information is -1. — UCIS_FORMAL_FAILURE - depth of counterexample
	radius	Set the radius for assertion with respect to a test.
	clock_name	The radius is measured in terms of this clock. The clock is specified as a hierarchical name string and can be NULL.
Return value	Returns 0 if successful, or non-zero if error (and the formal radius remains unchanged).	

8.19.7 ucis_GetFormalRadius

Purpose	Get formal radius and associated clock.	

Syntax	`int ucis_GetFormalRadius(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *assertscope*,`
	`  int*  *radius*,`
	`  char**  *clock*);`	

Arguments	*db*	Database.
	test	A UCISDB test object.
	assertscope	Assertion scope.
	radius	A returned formal radius, or -1 if not set or unknown.
	clock	A returned hierarchical clock name string, NULL if a NULL clock was set, or NULL if a Formal Radius was not set.
Return value	Returns 0 if successful. If error, radius and clock are not returned.	

8.19.8 ucis_SetFormalWitness

Purpose	The function sets witness waveform(s) for the assertion result. For a failed property, this is the counter example waveform. For a proven property, this can be a passing waveform. This function sets the file name of the waveform file or the directory name of a set of waveform files. The waveform file(s) are assumed to be in a standard or widely used industry format. Set the waveform file or directory of waveform files for an assertion with respect to a test:	

Syntax	`int ucis_SetFormalWitness(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *assertscope*,`
	`  char*  *witness_waveform_file_or_dir_name*);`	

Arguments	*db*	Database.
	test	UCISDB test.
	assertscope	Assertion scope.
	witness_waveform_file_or_dir_name	A string describing the path and filename of a waveform file, or a directory containing the waveform files.
Return value	Returns 0 if successful, non-zero on error. On error, witness data is not set.	

8.19.9 ucis_GetFormalWitness

Purpose	The function gets witness waveform(s) for the assertion result. For a failed property, this is the counter example waveform. For a proven property, this can be a passing waveform. This function gets the file name of the waveform file or the directory name of a set of waveform files. The waveform file(s) are assumed to be in a standard or widely used industry format. Get the waveform file or directory of waveform files for an assertion with respect to a test:	

Syntax	`int ucis_GetFormalWitness(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *assertscope*,`
	`  char**  *witness_waveform_file_or_dir_name*);`	

Arguments	*db*	Database.
	test	A UCISDB test object.
	assertscope	Assertion scope.
	witness_waveform_file_or_dir_name	A returned character string that indicates the path/filename or directory of the witness. Returns NULL if witness not set
Return value	Returns 0 if successful, non-zero on error. On error, *witness_waveform_file_or_dir_name* is not returned	

8.19.10 Formally Unreachable Coverage API

Coverage collected from simulation expresses what has been covered. Given a suitable stimulus, uncovered items may in theory be coverable, or they may be impossible to cover. Thus, in terms of formal verification, it is desirable to be able to indicate that coverage items are proven formally to be unreachable. Also, depending on the assumptions applied to formal verification, coverage items may be unreachable.

8.19.11 ucis_SetFormallyUnreachableCoverTest

Purpose	Set formally unreachable status for a particular coverage item (with respect to a particular test).	

Syntax	`int ucis_SetFormallyUnreachableCoverTest(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *coverscope*,`
	`  int  *coverindex*);`	

Arguments	*db*	Database.
	test	A UCISDB test.
	coverscope	Scope. *coverscope* and *coverindex* together indicate the coverage item.
	coverindex	*coverscope* and *coverindex* together indicate the coverage item.
Return value	Returns 0 if successful, or non-zero if error. Unreachable status is not set on error.	

8.19.12 ucis_GetFormallyUnreachableCoverTest

Purpose	This function is used to get formally unreachable status for a particular coverage item (with respect to a particular test):	

Syntax	`int ucis_GetFormallyUnreachableCoverTest(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisScopeT  *coverscope*,`
	`  int  *coverindex*,`
	`  int*  *unreachable_flag*);`	

Arguments	*db*	Database.
	test	A UCISDB test object (the particular test).
	coverscope	Scope. *coverscope* and *coverindex* together indicate the coverage item.
	coverindex	*coverscope* and *coverindex* together indicate the coverage item.
	unreachable_flag	Returns a flag indicating if this coverage item is unreachable: 0 = Not formally unreachable (default state). 1 = Formally unreachable.
Return value	Returns 0 if successful, non-zero on error. On error, unreachable status is not returned.	

8.19.13 Formal Environment APIs

These are the functions used to define the formal environment associated with a formal verification run (a formal test). The formal environment describes the scope of analysis and assumptions used in the formal verification run.

8.19.14 Formal Environment typedef

```
typedef void* ucisFormalEnvT;
```

8.19.15 ucis_AddFormalEnv

Purpose	This function creates a new formal environment object and returns a handle to it. A unique name for the formal environment and a scope pointer are required. The scope pointer specifies the part of the design where formal was run. Once a formal environment object is created, assumption scopes can be associated with the environment in order to describe the environmental constraints represented by this particular formal environment (see "ucis_AssocAssumptionFormalEnv" on page 179). Formal environments can then be associated with formal test runs to express what the environmental conditions were for that formal run (see "ucis_AssocFormalInfoTest" on page 177).	

Syntax	`ucisFormalEnvT ucis_AddFormalEnv(`
	`  ucisT  *db*,`
	`  const char*  *name*,`
	`  ucisScopeT  *scope*`
	
	`);`	

Arguments	*db*	Database.
	name	Unique name for this environment.
	scope	Scope of formal analysis - the scope of the design analyzed by formal verification.
Return value	Returns the formal environment or NULL on error.	

8.19.16 ucis_FormalEnvGetData

Purpose	Get the data from a formal environment.	

Syntax	`int ucis_FormalEnvGetData(`
	`  ucisT  *db*,`
	`  ucisFormalEnvT  *formal_environment*,`
	`  const char**  *name*,`
	`  ucisScopeT*  *scope*`
	`);`	

Arguments	*db*	Database.
	formal_environment	The UCISDB formal environment object.
	name	Returned name of the formal environment.
	scope	Returned scope of formal analysis. Scope of formal analysis is the scope of the design analyzed by formal verification.
Return value	Returns 0 if successful. On error, formal environment data is not returned.	

8.19.17 ucis_NextFormalEnv

Purpose	Iterate the collection of formal environments.	

Syntax	`ucisFormalEnvT ucis_NextFormalEnv(`
	`  ucisT  *db*,`
	`  ucisFormalEnvT  *formal_environment*`
	`);`	

Arguments	*db*	Database.
	formal_environment	To get the first environment, set *formal_environment* to NULL. To get the next environment, pass in the last returned environment.
Return value	Returns the next formal environment object. If NULL is passed in then it returns the first formal environment. Pass in last returned formal environment to get the next one. Returns NULL at the end of the list. i.e. there is no next formal environment.	

8.19.18 ucis_AssocFormalInfoTest

<table>
<tr><td>Purpose</td><td>To associate a formal environment, tool specific information, and the formal coverage context, with a test:
This makes the test a formal test. The argument formal_tool_info is a pointer to a structure containing tool specific information about that formal run:
— <code>formal_tool</code> The name of formal formal tool
— <code>formal_tool_version</code> The version of the formal tool
— <code>formal_tool_setup</code> A formal tool specific setup file (file name, named file assumed to be a text file)
— <code>formal_tool_db</code> The formal tool specific database (file name, file named is assumed to be a binary file)
— <code>formal_tool_rpt</code> The formal tool report file (file name, named file assumed to be text file)
— <code>formal_tool_log</code> The formal tool log file (file name, named file assumed to be text file)
<pre>typedef struct {
    char* formal_tool;
    char* formal_tool_version;
    char* formal_tool_setup;
    char* formal_tool_db;
    char* formal_tool_rpt;
    char* formal_tool_log;
} ucisFormalToolInfoS ucisFormalToolInfoT;</pre></td></tr>
<tr><td>Syntax</td><td><pre>int ucis_AssocFormalInfoTest(
    ucisT   db,
    ucisHistoryNodeT   test,
    ucisFormalToolInfoT* formal_tool_info,
    ucisFormalEnvT   formal_environment,
    char *   formal_coverage_context);</pre></td></tr>
<tr><td rowspan="5">Arguments</td><td>db</td><td>Database.</td></tr>
<tr><td>test</td><td>A UCISDB test object.</td></tr>
<tr><td>formal_tool_info</td><td>Tool specific information as described above.</td></tr>
<tr><td>formal_environment</td><td>A formal environment object.</td></tr>
<tr><td>formal_coverage_context</td><td>Describes how the formal coverage information associated with this test/environment was obtained or is to be interpreted (see the following section, “Formal Coverage Context” on page 178).</td></tr>
<tr><td>Return value</td><td colspan="2">Returns 0 if successful. On error the formal information is not associated with the test.</td></tr>
</table>

8.19.19 Formal Coverage Context

Coverage information from a formal verification tool could have different meanings and could be applied in different use models. The formal coverage context string indicates the context for interpreting the formal coverage information. This string is specified as one of the predefined UCISDB formal context attribute values described below, or it can be a user defined (application or tool specific) string, or it can be NULL meaning no formal coverage context specified.

```
#define UCIS_FORMAL_COVERAGE_CONTEXT_STIMULUS \
 "UCIS_FORMAL_COVERAGE_CONTEXT_STIMULUS"
#define UCIS_FORMAL_COVERAGE_CONTEXT_RESPONSE \
 "UCIS_FORMAL_COVERAGE_CONTEXT_REPONSE"
#define UCIS_FORMAL_COVERAGE_CONTEXT_TARGETED \
 "UCIS_FORMAL_COVERAGE_CONTEXT_TARGETED"
#define UCIS_FORMAL_COVERAGE_CONTEXT_ANCILLARY \
"UCIS_FORMAL_COVERAGE_CONTEXT_ANCILLARY"
#define UCIS_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS \
"UCIS_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS"
```

UCIS_FORMAL_COVERAGE_CONTEXT_STIMULUS indicates that the coverage information associated with this test attempts to approximate the set of legal stimuli permissible within the constraints of the formal verification run. An application for this formal coverage context is checking the assumption (formal constraints) to see if they are over or under constraining with respect to the users expectations.

UCIS_FORMAL_COVERAGE_CONTEXT_RESPONSE indicates that the coverage is related to the structures under observation by the assertions. Here the coverage helps to identify what parts of the design are "checked". An application for this formal coverage context is checking the completeness of the assertions.

UCIS_FORMAL_COVERAGE_CONTEXT_TARGETED indicates that an objective of the formal verification run was coverage analysis. The purpose of the coverage analysis could be to identify the controllable elements of the design, or to evaluate the particular assumptions applied. The coverage results represent a comprehensive coverage analysis. An application for this formal coverage context is coverage analysis.

UCIS_FORMAL_COVERAGE_CONTEXT_ANCILLARY indicates that the coverage is a byproduct of the formal analysis and not necessarily the primary objective for the formal verification. For example the main objective of the formal verification run could be proving or disproving assertions. However, in the formal verification run, some coverage information results were obtained which could be helpful in understanding what was exercised. The results are informative but not a comprehensive coverage analysis. For example parts of the design which don't fan into assertions might simply be ignored. Here the coverage is a side effect of the formal analysis.

UCIS_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS indicates that the coverage information is intended to help understand the formal analysis of the assertion(s) that have partial results (UCIS_FORMAL_INCONCLUSIVE status).

The formal coverage context string attribute relates to coverage information which may be written into a UCISDB by a formal tool, the formal status information for the assertions is captured by the API function ucis_SetFormalStatus().

8.19.20 ucis_FormalTestGetInfo

Purpose	Get the formal information associated with a formal test.

Syntax	`int ucis_FormalTestGetInfo(`
	`  ucisT  *db*,`
	`  ucisHistoryNodeT  *test*,`
	`  ucisFormalToolInfoT**  *formal_tool_info*,`
	`  ucisFormalEnvT*  *formal_environment*,`
	`  char **  *formal_coverage_context*`
	`);`

Arguments	*db* — Database. *test* — A UCISDB test object. *formal_tool_info* — The returned tool-specific information. *formal_environment* — The returned associated formal environment. *formal_coverage_context* — Retuned formal coverage context that describes how the formal coverage information associated with this test/environment was obtained or is to be interpreted (see "Formal Coverage Context" on page 178).
Return value	Returns 0 if successful, non-zero on error. On error, formal test information is not returned.

8.19.21 ucis_AssocAssumptionFormalEnv

Purpose	Associate an assumption with this formal environment.

Syntax	`int ucis_AssocAssumptionFormalEnv(`
	`  ucisT  *db*,`
	`  ucisFormalEnvT  *formal_env*,`
	`  ucisScopeT  *assumption_scope*`
	`);`

Arguments	*db* — Database. *formal_env* — A formal environment. *assumption_scope* — The scope of an assumption.
Return value	Returns 0 if successful, non-zero on error. On error the assumption is not associated with the formal environment.

8.19.22 ucis_NextFormalEnvAssumption

Purpose	Iterate the assumptions associated with this formal environment:

Syntax	`ucisScopeT ucis_NextFormalEnvAssumption(`
	`  ucisT  *db*,`
	`  ucisFormalEnvT  *formal_env*,`
	`  ucisScopeT  *assumption_scope*`
	`);`

Arguments	*db* — Database.
	formal_env — A formal environment.
	assumption_scope — Assumption scope. Pass in NULL to get first assumption scope. Pass in last returned assumption scope to get the next one.
Return value	Returns the next assumption scope. To get the first assumption scope, pass in NULL. Pass in the last-returned assumption scope to get the next one. Returns NULL at the end of the list. i.e. there is no next formal environment assumption.

9 XML Interchange Format

This chapter contains the following sections:

9.1 Introduction

This chapter describes the W3C XML compliant schema to represent coverage data in XML according to the UCIS data model. The XML representation of coverage data shall serve as the interchange format and must conform to the schema.

9.2 XML schema approach

The XML schema for UCIS in this document is based on the following approach.

— An XML element (xsd:element) is used to model a mandatory UCIS item, and its presence is ensured by the schema. For example, `fileName` must be specified as shown here:

```
<xsd:element name="fileName" type=xsd:string/>
```

— An alternative way, used to specify a mandatory item, is modeling it as an XML attribute with the `use` field set to `required`, as shown here:

```
<xsd:attribute name="fileName" type=xsd:string use="required"/>
```

— An XML attribute (xsd:attribute) is most commonly used to model an optional UCIS item that may be omitted. Its predominant use is to model UCIS attributes. By default, XML schema attributes are optional and may be omitted in an XML specification. For example, `simtime` can be optionally specified with value of `type double`.

```
<xsd:attribute name="simtime" type=xsd:double/>
```

— A list of UCIS items is modeled as a list of XML elements. For example, the schema for a list of toggle bits is shown here:

```
<xsd:element name="toggleBit" type="TOGGLE_BIT" minOccurs="1"
             maxOccurs="unbounded"/>
```

Here, the list must contain a minimum of one element, but there is no limit imposed on the maximum number of elements in the list.

— A group of elements is modeled as an XML schema complex type (`xsd:complexType`). An XML schema complex type allows several useful features such as imposing an order on the contained elements and using that schema type in other contexts, like a structured macro, without the need to repeat the same elements contained in the group.

9.3 Definitions of XML Complex type used for modeling UCIS

This section describes XML complex types frequently used in modeling a UCIS data model. The definitions of these complex types is presented with a description of its constituent items.

9.3.1 NAME_VALUE

NAME_VALUE is a schema type that represents a pair of name of an item and its value, such as a design parameter name and its value.

Table 9-5 — NAME_VALUE

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name of an item.		string	No
value	Value of the item.		string	No

The XML schema of NAME_VALUE is shown below.

```
<xsd:complexType name="NAME_VALUE">
      <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
 </xsd:complexType>
```

9.3.2 SOURCE_FILE schema

The schema type SOURCE_FILE represents a design source file. Table 9-6 describes the SOURCE_FILE schema items. In other parts of the XML file, a source file is referenced by its id, which is a positive integer.

Table 9-6 — SOURCE_FILE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
fileName	Name of the source file.	string	string	No
Id	Identifier assigned and used by XML as a handle; it is not part of the UCIS data model.	integer	positiveInteger	No

A list of two files, one with file name "/root/user/lib/design/mod.v" and file id 23, and the other with file name "/root/user/test/test.v" and file id 11, is shown below:

```
<sourceFiles fileName="/root/user/lib/design/mod.v " id="23"/>
<sourceFiles fileName> /root/user/test/test.v " id="11"/>
```

The XML schema for SOURCE_FILE is shown below:

```
<xsd:complexType name="SOURCE_FILE">
   <xsd:attribute name="fileName" type="xsd:string" use="required"/>
   <xsd:attribute name="id" type="xsd:positiveInteger" use="required"/>
</xsd:complexType>
```

9.3.3 LINE_ID

The schema type LINE_ID represents a source line in a design source file. It consists of the elements shown in Table 9-7.

Table 9-7 — LINE_ID

Schema item Name	Description	UCIS type	Schema type	Optional
file	ID of the source file which contains the source line.	integer	positiveInteger	No
Id	Line number of the source line in the source file.	integer	positiveInteger	No

The XML schema for LINE_ID is shown below.

```
<xsd:complexType name="LINE_ID">
      <xsd:attribute name="file" type="xsd:positiveInteger" use="required"/>
      <xsd:attribute name="line" type="xsd:positiveInteger" use="required"/>
</xsd:complexType>
```

9.3.4 STATEMENT_ID

The schema type LINE_ID represents a source line in a design source file. It consists of the elements shown in Table 9-8.

Table 9-8 — STATEMENT_ID

Schema item Name	Description	UCIS type	Schema type	Optional
file	ID of the source file which contains the source statement.	integer	positiveInteger	No
line	ID of the source line that contains the beginning of the statement.	integer	positiveInteger	No
inlineCount	Index of the statement within the line. An index is needed since a line may contain more than one statement. The index of the first statement is 1.	integer	positiveInteger	No

The XML schema for STATEMENT_ID is shown below.

```
<xsd:complexType name="STATEMENT_ID">
      <xsd:attribute name="file" type="xsd:positiveInteger" use="required"/>
      <xsd:attribute name="line" type="xsd:positiveInteger" use="required"/>
      <xsd:attribute name="inlineCount" type="xsd:positiveInteger" use="required"/>
</xsd:complexType>
```

9.3.5 DIMENSION schema

The schema type DIMENSION specifies a dimension of a design object. Table 9-9 describes the DIMENSION schema items.

Table 9-9 — DIMENSION schema items

Schema item Name	Description	UCIS type	Schema type	Optional
left	Value of the left index.	integer	integer	No
right	Value of the right index.	integer	integer	No
downto	Direction of decreasing index. It is true if the index value from the left to the right index is decreasing, and false otherwise.	integer	boolean	No

Dimensions of an object req[3:0][31:0] are shown below:

```
<dimension left=”3” right=”0” downto=” true”/>
<dimension left=”31” right=”0” downto=”true”/>
```

The XML schema for DIMENSION is shown below:

```
<xsd:complexType name="DIMENSION">
    <xsd:attribute name="left" type="xsd:integer" use=”required”/>
    <xsd: attribute name="right" type="xsd:integer" use=”required”/>
    <xsd: attribute t name="downto" type="xsd:boolean" use=”required”/>
</xsd:complexType>
```

9.3.6 Bin attributes

A common set of UCIS attributes are used in coverage bins. These UCIS attributes are model as the XML schema attribute group binAttributes. The items in this attribute group are:

- Alias, to specify a user-defined named
- Coverage goal
- Excluded status to specify if the coverage count should be ignored for calculating the parent objects coverage
- excludedReason to specify the reason for excluded status
- Weight to be considered in coverage computation

The binAttributes XML schema is shown below:

```
<xsd:attributeGroup name="binAttributes">
    <xsd:attribute name="alias" type="xsd:string"/>
    <xsd:attribute name="coverageCountGoal" type="xsd:nonNegativeInteger"/>
    <xsd:attribute name="excluded" type="xsd:boolean" default=”false” />
    <xsd:attribute name="excludedReason" type="xsd:string"/>
    <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default=”1” />
</xsd:attributeGroup>
```

9.3.7 BIN schema

Coverage bins are modeled using the XML schema type BIN. The bin contains the name and type for constructing the UID of the object according to the UCIS data model. The coverage bin contains the actual coverage data and UCIS attributes. Table 9-10 describes the BIN schema items.

Table 9-10 — BIN schema items

Schema item Name	Description	UCIS type	Schema type	Optional
contents	Stores coverage and tests that cover its bin.		BIN_CONTENTS	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
binAttributes	A group of attributes for bins. See the definition of binAttributes.		binAttributes	Yes

The following example shows BIN, here representing a block coverage:

```
<blockBin coverageCountGoal="1" >
   <contents nameComponent="block” typeComponent=":22:" coverageCount="4">
      <historyNodeId>2</historyNodeId>
      <historyNodeId>5</historyNodeId>
   </contents>
</blockBin>
```

The XML schema for BIN is shown below:

```
<xsd:complexType name="BIN">
   <xsd:sequence>
      <xsd:element name="contents" type="BIN_CONTENTS"/>
      <xsd:element name="userAttr" type="USER_ATTR" minOccurs="0"
                   maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="binAttributes"/>
</xsd:complexType>
```

9.3.8 BIN_CONTENTS schema

The coverage data stored for a bin is modeled by the XML complex type BIN_CONTENTS. Table 9-11 describes the BIN_CONTENTS schema items.

Table 9-11 — BIN_CONTENTS schema items

Schema item Name	Description	UCIS type	Schema type	Optional
coverageCount	Specifies how many times the item was covered.	integer	nonNegativeInteger	No
historyNodeId	A list of history node identifiers which covered this bin.		nonNegativeInteger	Yes
nameComponent	Name component for the UID.	string	string	Yes
typeComponent	Type component for the UID.	string	string	Yes

The XML schema for BIN_CONTENTS is shown below:

```
<xsd:complexType name="BIN_CONTENTS">
   <xsd:sequence>
      <xsd:element name="historyNodeId" type="xsd:nonNegativeInteger"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="coverageCount" type="xsd:nonNegativeInteger"
                  use=”required”/>
   <xsd:attribute name="nameComponent" type="xsd:string"/>
   <xsd:attribute name="typeComponent" type="xsd:string"/>
</xsd:complexType>
```

9.3.9 Object attributes

Some UCIS objects are often associated with a common set of attributes. These attributes are grouped by the XML schema attributeGroup objAttributes. The UCIS attributes contained in this group are itemized in Table 9-12.

Table 9-12 — Object attributes

Schema item Name	Description	UCIS type	Schema type	Optional
alias	An alias name assigned to the UCIS object by the user.	string	string	Yes
excluded	Specifies if the UCIS object is excluded from coverage counts. By default, UCIS objects are not excluded.	integer	boolean	Yes
excludedReason	Specifies the reason for excluding the UCIS object. It is provided only when attribute "excluded" is true.	string	string	Yes
weight	Relative weight of the object in computing coverage metric. By default, UCIS objects have a weight of 1.	integer	nonNegativeInteger	Yes

The XML schema of objAttributes is shown below.

```
<xsd:attributeGroup name="objAttributes">
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="excluded" type="xsd:boolean" default="false"/>
   <xsd:attribute name="excludedReason" type="xsd:string"/>
   <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
</xsd:attributeGroup>
```

9.3.10 Metric attributes

UCIS coverage metrics are often associated with a common set of attributes. These attributes are grouped by the XML schema attributeGroup metricAttributes. The UCIS attributes contained in this group are itemized in Table 9-13.

Table 9-13 — Metric attributes

Schema item Name	Description	UCIS type	Schema type	Optional
metricMode	Name of the mode in which a particular metric is being monitored for coverage.		string	Yes
weight	Weight applied for this mode in overall coverage metric calculation. By default, modes have a value of 1.		nonNegativeInteger	Yes

The XML schema of metricAttributes is shown below.

```
<xsd:attributeGroup name="metricAttributes">
   <xsd:attribute name="metricMode" type="xsd:string"/>
   <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
</xsd:attributeGroup>
```

9.3.11 METRIC_MODE

The XML schema type METRIC_MODE is used when several modes are used for a coverage metric, and each mode is associated with user defined attributes. Table 9-14 shows the items of METRIC_MODE.

Table 9-14 — METRIC_MODE

Schema item Name	Description	UCIS type	Schema type	Optional
metricMode	Name of the mode in which a particular metric is being monitored for coverage.		string	No
userAttr	A list of user defined attributes. These attributes are associated with the metric mode.		USER_ATTR	Yes

The XML schema of metricAttributes is shown below.

```
<xsd:complexType name="METRIC_MODE">
   <xsd:sequence>
      <xsd:element name="userAttr" type="USER_ATTR"
                           minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
 <xsd:attribute name="metricMode" type="xsd:string" use="required"/>
```

9.3.12 User defined attributes

A user can introduce his own UCIS attribute (optional item) using the schema type USER_ATTR. USER_ATTR allows users to add one or more user defined UCIS attributes at various scopes and objects. The user provides the name of the attribute (as key), the type of the attribute (as type), and a value of the attribute. UCIS allows restricted type of values for the attributes as follows: string, int, int64, float, double and bits. The length of bits can be specified optionally by USER_ATTR attribute “len”. These user defined attributes are typically required for storing important operational aspects of a tool that are not predefined by the UCIS standard.

The following example shows an XML example of user defined UCIS attributes. There are two attributes: one with the name optimization_level, type integer and value 3, and the other with the name “run_mask”, type “bits” and value “01101100”.

```
<userAttr key="optimization_level" type="int"> 3 </userAttr>
<userAttr key="run_mask" type="bits" len=”8”> 01101100 </userAttr>
```

The schema of USER_ATTR is shown below:

```
<xsd:complexType name="USER_ATTR" mixed="true">
   <xsd:attribute name="key" type="xsd:string" use="required"/>
   <!-- type restrictions for the asttribute: -->
   <xsd:attribute name="type" use="required">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:enumeration value="int"/>
            <xsd:enumeration value="float"/>
            <xsd:enumeration value="double"/>
            <!-- string value: -->
            <xsd:enumeration value="str"/>
            <!-- binary value: -->
            <xsd:enumeration value="bits"/>
            <xsd:enumeration value="int64"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:attribute>
   <!-- length of binary attribute (type=="bits"): -->
   <xsd:attribute name="len" type="xsd:integer"/>
</xsd:complexType>
```

9.4 UCIS top-level XML schema

The XML schema is structured in a hierarchical fashion to represent the data model of UCIS. The top level element of the schema is a schema type UCIS. Table 9-15 describes the UCIS top-level schema items.

Table 9-15 — UCIS top-level schema items

Schema item Name	Description	UCIS type	Schema type	Optional
ucisVersion	UCIS version name.	string	string	No
writtenBy	Name of the tool that created the XML source.	string	string	No
writtenTime	Date and time when the XML source was created.		dateTime	No
sourceFiles	List of design source files.	string	SOURCE_FILE	At least one source file is required
historyNodes	List of history nodes; If there is more than one test, the contents of the XML file represent a merged coverage.		HISTORY_NODE	At least one test is required
instanceCoverages	Coverage for each module instance. The coverage hierarchy consists of encapsulation of coverage data for all metrics in accordance with the data model of UCIS.		INSTANCE_COVERAGE	At least one instance is required.

The XML schema of UCIS is shown below:

```
<xsd:element name="UCIS">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="sourceFiles" type="SOURCE_FILE"
                  minOccurs="1" maxOccurs="unbounded"/>
         <xsd:element name="historyNodes" type="HISTORY_NODE" minOccurs="1"
   maxOccurs="unbounded"/>
         <xsd:element name="instanceCoverages" type="INSTANCE_COVERAGE"
                  minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="ucisVersion" type="xsd:string" use="required"/>
      <xsd:attribute name="writtenBy" type="xsd:string" use="required"/>
      <xsd:attribute name="writtenTime" type="xsd:dateTime" use="required"/>
   </xsd:complexType>
</xsd:element>
```

The complex XML types TEST and INSTANCE_COVERAGE are described in later sections.

9.5 HISTORY_NODE schema and description

A test is described by schema type HISTORY_NODE. Table 9-16 describes the HISTORY_NODE schema items.

Table 9-16 — HISTORY_NODE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
historyNodeId	Identifier assigned and used by XML as a handle; it is not part of the UCIS data model.		nonNegativeInteger	No
parentId	ID of parent history node.		nonNegativeInteger	No
logicalName	Name assigned to the historyNode.		string	No
physicalName	Name of the physical file for the history node.		string	No
testStatus	Run status indicating if the test passed or failed.		boolean	No
kind	Type of history node.		string	Yes
date	Date and time when the test was run.		dateTime	No
toolCategory	The tool type.		string	No
ucisVersion	UCIS version in which the coverage for the test was originally represented.		string	No
vendorId	A vendor identification.		string	No
vendorTool	The vendor tool name.		string	No
vendorToolVersion	The vendor tool version string.		string	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
simtime	Simulation time to complete the test.		double	Yes
timeunit	Time unit in which the simulation time was measured.		string	Yes
cpuTime	CPU time to complete the test.		double	Yes
userName	Name of the user who ran the test.		string	Yes
seed	Seed used to conduct the test.		string	Yes
cost	Cost associated with the test.		decimal	Yes
args	Command string used to start the test.		string	Yes
cmd	Specific test command used for the test.		string	Yes
runCwd	Directory name in which the test was run.		string	Yes
compulsory	Compulsory UCIS attribute to specify if the test is a must-run test.		string	Yes
sameTests	Identification of the test to which this test is equivalent.		nonNegativeInteger	Yes
comment	A user comment.		string	Yes

An XML example of a test is shown below:

```
<historyNode
   historyNodeId=”11”
   logicalName =“test12”
   testStatus=“true”
   date =“12/2/2010 4:35PM”
   toolType=“simulation”
   ucisVersion =“1.0”
   vendorId =“vendorX”
   vendorTool=”XYZ_sim”
   vendorToolVersion=”7.0”
   simtime="25000"
      timeunit="ns"
      cpuTime="4500"
      userName="Joe Smith"
      physicalName="test12.pl"
      seed="45.34"
      kind="UCIS_HISTORYNODE_TEST"
      cost="3.2"
      args="sim –t test12.pl –f files"
      cmd="Add"
      runCwd="cpuTest"
      compulsory="Unknown"
      sameTests="9"
      comment="This is a comment">
   <userAttr key="optimization_level" type="int"> 3 </userAttr>
   <userAttr key="run_mask" type="bits" len=”8”> 01101100 </userAttr>
</test>
```

The XML schema of HISTORY_NODE is shown below:

```
<xsd:complexType name="HISTORY_NODE">
   <xsd:sequence>
      <xsd:element name="userAttr" type="USER_ATTR" minOccurs="0"
   maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd: attribute name="historyNodeId" type="xsd:nonNegativeInteger"
   use=”required”/>
   <xsd:attribute name="logicalName" type="xsd:string" use=”required”/>
   <xsd:attribute name="kind" type="xsd:string" use="required"/>
   <xsd:attribute name="testStatus" type="xsd:boolean" use=”required”/>
   <xsd:attribute name="date" type="xsd:dateTime" use=”required”/>
   <xsd:attribute name="toolCategory" type="xsd:string" use=”required”/>
   <xsd:attribute name="ucisVersion" type="xsd:string" use=”required”/>
   <xsd:attribute name="vendorId" type="xsd:string" use=”required”/>
   <xsd:attribute name="vendorTool" type="xsd:string" use=”required”/>
   <xsd:attribute name="vendorToolVersion" type="xsd:string" use=”required”/>
   <xsd:attribute name="simtime" type="xsd:double"/>
   <xsd:attribute name="timeunit" type="xsd:string"/>
   <xsd:attribute name="cpuTime" type="xsd:double"/>
   <xsd:attribute name="userName" type="xsd:string"/>
   <xsd:attribute name="physicalName" type="xsd:string"/>
   <xsd:attribute name="seed" type="xsd:string"/>
   <xsd:attribute name="cost" type="xsd:decimal"/>
   <xsd:attribute name="args" type="xsd:string"/>
   <xsd:attribute name="cmd" type="xsd:string"/>
   <xsd:attribute name="runCwd" type="xsd:string"/>
   <xsd:attribute name="compulsory" type="xsd:string"/>
    <xsd:attribute name="sameTests" type="xsd:nonNegativeInteger"/>
   <xsd:attribute name="comment" type="xsd:string"/>
</xsd:complexType>
```

9.6 INSTANCE_COVERAGE schema

The instance coverage is represented by the XML schema type INSTANCE_COVERAGE. It contains the full hierarchical representation of each coverage metric for the coverage of a design module instance. Each instance is represented separately. The instances appear as a list, containing as many instances as there are in the design. Table 9-17 describes the INSTANCE_COVERAGE schema items.

Table 9-17 — INSTANCE_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name of the module instance.	string	string	No
key	UCIS key value.		string	No
designParameter	A list of design parameters used for the instance; a design parameter consists of the name of the parameter and its value for the instance.		NAME_VALUE	Yes
id	Source statement identification where the instance was declared.		STATEMENT_ID	No
toggleCoverage	Toggle coverage metric hierarchy; more than one is allowed.	string	TOGGLE_COVERAGE	Yes
blockCoverage	Block coverage metric hierarchy; more than one is allowed.		BLOCK_COVERAGE	Yes
conditionCoverage	Condition coverage metric hierarchy; more than one is allowed.		CONDITION_COVERAGE	Yes
branchCoverage	Branch coverage metric hierarchy; more than one is allowed.		BRANCH_COVERAGE	Yes
fsmCoverage	FSM coverage metric hierarchy; more than one is allowed.		FSM_COVERAGE	Yes
assertionCoverage	Assertion coverage metric hierarchy; more than one is allowed.		ASSERTION_COVERAGE	Yes
covergroupCoverage	Covergroup coverage metric hierarchy; more than one is allowed.		COVERGROUP_COVERAGE	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
instanceId	Instance identifier assigned and used by XML as a handle to an instance, and is not part of the UCIS data model.		integer	Yes
alias	An alias for the name of the instance.		string	Yes
moduleName	Name of the module of which this is an instance.		string	Yes
parentInstanceId	Identifier of the parent instance in the design hierarchy.		integer	Yes

An XML example of INSTANCE_COVERAGE is shown below. For brevity, schema types for coverage metrics are not expanded in the example. Note that the toggle and condition coverage appear twice, while assertion coverage is omitted. A coverage metric can appear more than once to capture the data separately for each mode of the metric.

```
<instanceCoverage
   name=" eBusControlMode"
   key = "4"
   instanceId="49"
   alias="my_instance_1"
   moduleName="eBUS"
   parentInstanceId="5"
   >
      <designParameter>
         <name> eBusSize </name>
         <value> 32 </value>
      </designParameter>
      <id>
         <file>23</file>
         <line>12</line>
         <inlineCount>1</inlineCount>
      </id>
   <toggleCoverage> TOGGLE_COVERAGE </toggleCoverage>
   <toggleCoverage> TOGGLE_COVERAGE </toggleCoverage>
   <blockCoverage> BLOCK_COVERAGE </blockCoverage>
   <conditionCoverage> CONDITION_COVERAGE </conditionCoverage>
   <conditionCoverage> CONDITION_COVERAGE </conditionCoverage>
   <branchCoverage> BRANCH_COVERAGE </branchCoverage>
   <fsmCoverage> FSM_COVERAGE </fsmCoverage>
   <covergroupCoverage> COVERGROUP_COVERAGE </covergroupCoverage>
</instanceCoverage>
```

The XML schema of INSTANCE_COVERAGE is shown below:

```
<xsd:complexType name="INSTANCE_COVERAGE">
   <xsd:sequence>
      <xsd:element name="designParameter" type="NAME_VALUE"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="id" type="STATEMENT_ID"/>
      <xsd:element name="toggleCoverage" type="TOGGLE_COVERAGE" minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="blockCoverage" type="BLOCK_COVERAGE" minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="conditionCoverage" type="CONDITION_COVERAGE"
                   minOccurs="0"  maxOccurs="unbounded"/>
      <xsd:element name="branchCoverage" type="BRANCH_COVERAGE"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="fsmCoverage" type="FSM_COVERAGE"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="assertionCoverage" type="ASSERTION_COVERAGE"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="covergroupCoverage" type="COVERGROUP_COVERAGE"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR" minOccurs="0"
                   maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use=”required”/>
   <xsd:attribute name="key" type="xsd:string" use=”required”/>
   <xsd:attribute name="instanceId" type="xsd:integer"/>
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="moduleName" type="xsd:string"/>
   <xsd: attribute name="parentInstanceId" type="xsd:integer"/>
</xsd:complexType>
```

9.7 TOGGLE_COVERAGE schema

The schema complex type TOGGLE_COVERAGE encapsulates the toggle metric. Table 9-18 describes the TOGGLE_COVERAGE schema items. The TOGGLE_OBJECT schema is described in the next section.

Table 9-18 — TOGGLE_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
toggleObject	A list of design objects for which the toggle coverage data is present. An object in this representation contains the coverage data.		TOGGLE_OBJECT	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
metricMode	A list of metric modes that store user defined attributes specific for each mode.		METRIC_MODE	Yes
weight	Weight to be considered in calculating coverage contribution of toggle coverage with a default of 1.	integer	nonNegativeInteger	Yes

Note: There may be multiple instances of toggle coverage, where each instance stores information about a specific toggle coverage mode. For example, one test may monitor coverage in mode 2STOGGLE, while another test may monitor coverage in mode 3STOGGLE.

Two instances of toggle coverage representation in XML are shown below. For brevity, the toggle objects are not expanded. The difference between the two instances of coverage is the toggle coverage mode under which the coverage was monitored.

```
<toggleCoverage weight="2">
    <metricMode metricMode="2STOGGLE ">
    </metricMode>
    <toggleObject> TOGGLE_OBJECT </toggleObject>
    <userAttr key="designLevel" type="string">rtl1</userAttr>
</toggleCoverage>

<toggleCoverage weight="2">
    <metricMode metricMode="3STOGGLE ">
    </metricMode>
    <toggleObject> TOGGLE_OBJECT </toggleObject>
    <userAttr key="designLevel" type="string">gate</userAttr>
</toggleCoverage>
```

The XML schema for TOGGLE_COVERAGE is shown below:

```
<xsd:complexType name="TOGGLE_COVERAGE">
    <xsd:sequence>
        <xsd:element name="toggleObject" type="TOGGLE_OBJECT" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR" minOccurs="0"
                     maxOccurs="unbounded"/>
        <xsd:element name="metricMode" type="METRIC_MODE"
                     minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="weight" type="xsd: nonNegativeInteger " default=”1” />
</xsd:complexType>
```

9.7.1 TOGGLE_OBJECT schema

A toggle object captures toggle coverage of a design object, such as ff1[3:0]. A toggle object is expressed as a schema complex type TOGGLE_OBJECT. Any toggle object may be a vector, or a multi-dimensional, in which case a list of schema type DIMENSION specifies the dimensions, and a list of total bits for all the dimensions is contained in the toggle object. Table 9-19 describes the TOGGLE_OBJECT schema items. The schema type TOGGLE_BIT, described in the next section, represents coverage for a bit of the object.

Table 9-19 — TOGGLE_OBJECT schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name of the design object as declared in the design with dimensions according to the syntax of the source language.	string	string	No
key	UCIS key value.		string	No
dimension	A list containing one or more dimensions expressed as a schema type DIMENSION.		DIMENSION	Yes
type	Specifies the declared type of the design object and is domain specific.	string	string	No
id	Source statement identification where the object was declared.	string	STATEMENT_ID	No
toggleBit	A list containing one or more TOGGLE_BIT, corresponding to each bit of the design object.		TOGGLE_BIT	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
objAttributes	A group of attributes for UCIS objects.		objAttributes	Yes
portDirection	Port direction if the design object is a port of the parent module instance.		string	Yes

A toggle object named ff1 with a single dimension [3:0] is shown below. The schema complex type TOGGLE_BIT is not expanded for clarity.

```
<toggleObject
    name=”ff1”
    Key = “0”
    type=”wire”
    alias="my_ff1"
    weight="3"
    excluded="true"
    excludedReason="debug variable"
    portDirection="input">
<dimension left=” 3” right=” 0” downto=” true”/>
    <id>
        <file>3</file>
        <line> 56</line>
        <inlineCount>1</inlineCount>
    </id>
    <toggleBit> TOGGLE_BIT </toggleBit>
    <userAttr key="object_kind" type="string">2-valued</userAttr>
</toggleObject>
```

The XML schema of TOGGLE_OBJECT is shown below:

```
<xsd:complexType name="TOGGLE_OBJECT">
   <xsd:sequence>
   <xsd:element name="dimension" type="DIMENSION" minOccurs="0"
            maxOccurs="unbounded"/>
   <xsd:element name="id" type="STATEMENT_ID"/>
   <xsd:element name="toggleBit"  type ="TOGGLE_BIT" minOccurs="1"
            maxOccurs="unbounded"/>
   <xsd:element name="userAttr" type="USER_ATTR" minOccurs="0"
   maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string"  use="required”/
   <xsd:attribute name="key"  type="xsd:string"  use="required”/
   <xsd:attribute name="type" type="xsd:string"/  use="required”>
   <xsd:attribute name="portDirection" type="xsd:string"/>
   <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.7.2 TOGGLE_BIT schema

The schema type TOGGLE_BIT captures coverage of a single bit of a design object. It specifies the transition to be covered for a bit.Table 9-20 describes the TOGGLE_BIT schema items.

Table 9-20 — TOGGLE_BIT schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name, including the dimensions, of the bit, such as ff1[1].	string	string	No
key	UCIS key value.		string	No
index	If the toggle object is multi-dimensional, then a list of indices corresponding to the dimensions of the design object.		nonNegativeInteger	No
toggle	A list of coverage bins to hold toggle coverage for the bit.		TOGGLE	No
objAttributes	A group of attributes for UCIS objects.		objAttributes	Yes

A toggle bit in XML is shown below. The schema type TOGGLE is not expanded.

```
<toggleBit
   name=”ff1[2]”
   Key =”0”
   alias="ff1_bit_2"
   weight="1"
   excluded="false"
   <index> 2 </index>
   <toggle> ----
   </toggle>
</toggleBit>
```

The TOGGLE_BIT XML schema is shown below:

```
<xsd:complexType name="TOGGLE_BIT">
   <xsd:sequence>
      <xsd:element name="index" type="xsd: nonNegativeInteger " minOccurs="0"
                   maxOccurs="unbounded"/>
      <xsd:element name="toggle" type="TOGGLE" minOccurs="1"
                   maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
   <xsd:element name="name" type="xsd:string" use=”required”/>
   <xsd:element name="key" type="xsd:string" use=”required”/ >
   <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.7.3 TOGGLE schema

The schema type TOGGLE represents the value transition for a bit. The value transition can be customized for the mode of the toggle coverage. Value-identification syntax may be any recognized form appropriate to the variable type such as integer, state variable, enumerated name, binary string form, and so on. The coverage of TOGGLE is stored in the schema type BIN. Table 9-21 describes the TOGGLE schema items.

Table 9-21 — TOGGLE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
bin	Stores coverage and tests that cover its bin.	string	BIN	No
from	Beginning value of the transition.		integer	No
to	Ending value of the transition.		integer	No

TOGGLE is shown below:

```
<toggle
   name=”uid_name” key=”9“ from=”0” “to=”1”
   alias="ff1_2_b1"
   coverageCountGoal="5"
   excluded="false"
   weight="1">
   <bin alias="string" coverageCountGoal="0" excluded="false"
               excludedReason="string" weight="1">
      <contents coverageCount="3">
               <historyNodeId> 4 </historyNodeId>
               <historyNodeId> 8 </historyNodeId>
      </contents>
   </bin>
</toggle>
```

The XML schema of TOGGLE is shown below:

```
<xsd:complexType name="TOGGLE">
   <xsd:sequence>
      <xsd:element name="bin" type="BIN"/>
   </xsd:sequence>
   <xsd:attribute name="from" type="xsd:string" use=”required”/>
   <xsd:attribute name="to" type="xsd:string" use=”required”/>
</xsd:complexType>
```

9.8 COVERGROUP_COVERAGE schema

The schema complex type COVERGROUP_COVERAGE encapsulates the covergroup metric. Table 9-22 describes the COVERGROUP_COVERAGE schema items.

Table 9-22 — COVERGROUP_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
cgInstance	A list of covergroup instances for which the covergroup coverage data is present. A cgInstance in this representation contains the coverage data. In case the coverage is meant for the covergroup as a whole, such as with the option per_instance as false, there will be only one instance representing the coverage for the covergroup.		CGINSTANCE	No
metricAttributes	Attributes associated with a coverage metric.		metricAttributes	Yes
userAttr	One or more user defined UCIS attributes.	ucisAttrTypeT	USER_ATTR	Yes
weight	Weight to be considered in calculating coverage contribution of covergroup coverage with a default of 1.	int	nonNegativeInteger	No

Please note that there could be multiple instances of covergroup coverage, where each instance stores information about covergroup coverage with specific covergroup options selected for a test.

An example of Covergroup coverage in XML is shown below:

```
<covergroupCoverage metricMode="string" weight="1">
     <cgInstance excluded="false">
         ----
     </cgInstance>
    <userAttr key="func1" type="str"> </userAttr>
 </covergroupCoverage>
```

The XML schema of COVERGROUP_COVERAGE is shown below:

```
<xsd:complexType name="COVERGROUP_COVERAGE">
   <xsd:sequence>
      <xsd:element name="cgInstance" type="CGINSTANCE"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
</xsd:complexType>
```

9.8.1 CGINSTANCE (covergroup instance) schema

The coverage for a covergroup instance is modeled as the XML schema type CGINSTANCE. Table 9-23 describes the CGINSTANCE schema items.

Table 9-23 — CGINSTANCE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name of the covergroup instance either explicitly declared or automatically assigned.	const char*	string	No
key	UCIS key value.		string	No
options	A list containing one or more options as declared in the source code.		CGINST_OPTIONS	No
cgId	Source statement identification where the instance was declared.	string	CG_ID	No
cgParms	A list of design parameters used for the instance; a design parameter consists of the name of the parameter and its value for the instance.	string	NAME_VALUE	No
coverpoint	A list containing zero or more COVERPOINT, corresponding to each coverpoint of the covergroup instance. The list may be empty if there is no coverpoint in the source.		COVERPOINT	No
cross	A list containing zero or more CROSS, corresponding to each cross of the covergroup instance. The list may be empty if there is no cross in the source.		CROSS	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
alias	An alias for the name of the covergroup instance.		string	Yes
excluded	A UCIS attribute to indicate if the covergroup instance should be excluded from calculating coverage.		boolean	No
excludedReason	A reason for exclusion if the covergroup instance is excluded.		string	Yes

An example of covergroup instance coverage is shown below:

```
<cgInstance name=”pfail_cover” key=”12” alias="my_pfail_cover " excluded="false" >
        <options weight="1" goal="100" at_least="1" detect_overlap="false"
          auto_bin_max="64" cross_num_print_missing="0" per_instance="false"
          merge_instances="false"/>
         <cgId cgName="string" moduleName="string">
            <cgSourceId file="1" line="1" inlineCount="1"/>
         </cgId>
        <cgParms>
            <name>string</name>
```

```
            <value>string</value>
          </cgParms>
          <coverpoint name="string" alias="string" exprString="string">
            ------
          </coverpoint>
          <cross name="string" alias="string">
            -------
          </cross>
          <name name=”top.m1.inst1”/>
</cgInstance>
```

The XML schema for CGINSTANCE is shown below.

```
<xsd:complexType name="CGINSTANCE">
     <xsd:sequence>
        <xsd:element name="options" type="CGINST_OPTIONS"/>
        <xsd:element name="cgId" type="CG_ID"/>
        <xsd:element name="cgParms" type="NAME_VALUE"
         minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="coverpoint" type="COVERPOINT"
        minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="cross" type="CROSS"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="userAttr" type="USER_ATTR"
       minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use="required"/>
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="excluded" type="xsd:boolean" default="false"/>
   <xsd:attribute name="excludedReason" type="xsd:string"/>
</xsd:complexType>
```

9.8.2 CG_ID schema

The covergroup instance identifier is modeled by the XML schema type CG_ID. Table 9-24 describes the CG_ID schema items.

Table 9-24 — CG_ID schema items

Schema item Name	Description	UCIS type	Schema type	Optional
cginstSourceId	Source statement where the covergroup instance was declared.		STATEMENT_ID	No
cgSourceId	Source statement where the covergroup was declared.		STATEMENT_ID	No
name	Name of the covergroup.	string	string	No
moduleName	Name of the module or package in which the covergroup was declared.	string	string	No

An example of CG_ID is shown below:

```
<cgId cgName="cacheMiss" moduleName="cacheMod">
   <cginstSourceId file="4" line="39" inlineCount="1"/>
   <cgSourceId file="5" line="30" inlineCount="1"/>
</cgId>
```

The XML schema for CG_ID is shown below:

```
<xsd:complexType name="CG_ID">
   <xsd:sequence>
      <xsd:element name="cginstSourceId" type="STATEMENT_ID"/>
      <xsd:element name="cgSourceId" type="STATEMENT_ID"/>
   </xsd:sequence>
   <xsd:attribute name="cgName" type="xsd:string" use="required" />
   <xsd:attribute name="moduleName" type="xsd:string" use="required"/>
</xsd:complexType>
```

9.8.3 CGINST_OPTIONS (covergroup instance options) schema

The covergroup instance options are declared as part of the covergroup declaration according to SystemVerilog. Table 9-25 describes the CGINST_OPTIONS schema items.

Table 9-25 — CGINST_OPTIONS schema items

Schema item Name	Description	Default	UCIS type	Schema type	Optional
weight	Specifies the weight used for the instance in computing overall coverage.	1		nonNegativeInteger	No
goal	Specifies the target goal used for the covergroup instance.	100		nonNegativeInteger	No
comment	A comment for the covergroup instance to be saved in the coverage database.	“”		string	No
at_least	Minimum number of hits for each bin. A bin with a hit count that is less than number is not considered covered.	1		nonNegativeInteger	No
detect_overlap	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.	false		boolean	No
auto_bin_max	Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.	64		nonNegativeInteger	No
cross_num_printing_missing	Number of missing (not covered) cross product bins that shall be saved to the coverage database and printed in the coverage report.	0		nonNegativeInteger	No
per_instance	Each instance contributes to the overall coverage information for the **covergroup** type. When true, coverage information for this covergroup instance shall be saved in the coverage database and included in the coverage report. When false, implementations are not required to save instance-specific information.	false		boolean	No
merge_instances	When true, cumulative (or type) coverage is computed by merging instances together as the union of coverage of all instances. When false, type coverage is computed as the weighted average of instances. This is defined in the source for the covergroup and applies to all its instances.	false		boolean	No

An example of CGINST_OPTIONS is shown below:

```
<options weight="1" goal="100" comment="no comment" at_least="1"
   direct_overlap="false"
               auto_bin_max="32" cross_num_print_missing="0" per_instance="true"
               merge_instances="false"
/>
```

The XML schema for CGINST_OPTIONS is shown below:

```
<xsd:complexType name="CGINST_OPTIONS">
      <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="goal" type="xsd:nonNegativeInteger" default="100"/>
      <xsd:attribute name="comment" type="xsd:string" default=""/>
      <xsd:attribute name="at_least" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="direct_overlap" type="xsd:boolean" default="false"/>
     <xsd:attribute name="auto_bin_max" type="xsd:nonNegativeInteger" default="64"/>
      <xsd:attribute name="cross_num_print_missing" type="xsd:nonNegativeInteger"
                           default="0"/>
      <xsd:attribute name="per_instance" type="xsd:boolean" default="false"/>
      <xsd:attribute name="merge_instances" type="xsd:boolean" default="false"/>
</xsd:complexType>
```

9.8.4 COVERPOINT schema

The schema type COVERPOINT captures coverage of a single coverpoint of a covergroup. Table 9-26 describes the COVERPOINT schema items.

Table 9-26 — COVERPOINT schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name of the coverpoint.	string	string	No
key	UCIS key value.		string	No
options	A list containing one or more options as declared in the source code.		COVERPOINT_OPTIONS	No
coverpointBin	A list of coverage bins to hold coverpoint coverage.		COVERPOINT_BIN	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
alias	An alias for the name for the bit.		string	Yes
exprString	Expression representing the coverpoint.		string	No

An XML example of COVERPOINT is shown below.

```
<coverpoint name="string" key=”14” alias="string" exprString="string">
   <options weight="1" goal="100" at_least="1" direct_overlap="false"
            auto_bin_max="64"/>
   <coverpointBin name="string" alias="string" type="string">
            ----------
   </coverpointBin>
</coverpoint>
```

The XML schema for COVERPOINT is shown below.

```
<xsd:complexType name="COVERPOINT">
   <xsd:sequence>
      <xsd:element name="options" type="COVERPOINT_OPTIONS"/>
      <xsd:element name="coverpointBin" type="COVERPOINT_BIN"
                   minOccurs="1" maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use="required"/>
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="exprString" type="xsd:string"/>
</xsd:complexType>
```

9.8.5 COVERPOINT_OPTIONS schema

The coverpoint options are declared as part of the covergroup declaration according to SystemVerilog. Table 9-27 describes the COVERPOINT_OPTIONS schema items.

Table 9-27 — COVERPOINT_OPTIONS schema items

Schema item Name	Description	Default	UCIS type	Schema type	Optional
Weight	Specifies the weight used for the coverpoint in computing overall covergroup coverage.	1		nonNegativeInteger	No
Goal	Specifies the target goal used for the coverpoint coverage.	100		nonNegativeInteger	No
Comment	A comment for the coverpoint to be saved in the coverage database.	“”		string	No
at_least	Minimum number of hits for each bin. A bin with a hit count that is less than number is not considered covered.	1		nonNegativeInteger	No
detect_overlap	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.	false		boolean	No
auto_bin_max	Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.	64		nonNegativeInteger	No

An XML example of COVERPOINT_OPTIONS is shown below.

```
<options weight="1" goal="100" comment="medium_priority" at_least="1"
          detect_overlap="false" auto_bin_max="64"/>
```

The XML schema for COVERPOINT_OPTIONS is shown below.

```
<xsd:complexType name="COVERPOINT_OPTIONS">
   <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
   <xsd:attribute name="goal" type="xsd:nonNegativeInteger" default="100"/>
   <xsd:attribute name="comment" type="xsd:string" default=""/>
   <xsd:attribute name="at_least" type="xsd:nonNegativeInteger" default="1"/>
   <xsd:attribute name="detect_overlap" type="xsd:boolean" default="false"/>
   <xsd:attribute name="auto_bin_max" type="xsd:nonNegativeInteger" default="64"/>
   </xsd:complexType>
```

9.8.6 COVERPOINT_BIN schema

The schema type COVERPOINT_BIN specifies a coverage bin for a coverpoint. A bin specifies either a single value or a sequence of values to cover. The coverage of the bin is stored in the XML schema type BIN_CONTENTS. Table 9-28 describes the COVERPOINT_BIN schema items.

Table 9-28 — COVERPOINT_BIN schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name, UOR for the bin.	string	string	No
key	UCIS key value.		string	No
type	Type of the bin: default, illegal or ignore.		string	No
range	A list of values compressed into ranges of values. Either the range or sequence is used for a coverpoint.		RANGE_VALUE	No
sequence	A list values defined as sequences.		SEQUENCE	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
alias	An alias for the name for the bin.		string	Yes

An XML example of COVERPOINT_BIN is shown below.

```
<coverpointBin name="UOR_name" key="0" alias="my_cv_bin" type="ignore">
   <range from="20" to="88">
      <contents coverageCount="0">
      </contents>
   </range>
   <userAttr key="string" type="str" > Not needed </userAttr>
</coverpointBin>
```

The XML schema for COVERPOINT_BIN is shown below:

```
<xsd:complexType name="COVERPOINT_BIN">
   <xsd:sequence>
   <xsd:choice>
      <xsd:element name="range" type="RANGE_VALUE"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="sequence" type="SEQUENCE"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:choice>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use=”required”/>
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="type" type="xsd:string" use="required"/>
</xsd:complexType>
```

9.8.7 RANGE_VALUE schema

The coverpoint bins may be collected and modeled as ranges of values by the XML schema type RANGE_VALUE. SystemVerilog allows a coverpoint bin to contain ranges of values. A range is specified from a low value to a high value. For specifying a single value, same value is used for low and high. Table 9-29 describes the RANGE_VALUE schema items.

Table 9-29 — RANGE_VALUE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
contents	Stores coverage and tests that cover its bin.		BIN_CONTENTS	No
from	Minimum value of the range.		integer	No
to	Maximum value of the range.		integer	No

The XML schema for RANGE_VALUE is shown below.

```
<xsd:complexType name="RANGE_VALUE">
    <xsd:sequence>
        <xsd:element name="contents" type="BIN_CONTENTS"/>
    </xsd:sequence>
    <xsd:attribute name="from" type="xsd:integer" use="required"/>
    <xsd:attribute name="to" type="xsd:integer" use="required"/>
</xsd:complexType> >
```

9.8.8 SEQUENCE schema

A coverpoint bin may be collected for a sequence of values and modeled as SEQUENCE in the XML schema as described in Table 9-30. SystemVerilog allows a bin to contain a sequence of values.

Table 9-30 — SEQUENCE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
contents	Stores coverage and tests that cover its bin.		BIN_CONTENTS	No
seqValue	A list of values representing the sequence.		integer	No

An XML example of SEQUENCE is shown below.

```
<coverpointBin name="UOR_bin_name" key=”0” type="default">
   <sequence>
      <seqValue> 3 </seqValue>
      <seqValue> 5 </seqValue>
      <contents coverageCount="12">
      </contents>
   </sequence>
</coverpointBin>
```

The XML schema for SEQUENCE is shown below.

```
<xsd:complexType name="SEQUENCE">
   <xsd:sequence>
      <xsd:element name="contents" type="BIN_CONTENTS"/>
      <xsd:element name="seqValue" type="xsd:integer"
                   minOccurs="1" maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

9.8.9 CROSS schema

The schema type CROSS captures coverage of a single cross within a covergroup. The schema type CROSS_BIN specifies a coverage bin for a cross. Table 9-31 describes the CROSS schema items.

Table 9-31 — CROSS schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name of the cross.	string	string	No
key	UCIS key value.		string	No
options	A list containing one or more options as declared in the source code.		CROSS_OPTIONS	No
crossExpr	A list of expressions declared for the cross.		string	Yes
crossBin	A list of coverage bins to hold coverpoint coverage.		CROSS_BIN	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
alias	An alias for the name for the bit.		string	Yes

An XML example of a cross is shown below:

```
<cross name="UOR_name" key=”15” alias="string">
   <options weight="1" goal="100" comment="string" at_least="1"
            cross_num_print_missing="0"/>
   <crossExpr>string</crossExpr>
   <crossBin name= "UOR_name" key="0" alias="string">
   -------
   </crossBin>
   <userAttr key="x1" type="double" />
</cross>
```

The XML schema for CROSS is shown below.

```
<xsd:complexType name="CROSS">
   <xsd:sequence>
      <xsd:element name="options" type="CROSS_OPTIONS"/>
      <xsd:element name="crossExpr" type="xsd:string"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="crossBin" type="CROSS_BIN"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use="required"/>
   <xsd:attribute name="alias" type="xsd:string"/>
</xsd:complexType>
```

9.8.10 CROSS_OPTIONS schema

The cross options are declared as part of the covergroup declaration according to SystemVerilog. Table 9-32 describes the CROSS_OPTIONS schema items.

Table 9-32 — SROSS_OPTIONS schema items

Schema item Name	Description	Default	UCIS type	Schema type	Optional
weight	Specifies the weight used for the cross in computing overall covergroup coverage.	1		nonNegativeInteger	No
goal	Specifies the target goal used for the cross coverage.	100		nonNegativeInteger	No
comment	A comment for the cross to be saved in the coverage database.	“”		string	No
at_least	Minimum number of hits for each bin. A bin with a hit count that is less than number is not considered covered.	1		nonNegativeInteger	No
cross_num_printing_missing	Number of missing (not covered) cross product bins that shall be saved to the coverage database and printed in the coverage report.	0		nonNegativeInteger	No

An XML example of CROSS_OPTIONS is shown below:

```
<options weight="1" goal="100" comment="medium_priority" at_least="1"
          direct_overlap="false" auto_bin_max="64"/>
```

The XML schema for CROSS_OPTIONS is shown below.

```
<xsd:complexType name="CROSS_OPTIONS">
   <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
   <xsd:attribute name="goal" type="xsd:nonNegativeInteger" default="100"/>
   <xsd:attribute name="comment" type="xsd:string" default=""/>
   <xsd:attribute name="at_least" type="xsd:nonNegativeInteger" default="1"/>
   <xsd:attribute name="cross_num_print_missing" type="xsd:nonNegativeInteger"
                  default="0"/>
</xsd:complexType>
```

9.8.11 CROSS_BIN schema

The schema type CROSS_BIN represents the bin to capture the coverage of a cross. A bin for a cross specifies a cross value between two or more expressions or coverpoints as declared in the source. The coverage of the bins is stored in the schema type BIN_CONTENTS. Table 9-33 describes the CROSS_BIN schema items.

Table 9-33 — CROSS_BIN schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name, UOR for the bin.	const char*	string	No
key	UCIS key value.	int	nonNegativeInteger	No
type	Type of the bin: default, illegal or ignore. The default is “default”.		string	No
index	A list of indices of the coverpoint bins that compose the cross bin.		integer	No
contents	Stores coverage and tests that cover its bin.		BIN_CONTENTS	No
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
alias	An alias for the name for the bin.		string	Yes

An XML example of CROSS_BIN is shown below:

```
<crossBin name="UOR_NAME" key="0" type="default" >
   <index>1</index>
   <index>5</index>
      <contents coverageCount="2">
      </contents>
</crossBin>
```

The XML schema for CROSS_BIN is shown below.

```
<xsd:complexType name="CROSS_BIN">
   <xsd:sequence>
      <xsd:element name="index" type="xsd:integer"
                   minOccurs="1" maxOccurs="unbounded"/>
      <xsd:element name="contents" type="BIN_CONTENTS"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use=”required”/>
   <xsd:attribute name="key" type="xsd:string" use=”required”/>
   <xsd:attribute name="type" type="xsd:string" default=”default”/>
   <xsd:attribute name="alias" type="xsd:string"/>
</xsd:complexType>
```

9.8.12 An Example of covergroup coverage in XML

Consider the following SystemVerilog example:

```
module top;
    bit [3:0] a;
    bit [3:0] b;
    covergroup cg;
        type_option.merge_instances = 0;
        option.at_least = 2;
        cvpt_a: coverpoint a {
                                bins a1 = { [0:7] [9:10]};
                                bins a2 = { [11:12], [13:15] };
                              }
        cvpt_b: coverpoint b { bins bb[] = { [0:1]  }; }
        xab: cross cvpt_a, cvpt_b;
    endgroup
    cg cv = new;
    initial begin
        #1; a = 2; b = 4; cv.sample();
        #10; a = 9; b = 11; cv.sample();
    end
endmodule
```

The XML for the above example is:

```
<covergroupCoverage>
  <cgInstance name="top.cv" key=”12” alias="my_name">
   <options at_least="2" merge_instances="false"/>
   <cgId cgName="cg" moduleName="top">
    <cginstSourceId file="2" line="4" inlineCount="1"/>
    <cgSourceId file="4" line="42" inlineCount="1"/>
   </cgId>
   <coverpoint name="cvpt_a" key=”14” exprString="a">
     <options at_least=”2”/>
       <coverpointBin name="a1" key=”0” alias="string"
                      type="string">
          <range from="0" to="7">
          <range from="9" to="10">
          <contents coverageCount="5">
          </contents>
       </coverpointBin>
       <coverpointBin name="a2" key=”0”>
          <range from="11" to="12">
          <range from="13" to="15">
          <contents coverageCount="2">
          </contents>
       </coverpointBin>
   <coverpoint name="cvpt_b" exprString="b">
     <options at_least=”2”/>
       <coverpointBin name="bb[0]" key=”0”>
          <range from="0" to="0">
          <contents coverageCount="1">
          </contents>
      </coverpointBin>
       <coverpointBin name="bb[1]" key=”0” >
          <range from="1" to="1">
          <contents coverageCount="0">
          </contents>
```

```
        </coverpointBin>
    </coverpoint>
    <cross name="xab" key=”15” alias="my_xab">
        <options weight="1" goal="100"
                 comment="a comment" at_least="1"
                 cross_num_print_missing="0"/>
        <crossExpr> cvpt_a </crossExpr>
        <crossExpr> cvpt_b </crossExpr>
        <crossBin name="proper_uor_name" key=”0”>
          <index> 0 </index>
          <index> 0 </index>
          <contents coverageCount="2">
          </contents>
        </crossBin>
        <crossBin name="proper_uor_name" key=”0”>
          <index> 0 </index>
          <index> 1 </index>
          <contents coverageCount="3">
          </contents>
        </crossBin>
        <crossBin name="proper_uor_name">
          <index> 1 </index>
          <index> 0 </index>
          <contents coverageCount="1">
          </contents>
        </crossBin>
        <crossBin name="proper_uor_name">
          <index> 1 </index>
          <index> 1 </index>
          <contents coverageCount="5">
          </contents>
        </crossBin>
    </cross>
  </cgInstance>
</covergroupCoverage>
```

9.9 CONDITION_COVERAGE schema

The schema complex type CONDITION_COVERAGE encapsulates both the condition and expression coverage metrics. Table 9-34 describes the CONDITION_COVERAGE schema items.

Table 9-34 — CONDITION_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
expr	A list of top level expressions for which the coverage is present.	const char*	EXPR	Yes
userAttr	One or more user defined UCIS attributes.	one of ucisAttrTypeT	USER_ATTR	Yes
metricAttributes	Attributes associated with a coverage metric.		metricAttributes	Yes

Please note that there could be multiple instances of condition coverage, where each instance of condition coverage stores information about a specific condition coverage mode. For example, one test may monitor coverage in mode UCIS:BITWISE_CONTROL, while another test may monitor coverage in mode UCIS_VECTOR.

The XML schema for condition coverage is shown below:

```
<xsd:complexType name="CONDITION_COVERAGE">
   <xsd:sequence>
      <xsd:element name="expr" type="EXPR"
                   minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
</xsd:complexType>
```

9.9.1 EXPR (condition and expression coverage) schema

Condition and expression coverage models coverage on expression objects when they are used in conditional and assignment contexts respectively. In XML schema an expression is modeled as a complex type EXPR. Table 9-35 describes the EXPR schema items.

Table 9-35 — EXPR schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	UCIS universal identifier for the expression.	const char*	string	No
key	UCIS key value.	int	nonNegativeInteger	No
exprString	String representing the expression.	const char*	string	
id	Source statement identification where the expression appears.	int elements are retrieved separately using ucis_GetIntProperty	STATEMENT_ID	No

Table 9-35 — EXPR schema items (Continued)

Schema item Name	Description	UCIS type	Schema type	Optional
index	Index of this expression as a sub expression of its hierarchical parent expression. For the highest level expression, the index is 0. Otherwise it is 1 or greater.	int	nonNegativeInteger	No
width	Bit-width of the expression.	int	nonNegativeInteger	No
subExpr	A list of sub-expression strings for which the coverage exists under this expression in the expression bins.	retrieved as CONDSTR string attribute	string	
exprBin	A list of bins to cover various combinations of the sub-expressions according to the metric mode.	correspond to coveritems in the data model	EXPR_BIN	No
hierarchicalExpr	A list of sub-expressions which are further broken down into a hierarchy and coverage is gathered for the hierarchical sub-expressions.	const char*	EXPR	No
userAttr	One or more user defined UCIS attributes.	one of ucisAttrTypeT	USER_ATTR	Yes
statementType	Type of the HDL statement which contains this expression. It must appear at the top level expression. For sub-expressions, it is optional.	const char* - e.g., "if", "switch"	string	No
objAttributes	Attributes associated with objects.		objAttributes	Yes

An expression representation in XML is shown below:

```
<expr uid="#cond#1#4#1#" key="3" exprString="(a&&b) || (c&&d)" index="1" width="1"
       weight="1" statementType="if">
    <id file="5" line="35" inlineCount="2"/>
    <subExpr>a</subExpr>
    <subExpr>b</subExpr>
    <exprBin coverageCountGoal="1"
    ------
    </exprBin>
    <exprBin coverageCountGoal="1"
    ------
    </exprBin>
    <hierarchicalExpr uid="#cond#1#" key="3" exprString="(a&&b)" index="1" width="1"
                    weight="1">
    <id file="5" line="35" inlineCount="2"/>
    <subExpr>a</subExpr>
    ----
    </hierarchicalExpr>
    <hierarchicalExpr uid="#cond#1#" key="3" exprString="(c&&d)" index="2" width="1" >
    <id file="5" line="35" inlineCount="2"/>
    -----
```

```
    </hierarchicalExpr>
</expr>
```

The XML schema for EXPR is shown below:

```
<xsd:complexType name="EXPR">
    <xsd:sequence>
        <xsd:element name="id" type="STATEMENT_ID"/>
        <xsd:element name="subExpr" type="xsd:string"
                     minOccurs="1" maxOccurs="unbounded"/>
        <xsd:element name="bin" type="BIN"
                     minOccurs="1" maxOccurs="unbounded"/>
        <xsd:element name="hierarchicalExpr" type="EXPR"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR"
                     minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="key" type=" xsd:nonNegativeInteger " use="required"/>
    <xsd:attribute name="exprString" type="xsd:string" use="required"/>
    <xsd:attribute name="index" type="xsd:nonNegativeInteger" use="required"/>
    <xsd:attribute name="width" type="xsd:nonNegativeInteger" use="required"/>
    <xsd:attribute name="statementType" type="xsd:string"/>
    <xsd:attributeGroup ref="objAttributes"/>
 </xsd:complexType>
```

9.9.2 An example of condition coverage in XML

Consider the following example in Verilog:

```
module top;                                                    // line 1
    bit a,b,c,d,e,f,z;                                         // line 2
    always @ (b or c or d or e or f) begin                     // line 3
        a=(b&((c&d) ? (e|f) : (e&f))); z=(b|(f?c:e));          // line 4
        if (a || (b&&c) || (d&&e))                             // line 5
        $display("covering conditions");                       // line 6
    end                                                        // line 7
    initial begin                                              // line 8
        #1; f = 1; c=1;                                        // line 9
        #1; d=1;                                               // line 10
        #1; b=1;                                               // line 11
        #1; e=1;                                               // line 12
        #1; f=0;                                               // line 13
    end                                                        // line 14
endmodule                                                      // line 15
```

9.9.2.1 XML for metric mode BITWISE_FLAT

In this section, we illustrate the XML representation for metric mode BITWISE_FLAT. The XML representation is shown for the expression `(b&((c&d) ? (e|f) : (e&f)));` on line 4 of the example above.

```
<conditionCoverage metricMode="UCIS:BITWISE_FLAT">
        <expr uid="#expr#1#4#1#" key="2" exprString="(b&((c&d)?(e|f):(e&f)))"
    index="0"
              width="1"
         statementType="asignment">
          <id file="1" line="4" inlineCount="1"/>
          <subExpr>b</subExpr>
```

```
<subExpr>c</subExpr>
<subExpr>d</subExpr>
<subExpr>e</subExpr>
<subExpr>f</subExpr>
<exprBin uid="0----" key=”7” coverageCountGoal="1">
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="3">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="-0---" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="2">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="--0--" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="6">
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin exprBin uid="---0-" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="7">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="----0" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <contents coverageCount="3">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="1----" key=”7”>
  <exprValue>1</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="5">
```

```
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="-1---" key=”7”>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <contents coverageCount="10">
            <historyNodeId>7</historyNodeId>
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="--1--" key=”7”>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <contents coverageCount="2">
            <historyNodeId>7</historyNodeId>
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="---1-" key=”7” coverageCountGoal="1">
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <exprValue>-</exprValue>
          <contents coverageCount="8">
            <historyNodeId>6</historyNodeId>
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="----1" key=”7”>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <contents coverageCount="9">
            <historyNodeId>7</historyNodeId>
            <historyNodeId>11</historyNodeId>
          </contents>
        </exprBin>
 </expr>
</conditionCoverage><conditionCoverage metricMode="UCIS:BITWISE_FLAT">
     <expr uid="#expr#1#4#1#" key="2" exprString="(b&((c&d)?(e|f):(e&f)))" index="0"
   width="1"
       statementType="asignment">
        <id file="1" line="4" inlineCount="1"/>
        <subExpr>b</subExpr>
        <subExpr>c</subExpr>
        <subExpr>d</subExpr>
        <subExpr>e</subExpr>
        <subExpr>f</subExpr>
        <exprBin uid="0----" key=”7” coverageCountGoal="1">
          <exprValue>0</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
```

```
  <contents coverageCount="3">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="-0---" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="2">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="--0--" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="6">
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin exprBin uid="---0-" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="7">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="----0" key=”7” coverageCountGoal="1">
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>0</exprValue>
  <contents coverageCount="3">
    <historyNodeId>2</historyNodeId>
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="1----" key=”7”>
  <exprValue>1</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="5">
    <historyNodeId>3</historyNodeId>
  </contents>
<exprBin uid="-1---" key=”7”>
  <exprValue>-</exprValue>
  <exprValue>1</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <exprValue>-</exprValue>
  <contents coverageCount="10">
    <historyNodeId>7</historyNodeId>
```

```
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="--1--" key=”7”>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <contents coverageCount="2">
            <historyNodeId>7</historyNodeId>
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="---1-" key=”7” coverageCountGoal="1">
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <exprValue>-</exprValue>
          <contents coverageCount="8">
            <historyNodeId>6</historyNodeId>
            <historyNodeId>3</historyNodeId>
          </contents>
        <exprBin uid="----1" key=”7”>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>-</exprValue>
          <exprValue>1</exprValue>
          <contents coverageCount="9">
            <historyNodeId>7</historyNodeId>
            <historyNodeId>11</historyNodeId>
          </contents>
        </exprBin>
 </expr>
</conditionCoverage>
```

9.9.2.2 XML for metric mode STD_HIER

In this section, we will illustrate the XML representation for metric mode STD_HIER. The XML representation is shown for the expression `(a || (b&&c) || (d&&e))` on line 5 of the example in section 9.9.2 on page 221.

```
<conditionCoverage metricMode="UCIS: STD_HIER ">
      <expr uid="#cond#1#5#1#" key="3" exprString="(a || (b&&c) || (d&&e))" index="0"
              width="1"
        statementType="if">
         <id file="1" line="4" inlineCount="1"/>
         <subExpr>a</subExpr>
         <subExpr>(b&&c)</subExpr>
         <subExpr>(d&&e)</subExpr>
         <exprBin uid="000" key=”8” coverageCountGoal="1">
            <exprValue>0</exprValue>
            <exprValue>0</exprValue>
            <exprValue>0</exprValue>
            <contents coverageCount="3">
              <historyNodeId>2</historyNodeId>
              <historyNodeId>3</historyNodeId>
            </contents>
         </exprBin>
         <exprBin uid="001" key=”8” coverageCountGoal="1">
           <exprValue>0</exprValue>
           <exprValue>0</exprValue>
           <exprValue>1</exprValue>
           <contents coverageCount="2">
             <historyNodeId>2</historyNodeId>
             <historyNodeId>3</historyNodeId>
           </contents>
         </exprBin>
         <exprBin uid="010" key=”8” coverageCountGoal="1">
           <exprValue>0</exprValue>
           <exprValue>1</exprValue>
           <exprValue>0</exprValue>
           <contents coverageCount="6">
             <historyNodeId>3</historyNodeId>
           </contents>
         </exprBin>
         <exprBin uid="100" key=”8” coverageCountGoal="1">
           <exprValue>1</exprValue>
           <exprValue>0</exprValue>
           <exprValue>0</exprValue>
           <contents coverageCount="7">
             <historyNodeId>2</historyNodeId>
             <historyNodeId>3</historyNodeId>
           </contents>
         </exprBin>
          <hierarchicalExpr uid="="#cond#1#" key="3" exprString="(b&&c)" index="1"
                              width="1" statementType="if">
                    <id file="1" line="4" inlineCount="1"/>
                    <subExpr>b</subExpr>
                    <subExpr>c</subExpr>
                    <exprBin uid="01" key=”8” coverageCountGoal="1">
                           <exprValue>0</exprValue>
                           <exprValue>1</exprValue>
                           <contents coverageCount="3">
                                  <historyNodeId>2</historyNodeId>
                                  <historyNodeId>3</historyNodeId>
                          </contents>
```

```
                    <exprBin uid="10" key=”8” coverageCountGoal="1">
                           <exprValue>1</exprValue>
                           <exprValue>0</exprValue>
                           <contents coverageCount="3">
                                  <historyNodeId>2</historyNodeId>
                                  <historyNodeId>3</historyNodeId>
                           </contents>
                    </exprBin>
                    <exprBin uid="11" key=”8” coverageCountGoal="1">
                           <exprValue>1</exprValue>
                           <exprValue>1</exprValue>
                           <contents coverageCount="3">
                                  <historyNodeId>2</historyNodeId>
                                  <historyNodeId>3</historyNodeId>
                           </contents>
                    </exprBin>
             </hierarchicalExpr>
               <hierarchicalExpr uid="#cond#2#" key="3"
                           exprString="(d&&e)" index="2" width="1" statementType="if">
                    <id file="1" line="4" inlineCount="1"/>
                    <subExpr>d</subExpr>
                    <subExpr>e</subExpr>
                    <exprBin uid="01" key=8 coverageCountGoal="1">
                           <exprValue>0</exprValue>
                           <exprValue>1</exprValue>
                           <contents coverageCount="3">
                                  <historyNodeId>2</historyNodeId>
                                  <historyNodeId>3</historyNodeId>
                           </contents>
                    <exprBin uid="10" key=8 coverageCountGoal="1">
                           <exprValue>1</exprValue>
                           <exprValue>0</exprValue>
                           <contents coverageCount="3">
                                  <historyNodeId>2</historyNodeId>
                                  <historyNodeId>3</historyNodeId>
                           </contents>
                    </exprBin>
                    <exprBin uid="11" key=8 coverageCountGoal="1">
                           <exprValue>1</exprValue>
                           <exprValue>1</exprValue>
                           <contents coverageCount="3">
                                  <historyNodeId>2</historyNodeId>
                                  <historyNodeId>3</historyNodeId>
                           </contents>
                    </exprBin>
             </hierarchicalExpr>
            </expr>
      </conditionCoverage>
```

9.10 ASSERTION_COVERAGE schema

The schema complex type ASSERTION_COVERAGE encapsulates coverage of assertions as declared in the source code. Assertions can be of different kinds, such as assert, cover, and assume. Table 9-36 describes the ASSERTION_COVERAGE schema items

Table 9-36 — ASSERTION_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
assertion	A list of assertions in the source code		ASSERTION	Yes
userAttr	One or more user defined UCIS attributes		USER_ATTR	Yes
metricAttributes	Attributes associated with a coverage metric.		metricAttributes	Yes

An instance of assertion coverage representation in XML is shown below:

```
<assertionCoverage metricMode="usb" weight="1">
    <assertion name="c1" assertionKind="cover">
           ------
    </assertion>
    <assertion name="c2" assertionKind="assert">
           ------
    </assertion>
           ---------
</assertionCoverage>
```

The XML schema for assertion coverage is shown below:

```
<xsd:complexType name="ASSERTION_COVERAGE">
   <xsd:sequence>
     <xsd:element name="assertion" type="ASSERTION"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
</xsd:complexType>
```

9.10.1 ASSERTION schema

An assertion is modeled with a XML schema type ASSERTION. It contains information about the assertion and a list of coverage bins associated with various aspects of assertion coverage. Table 9-37 describes the ASSERTION schema items

Table 9-37 — ASSERTION schema items

Schema item Name	Description	UCIS type	Schema type	Optional
nameComponent	Name as declared in the source code.	string	string	Yes
typeComponent	UCIS type component.	:17:	string	
assertionKind	Kind of assertions such as assert, assume and cover.		string	Yes
coverBin	Bin to store coverage of cover kind of assertion.		BIN	Yes
passBin	Bin to store coverage of the assertion when it passes.	Type component: :10: Name component: passbin	BIN	Yes
failBin	Bin to store coverage of the assertion when it fails.	Type component: :2: Name component: failbin	BIN	Yes
vacuousBin	Bin to store coverage of the assertion when it passes vacuously.	Type component: :15: Name component: vacuousbin	BIN	Yes
disabledBin	Bin to store coverage of the assertion when it was disabled.	Type component: :16: Name component: disabledbin	BIN	Yes
attemptBin	Bin to store coverage of the attempts made by the assertion.	Type component: :17: Name component: attemptbin	BIN	Yes
activeBin	Bin to store coverage of the active thread of the assertion.	Type component: :18: Name component: activebin	BIN	Yes
peakActiveBin	Bin to record the peak number thread active for the assertion at any time during simulation.	Type component: :22: Name component: peakactivebin	BIN	Yes
objAttributes	Attributes commonly associated with an object (alias, exclude, excludedReason and weight).		-	-
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of ASSERTION is shown below:

```
<assertion nameComponent="a1" typeComponent=”17:” assertionKind="assert">
        <passBin >
          <contents nameComponent="passbin" typeComponent=":10:" coverageCount="30">
            <historyNodeId> 6 </historyNodeId>
            <historyNodeId> 8 </historyNodeId>
          </contents>
        </passBin>
        <failBin >
```

```
          <contents nameComponent="failbin" typeComponent=":2:" coverageCount="2">
            <historyNodeId>2</historyNodeId>
            <historyNodeId>4</historyNodeId>
          </contents>
         </failBin>
         <vacuousBin >
        <contents nameComponent="vacuousbin" typeComponent=":15:" coverageCount="40">
            <historyNodeId>6</historyNodeId>
            <historyNodeId>8</historyNodeId>
          </contents>
         </vacuousBin>
         <disabledBin >
        <contents nameComponent="disabledbin" typeComponent=":16:" coverageCount="2">
            <historyNodeId>6</historyNodeId>
            <historyNodeId>8</historyNodeId>
          </contents>
         </disabledBin>
         <attemptBin >
         <contents nameComponent="atemptbin" typeComponent=":17:" coverageCount="74">
            <historyNodeId>2</historyNodeId>
            <historyNodeId>4</historyNodeId>
            <historyNodeId>6</historyNodeId>
            <historyNodeId>8</historyNodeId>
          </contents>
         </attemptBin>
</assertion>
```

The corresponding XML schema of ASSERTION is shown below:

```
<xsd:complexType name="ASSERTION">
      <xsd:sequence>
        <xsd:element name="coverBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="passBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="failBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="vacuousBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="disabledBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="attemptBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="activeBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="peakActiveBin" type="BIN"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="nameComponent" type="xsd:string"/>
      <xsd:attribute name="typeComponent" type="xsd:string"/>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="assertionKind" type="xsd:string" use="required"/>
      <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.11 FSM_COVERAGE schema

The schema complex type FSM_COVERAGE encapsulates coverage of Finite State Machine (FSM) components inferred from the source code. An inferred FSM is declared in the source code as an object such as a register and is associated with certain behavior as expressed in the code. The coverage for an FSM is modeled as a list of FSM states and a list of transitions between states. The coverage is recorded for states and transitions. Table 9-38 describes the FSM_COVERAGE schema items

Table 9-38 — FSM_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
fsm	A list of FSMs in the source code.		FSM	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
metricAttributes	Attributes associated with a coverage metric.		metricAttributes	Yes

An instance of FSM_COVERAGE representation in XML is shown below:

```
<fsmCoverage >
      <fsm >
        <name> ctrl </name>
        <type> reg </type>
        <width> 4 </width>
              ------
      </fsm>
      <fsm >
        <name> ctrl2 </name>
        <type> reg </type>
        <width> 4 </width>
              ------
      </fsm>
      ---------
</fsmCoverage>
```

The XML schema for FSM_COVERAGE is shown below.

```
<xsd:complexType name="FSM_COVERAGE">
   <xsd:sequence>
     <xsd:element name="fsm" type="FSM"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
</xsd:complexType>
```

9.11.1 FSM schema

An FSM is modeled with a XML schema type FSM. It contains information about the fsm and a list of states and transitions. Table 9-39 describes the FSM schema items

Table 9-39 — FSM schema items

Schema item Name	Description	UCIS type	Schema type	Optional
name	Name as declared in the source code.		string	Yes
type	Type of the source code object representing the fsm.		string	Yes
width	Width of the source code object representing the fsm.		positiveInteger	Yes
state	List of fsm states.		FSM_STATE	Yes
stateTransition	List of fsm state transitions.		FSM_TRANSITION	Yes
objAttributes	Attributes commonly associated with an object (alias, exclude, excludedReason and weight).		-	-
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

```
An XML example of FSM is shown below:

<fsm name=”ctrl” type=”reg” width=”4”>
        <state stateName="st1" stateValue="4">
            ------
        </state>
        <state stateName="st2" stateValue="6">
            ------
        </state>
        ------
        <stateTransition>
          <state>4</state>
          <state>6</state>
          ------
        </stateTransition>
        <stateTransition>
          <state>6</state>
          <state>4</state>
          ------
        </stateTransition>
        ------
</fsm>
```

The corresponding XML schema of FSM is shown below:

```
<xsd:complexType name="FSM">
      <xsd:sequence>
        <xsd:element name="state" type="FSM_STATE"
                          minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="stateTransition" type="FSM_TRANSITION"
                          minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
```

```
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="type" type="xsd:string"/>
      <xsd:attribute name="width" type="xsd:positiveInteger"/>
      <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.11.2 FSM_STATE schema

Coverage of an FSM state is modeled as a XML schema complex type FSM_STATE. It stores information about the state and the coverage for it. Table 9-40 describes the FSM_STATE schema items

Table 9-40 — FSM_STATE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
stateName	Name of the state.		string	No
stateValue	Value representing the state.		string	Yes
stateBin	A bin for the state to store coverage information.		BIN	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of FSM_STATE is shown below:

```
<state stateName="st1" stateValue="4">
          <stateBin >
            <contents nameComponent="4" typeComponent=":11:" coverageCount="3">
              <historyNodeId>1</historyNodeId>
              <historyNodeId>5</historyNodeId>
            </contents>
          </stateBin>
</state>
```

The XML schema for FSM_STATE is shown below:

```
<xsd:complexType name="FSM_STATE">
    <xsd:sequence>
      <xsd:element name="stateBin" type="BIN"/>
      <xsd:element name="userAttr" type="USER_ATTR"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="stateName" type="xsd:string"/>
    <xsd:attribute name="stateValue" type="xsd:integer"/>
</xsd:complexType>
```

9.11.3 FSM_TRANSITION schema

Coverage of an FSM transition is modeled as a XML schema complex type FSM_TRANSITION. It stores information about the state transition and the coverage for it. Table 9-41 describes the FSM_TRANSITION schema items

Table 9-41 — FSM_TRANSITION schema items

Schema item Name	Description	UCIS type	Schema type	Optional
state	A list of values representing the state transition.		string	No
transitionBin	A bin for the transition to store coverage information.		BIN	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of FSM_TRANSITION is shown below:

```
<stateTransition>
          <state>4</state>
          <state>6</state>
          <transitionBin>
            <contents nameComponent="4->6" typeComponent=":11:" coverageCount="4">
              <historyNodeId>3</historyNodeId>
            </contents>
          </transitionBin>
</stateTransition>
```

The XML schema for FSM_TRANSITION is shown below:

```
<xsd:complexType name="FSM_TRANSITION">
    <xsd:sequence>
      <xsd:element name="state" type="xsd:string"
      minOccurs="2" maxOccurs="unbounded"/>
      <xsd:element name="transitionBin" type="BIN"/>
      <xsd:element name="userAttr" type="USER_ATTR"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

9.12 BLOCK_COVERAGE (statement and block coverage) schema

The schema complex type BLOCK_COVERAGE encapsulates both the block and statement coverage metrics. The block coverage can be modeled as a flat list of statements, a list of processes which contain the blocks, or a list of blocks. In addition, a block may be hierarchical and may be nested to the depth as needed by the source code. Table 9-42 describes the BLOCK_COVERAGE schema items

Table 9-42 — BLOCK COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
process	A list of processes under which the block coverage is monitored. One and only one of items process, block and statement must be present.	UCIS_PROCESS	PROCESS_BLOCK	Yes
block	A list of blocks to be covered.	UCIS_BBLOCK	BLOCK	Yes
statement	A list of statements.		STATEMENT	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
metricAttributes	Attributes associated with a coverage metric.		metricAttributes	Yes

Please note that there could be multiple instances of block coverages, where each instance of block coverage stores information about a specific block coverage mode.

Three instances of block coverage representation in XML, one for the structure of each choice, are shown below:

```
<blockCoverage
          metricMode=" BLOCK_MODE2">
       <statement >
           <id file="5" line="25" inlineCount="2"/>
          <bin  ------  </bin>
       </statement>
        ----
</blockCoverage>

<blockCoverage
          metricMode="BLOCK_MODE1" >
       <block parentProcess="always">
         <statementId file="5" line="25" inlineCount="2"/>
              -----
         <blockId file="5" line="25" inlineCount="1"/>
          <bin  ------  </bin>
          <hierarchicalBlock ------  </ hierarchicalBlock>
       </block>
             ----
</blockCoverage>

<blockCoverage
          metricMode="BLOCK_MODE3" >
       <process processType="always">
          <block parentProcess="always">
             <statementId file="5" line="25" inlineCount="2"/>
              -----
             <blockId file="5" line="25" inlineCount="1"/>
             <bin  ------  </bin>
```

```
          </block>
               ----
       </process>
</blockCoverage>
```

The XML schema for block coverage is shown below:

```
<xsd:complexType name="BLOCK_COVERAGE">
    <xsd:sequence>
       <xsd:choice>
          <xsd:element name="process" type="PROCESS_BLOCK"
            minOccurs="1" maxOccurs="unbounded"/>
          <xsd:element name="block" type="BLOCK"
            minOccurs="1" maxOccurs="unbounded"/>
          <xsd:element name="statement" type="STATEMENT"
            minOccurs="1" maxOccurs="unbounded"/>
       </xsd:choice>
       <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
  </xsd:complexType>
```

9.12.1 STATEMENT schema

Coverage of a single statement is modeled as a XML schema complex type STATEMENT. It stores information about the statement and the coverage for it. Table 9-43 describes the STATEMENT schema items

Table 9-43 — STATEMENT schema items

Schema item Name	Description	UCIS type	Schema type	Optional
id	Statement source identifier.		STATEMENT_ID	No
bin	Bin to store coverage information.		BIN	No
objAttributes	Attributes commonly associated with an object (alias, exclude, excludedReason and weight).			
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of STATEMENT is shown below:

```
<statement>
   <id file="5" line="24" inlineCount="1"/>
   <bin coverageCountGoal="1" weight="1">
      <contents nameComponent="stmt" typeComponent=":5:" coverageCount="0">
         <historyNodeId>3</historyNodeId>
         <historyNodeId>5</historyNodeId>
      </contents>
      <userAttr key="label" type="st1" len="5441"/>
   </bin>
</statement>
```

The XML schema for STATEMENT is shown below:

```
<xsd:complexType name="STATEMENT">
   <xsd:sequence>
      <xsd:element name="id" type="STATEMENT_ID"/>
      <xsd:element name="bin" type="BIN"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.12.2 PROCESS_BLOCK schema

The schema type PROCESS_BLOCK represents a parent process of a block or statement as specified in the source code. This is an additional, optional layer of hierarchy over the coverage bins and is useful for categorizing blocks and statements. Table 9-44 describes the PROCESS_BLOCK schema items

Table 9-44 — PROCESS_BLOCK schema items

Schema item Name	Description	UCIS type	Schema type	Optional
block	A list of blocks to be covered for a process.		BLOCK	Yes
processType	Type of process such as always and initial.		string	No
objAttributes	Attributes commonly associated with an object (alias, exclude, excludedReason and weight).			
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of PROCESS_BLOCK is shown below:

```
<process processType="always">
        <block >
          <statementId file="5" line="24" inlineCount="2"/>
          <statementId file="5" line="25" inlineCount="1"/>          ----
          <blockBin >
            <contents nameComponent="block” typeComponent=":22:"  coverageCount="3">
              <historyNodeId>5</historyNodeId>
            </contents>
          </blockBin>
          <blockId file="5" line="24" inlineCount="1"/>
        </block>
          ----
</ process>
```

The XML schema of PROCESS_BLOCK is shown below:

```
<xsd:complexType name="PROCESS_BLOCK">
    <xsd:sequence>
        <xsd:element name="block" type="BLOCK"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="processType" type="xsd:string" use="required"/>
     <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.12.3 BLOCK schema

The schema type BLOCK can be optionally used to represent a procedural block of statements as specified in the source language. A block is modeled as a container for a block bin and information such as list of statements and parent process. Additionally, a block may contain a list of nested blocks representing the hierarchy within the block. Table 9-45 describes the BLOCK schema items

Table 9-45 — BLOCK schema items

Schema item Name	Description	UCIS type	Schema type	Optional
statementId	A list of statements contained in the block. This does not include any coverage bins, but is meant for information only.		STATEMENT_ID	Yes
hierarchicalBlock	A list of blocks nested in this block for coverage.		BLOCK	Yes
blockBin	Bin containing the coverage information for this block.		BLOCK_BIN	Yes
blockId	Statement identification for the block.		STATEMENT_ID	No
parentProcess	Type of process such as always and initial that contains this block.		string	Yes
objAttributes	Attributes commonly associated with an object (alias, exclude, excludedReason and weight).			
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of BLOCK is shown below:

```
<block parentProcess=”always”>
      <statementId file="5" line="24" inlineCount="2"/>
      <statementId file="5" line="25" inlineCount="1"/>         ----
      <blockBin >
           <contents nameComponent="block” typeComponent=":22:"  coverageCount="3">
              <historyNodeId>5</historyNodeId>
            </contents>
      </blockBin>
      <blockId file="5" line="24" inlineCount="1"/>
      <hierarchicalBlock parentProcess="always"/>
        ---------
      <hierarchicalBlock />
</block>
```

The XML schema of BLOCK is shown below:

```
<xsd:complexType name="BLOCK">
      <xsd:sequence>
        <xsd:element name="statementId" type="STATEMENT_ID"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="hierarchicalBlock" type="BLOCK"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="blockBin" type="BIN"/>
        <xsd:element name="blockId" type="STATEMENT_ID"/>
        <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
```

```
      </xsd:sequence>
      <xsd:attribute name="parentProcess" type="xsd:string"/>
     <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
```

9.13 BRANCH_COVERAGE schema

The schema complex type BRANCH_COVERAGE encapsulates coverage of branches of branching statements such as if and case. The branch coverage is modeled as a list of branching statements with coverage for each branch of a branching statement from the list. A branching statement may be nested with other branching statements. Table 9-46 describes the BRANCH_COVERAGE schema items

Table 9-46 — BRANCH_COVERAGE schema items

Schema item Name	Description	UCIS type	Schema type	Optional
statement	A list of branching statements for coverage.		BRANCH_STATEMENT	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes
metricAttributes	Attributes associated with a coverage metric.		metricAttributes	Yes

An instance of branch coverage representation in XML is shown below:

```
<branchCoverage metricMode="BRANCH_MODE1" >
      <statement branchExpr="(a|b)" statementType="case" >
        <id file="1" line="1" inlineCount="1"/>
        <branch>
              <bin >
              ------
              </bin>
            </branch>
        <branch>
              <bin >
              ------
              </bin>
            </branch>
    </statement>
</branchCoverage>
```

The XML schema for BRANCH_COVERAGE is shown below:

```
<xsd:complexType name="BRANCH_COVERAGE">
   <xsd:sequence>
     <xsd:element name="statement" type="BRANCH_STATEMENT"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
</xsd:complexType>
```

9.13.1 BRANCH_STATEMENT schema

A branching statement is modeled with a XML schema type BRANCH_STATEMENT. It contains information about the statement and a list of branches. Table 9-47 describes the BRANCH_STATEMENT schema items

Table 9-47 — BRANCH_STATEMENT schema items

Schema item Name	Description	UCIS type	Schema type	Optional
id	Source identification of the statement.		STATEMENT_ID	Yes
branch	A list of branches of this branching statement.		BRANCH	Yes
branchExpr	Expression associated with the branching statement such as case statement.		string	Yes
statementType	Type of the branching statement such as if and case.		string	Yes
objAttributes	Attributes commonly associated with an object (alias, exclude, excludedReason and weight).		-	-
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of BRANCH_STATEMENT is shown below:

```
<branchStatement branchExpr="(a&&b)" statementType="case”>
        <id file="8" line="11" inlineCount="1"/>
        <branch>
          <id file="8" line="13" inlineCount="1"/>
       ------
       </branch>
       <branch>
          <id file="8" line="16" inlineCount="1"/>
       ------
       </branch>
       ------
       <userAttr key="cond" type="unique"/>
</branchStatement>
```

The corresponding XML schema of BRANCH_STATEMENT is shown below:

```
<xsd:complexType name="BRANCH_STATEMENT">
      <xsd:sequence>
        <xsd:element name="id" type="STATEMENT_ID"/>
        <xsd:element name="branch" type="BRANCH"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="branchExpr" type="xsd:string"/>
      <xsd:attribute name="statementType" type="xsd:string" use="required"/>
      <xsd:attributeGroup ref="objAttributes"/>
 </xsd:complexType>
```

9.13.2 BRANCH schema

Coverage of a branch of a branching statement is modeled as a XML schema complex type BRANCH. It stores information about the branch and the coverage for it. Table 9-48 describes the BRANCH schema items

Table 9-48 — BRANCH schema items

Schema item Name	Description	UCIS type	Schema type	Optional
id	Branch statement source identifier.		STATEMENT_ID	No
nestedBranch	A list of nested branching statements.		BRANCH_STATEMENT	Yes
branchBin	A bin for the branch to store coverage information.		BIN	Yes
userAttr	One or more user defined UCIS attributes.		USER_ATTR	Yes

An XML example of BRANCH is shown below:

```
<branch>
          <id file="8" line="13" inlineCount="1"/>
          <nestedBranch branchExpr="string" statementType="case">
            <id file="13" line="14" inlineCount="1"/>
            ------
           </nestedBranch>
          <nestedBranch branchExpr="string" statementType="string">
                <id file="8" line="14" inlineCount="2"/>
            ------
           </nestedBranch>
          <branchBin >
              <contents nameComponent="true" typeComponent=":6:" coverageCount="0">
                  <historyNodeId>7</historyNodeId>
              </contents>
          </branchBin>
</branch>
```

The XML schema for BRANCH is shown below:

```
<xsd:complexType name="BRANCH">
      <xsd:sequence>
        <xsd:element name="id" type="STATEMENT_ID"/>
        <xsd:element name="nestedBranch" type="BRANCH_STATEMENT"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="branchBin" type="BIN"/>
        <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
 </xsd:complexType>
```

9.13.3 An example branch coverage in XML

Consider the following SystemVerilog code:

```
module top;                                  // line 1
 int x = 0;                                  // line 2
 int y = 0;                                  // line 3
 always @ (x)                                // line 4
  case (x)                                   // line 5
   0: if (y==0) $display("both 0");          // line 6
   1: if (y==1) $display("both 1");          // line 7
      else $display("x is 1 but y is not"); // line 8
  endcase                                    // line 9
 initial begin                               // line 10
  #1; y = 1;                                 // line 11
  #1; x = 1;                                 // line 12
  #1; y = 0;                                 // line 13
 end                                         // line 14
endmodule                                    // line 15
```

9.13.3.1 XML representation of the branch coverage for the example

```
<branchCoverage metricMode="branchMettric1" weight="1">
      <branchStatement branchExpr="(x)" statementType="case">
        <id file="3" line="5" inlineCount="1"/>
        <branch>
          <id file="3" line="6" inlineCount="1"/>
          <nestedBranch branchExpr="(y==0)" statementType="if">
            <id file="3" line="6" inlineCount="2"/>
            <branch>
              <id file="3" line="6" inlineCount="2"/>
              <branchBin>
                <contents nameComponent="true"
                          typeComponent=":6:" coverageCount="5">
                  <historyNodeId>5</historyNodeId>
                  <historyNodeId>34</historyNodeId>
                </contents>
              </branchBin>
            </branch>
            <branch>
              <id file="3" line="6" inlineCount="2"/>
              <branchBin>
                <contents nameComponent="all_false_bin"
                          typeComponent=":6:" coverageCount="3">
                  <historyNodeId>5</historyNodeId>
                  <historyNodeId>34</historyNodeId>
                </contents>
              </branchBin>
            </branch>
          </nestedBranch>
          <branchBin >
            <contents nameComponent="true"
                      typeComponent=":6:" coverageCount="0">
              <historyNodeId>5</historyNodeId>
              <historyNodeId>34</historyNodeId>
            </contents>
          </branchBin>
        </branch>

        <branch>
          <id file="3" line="7" inlineCount="1"/>
          <nestedBranch branchExpr="(y==1)" statementType="if">
            <id file="3" line="7" inlineCount="2"/>
            <branch>
              <id file="3" line="7" inlineCount="2"/>
              <branchBin>
```

```
            <contents nameComponent="true"
                      typeComponent=":6:" coverageCount="3">
              <historyNodeId>5</historyNodeId>
              <historyNodeId>34</historyNodeId>
            </contents>
          </branchBin>
        </branch>
        <branch>
          <id file="3" line="7" inlineCount="2"/>
          <branchBin>
            <contents nameComponent="all_false_bin"
                      typeComponent=":6:"
                      coverageCount="2">
              <historyNodeId>5</historyNodeId>
              <historyNodeId>34</historyNodeId>
            </contents>
          </branchBin>
        </branch>
      </nestedBranch>
      <branchBin >
        <contents nameComponent="true"
                  typeComponent=":6:" coverageCount="2">
          <historyNodeId>5</historyNodeId>
          <historyNodeId>34</historyNodeId>
        </contents>
      </branchBin>
    </branch>
  </branchStatement>
  <userAttr key="string" type="str" len="3264"/>
  <userAttr key="string" type="str" len="9990"/>
</branchCoverage>
```

9.14 Complete XML schema for UCIS

Below is the complete XML schema for UCIS.

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns="UCIS" targetNamespace="UCIS"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!--  Definition of NAME_VALUE -->
  <xsd:complexType name="NAME_VALUE">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
  </xsd:complexType>
 <!--  Definition of USER_ATTR -->
  <xsd:complexType name="USER_ATTR" mixed="true">
      <xsd:attribute name="key" type="xsd:string" use="required"/>
      <!-- type restrictions for the asttribute: -->
      <xsd:attribute name="type" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
            <xsd:enumeration value="int"/>
            <xsd:enumeration value="float"/>
            <xsd:enumeration value="double"/>
            <!-- string value: -->
            <xsd:enumeration value="str"/>
            <!-- binary value: -->
            <xsd:enumeration value="bits"/>
            <xsd:enumeration value="int64"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>
      <!-- length of binary attribute (type=="bits"): -->
      <xsd:attribute name="len" type="xsd:integer"/>
  </xsd:complexType>
 <!--  Definition of BIN_CONTENTS -->
  <xsd:complexType name="BIN_CONTENTS">
    <xsd:sequence>
      <xsd:element name="historyNodeId" type="xsd:nonNegativeInteger"
                          minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="nameComponent" type="xsd:string"/>
    <xsd:attribute name="typeComponent" type="xsd:string"/>
    <xsd:attribute name="coverageCount" type="xsd:nonNegativeInteger" use="required"/>
  </xsd:complexType>
  <!--  Definition of BIN_ATTRIBUTES-->
  <xsd:attributeGroup name="binAttributes">
    <xsd:attribute name="alias" type="xsd:string"/>
    <xsd:attribute name="coverageCountGoal" type="xsd:nonNegativeInteger"/>
    <xsd:attribute name="excluded" type="xsd:boolean" default="false"/>
    <xsd:attribute name="excludedReason" type="xsd:string"/>
    <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
  </xsd:attributeGroup>
    <!-- Definition of BIN-->
  <xsd:complexType name="BIN">
    <xsd:sequence>
      <xsd:element name="contents" type="BIN_CONTENTS"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="binAttributes"/>
  </xsd:complexType>
  <!--  Definition of OBJECT_ATTRIBUTES-->
  <xsd:attributeGroup name="objAttributes">
    <xsd:attribute name="alias" type="xsd:string"/>
    <xsd:attribute name="excluded" type="xsd:boolean" default="false"/>
    <xsd:attribute name="excludedReason" type="xsd:string"/>
    <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
  </xsd:attributeGroup>
```

```
 <!--  Definition of METRIC_ATTRIBUTES-->
 <xsd:attributeGroup name="metricAttributes">
   <xsd:attribute name="metricMode" type="xsd:string"/>
   <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
 </xsd:attributeGroup>
 <!--  Definition of SOURCE_FILE-->
 <xsd:complexType name="SOURCE_FILE">
     <xsd:attribute name="fileName" type="xsd:string" use="required"/>
     <xsd:attribute name="id" type="xsd:positiveInteger" use="required"/>
 </xsd:complexType>
 <!--  Definition of HISTORY_NODE-->
 <xsd:complexType name="HISTORY_NODE">
  <xsd:sequence>
     <xsd:element name="userAttr" type="USER_ATTR"
                         minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="historyNodeId" type="xsd:nonNegativeInteger" use="required"/>
  <xsd:attribute name="parentId" type="xsd:nonNegativeInteger"/>
  <xsd:attribute name="logicalName" type="xsd:string" use="required"/>
  <xsd:attribute name="physicalName" type="xsd:string"/>
  <xsd:attribute name="kind" type="xsd:string"/>
  <xsd:attribute name="testStatus" type="xsd:boolean" use="required"/>
  <xsd:attribute name="simtime" type="xsd:double"/>
  <xsd:attribute name="timeunit" type="xsd:string"/>
  <xsd:attribute name="runCwd" type="xsd:string"/>
  <xsd:attribute name="cpuTime" type="xsd:double"/>
  <xsd:attribute name="seed" type="xsd:string"/>
  <xsd:attribute name="cmd" type="xsd:string"/>
  <xsd:attribute name="args" type="xsd:string"/>
  <xsd:attribute name="compulsory" type="xsd:string"/>
  <xsd:attribute name="date" type="xsd:dateTime" use="required"/>
  <xsd:attribute name="userName" type="xsd:string"/>
  <xsd:attribute name="cost" type="xsd:decimal"/>
  <xsd:attribute name="toolCategory" type="xsd:string" use="required"/>
  <xsd:attribute name="ucisVersion" type="xsd:string" use="required"/>
  <xsd:attribute name="vendorId" type="xsd:string" use="required"/>
  <xsd:attribute name="vendorTool" type="xsd:string" use="required"/>
  <xsd:attribute name="vendorToolVersion" type="xsd:string" use="required"/>
  <xsd:attribute name="sameTests" type="xsd:nonNegativeInteger"/>
  <xsd:attribute name="comment" type="xsd:string"/>
 </xsd:complexType>
 <!--  Definition of DIMENSION  -->
 <xsd:complexType name="DIMENSION">
     <xsd:attribute name="left" type="xsd:integer" use="required"/>
     <xsd:attribute name="right" type="xsd:integer" use="required"/>
     <xsd:attribute name="downto" type="xsd:boolean" use="required"/>
</xsd:complexType>
 <!--  Definition of TOGGLE  -->
 <xsd:complexType name="TOGGLE">
   <xsd:sequence>
     <xsd:element name="bin" type="BIN"/>
   </xsd:sequence>
   <xsd:attribute name="from" type="xsd:string" use="required"/>
   <xsd:attribute name="to" type="xsd:string" use="required"/>
 </xsd:complexType>
 <!--  Definition of TOGGLE_BIT -->
 <xsd:complexType name="TOGGLE_BIT">
     <xsd:sequence>
       <xsd:element name="index" type="xsd:nonNegativeInteger"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="toggle" type="TOGGLE"
                         minOccurs="1" maxOccurs="unbounded"/>
       <xsd:element name="userAttr" type="USER_ATTR"
                    minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string" use="required"/>
     <xsd:attribute name="key" type="xsd:string" use="required"/>
     <xsd:attributeGroup ref="objAttributes"/>
   </xsd:complexType>
 <!--  Definition of TOGGLE_OBJECT -->
 <xsd:complexType name="TOGGLE_OBJECT">
     <xsd:sequence>
       <xsd:element name="dimension" type="DIMENSION"
                         minOccurs="0" maxOccurs="unbounded"/>
```

```
      <xsd:element name="id" type="STATEMENT_ID"/>
      <xsd:element name="toggleBit" type="TOGGLE_BIT"
                   minOccurs="1" maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="key" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="portDirection" type="xsd:string"/>
    <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
 <!--  Definition of METRIC_MODE  -->
 <xsd:complexType name="METRIC_MODE">
    <xsd:sequence>
      <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="metricMode" type="xsd:string" use="required"/>
</xsd:complexType>

 <!--  Definition of TOGGLE_COVERAGE -->
 <xsd:complexType name="TOGGLE_COVERAGE">
  <xsd:sequence>
    <xsd:element name="toggleObject" type="TOGGLE_OBJECT"
                          minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="metricMode" type="METRIC_MODE"
                   minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="userAttr" type="USER_ATTR"
                   minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="metricAttributes"/>
 </xsd:complexType>
  <!-- Definition of LINE_ID -->
 <xsd:complexType name="LINE_ID">
    <xsd:attribute name="file" type="xsd:positiveInteger" use="required"/>
    <xsd:attribute name="line" type="xsd:positiveInteger" use="required"/>
 </xsd:complexType>
 <!-- Definition of STATEMENT_ID -->
 <xsd:complexType name="STATEMENT_ID">
    <xsd:attribute name="file" type="xsd:positiveInteger" use="required"/>
    <xsd:attribute name="line" type="xsd:positiveInteger" use="required"/>
    <xsd:attribute name="inlineCount" type="xsd:positiveInteger" use="required"/>
 </xsd:complexType>
 <!-- Definition of STATEMENT-->
 <xsd:complexType name="STATEMENT">
   <xsd:sequence>
    <xsd:element name="id" type="STATEMENT_ID"/>
    <xsd:element name="bin" type="BIN"/>
    <xsd:element name="userAttr" type="USER_ATTR"
    minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="objAttributes"/>
 </xsd:complexType>
 <!--  Definition of BLOCK -->
 <xsd:complexType name="BLOCK">
    <xsd:sequence>
      <xsd:element name="statementId" type="STATEMENT_ID"
      minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="hierarchicalBlock" type="BLOCK"
      minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="blockBin" type="BIN"/>
      <xsd:element name="blockId" type="STATEMENT_ID"/>
      <xsd:element name="userAttr" type="USER_ATTR"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="parentProcess" type="xsd:string"/>
    <xsd:attributeGroup ref="objAttributes"/>
 </xsd:complexType>
 <!--  Definition of PROCESS_BLOCK -->
 <xsd:complexType name="PROCESS_BLOCK">
   <xsd:sequence>
      <xsd:element name="block" type="BLOCK"
      minOccurs="0" maxOccurs="unbounded"/>
```

```
      <xsd:element name="userAttr" type="USER_ATTR"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="processType" type="xsd:string" use="required"/>
    <xsd:attributeGroup ref="objAttributes"/>
  </xsd:complexType>
  <!--  Definition of BLOCK_COVERAGE -->
  <xsd:complexType name="BLOCK_COVERAGE">
    <xsd:sequence>
       <xsd:choice>
          <xsd:element name="process" type="PROCESS_BLOCK"
            minOccurs="1" maxOccurs="unbounded"/>
          <xsd:element name="block" type="BLOCK"
            minOccurs="1" maxOccurs="unbounded"/>
          <xsd:element name="statement" type="STATEMENT"
            minOccurs="1" maxOccurs="unbounded"/>
       </xsd:choice>
       <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
  </xsd:complexType>
   <!--  Definition of EXPR -->
  <xsd:complexType name="EXPR">
      <xsd:sequence>
        <xsd:element name="id" type="STATEMENT_ID"/>
        <xsd:element name="subExpr" type="xsd:string"
        minOccurs="1" maxOccurs="unbounded"/>
        <xsd:element name="bin" type="BIN"
        minOccurs="1" maxOccurs="unbounded"/>
        <xsd:element name="hierarchicalExpr" type="EXPR"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="key" type="xsd:string" use="required"/>
      <xsd:attribute name="exprString" type="xsd:string" use="required"/>
      <xsd:attribute name="index" type="xsd:nonNegativeInteger" use="required"/>
      <xsd:attribute name="width" type="xsd:nonNegativeInteger" use="required"/>
      <xsd:attribute name="statementType" type="xsd:string"/>
      <xsd:attributeGroup ref="objAttributes"/>
  </xsd:complexType>
  <!--  Definition of CONDITION_COVERAGE -->
  <xsd:complexType name="CONDITION_COVERAGE">
   <xsd:sequence>
     <xsd:element name="expr" type="EXPR"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
                          minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
  </xsd:complexType>
  <!--  Definition of BRANCH -->
  <xsd:complexType name="BRANCH">
      <xsd:sequence>
        <xsd:element name="id" type="STATEMENT_ID"/>
        <xsd:element name="nestedBranch" type="BRANCH_STATEMENT"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="branchBin" type="BIN"/>
        <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
 </xsd:complexType>
  <!--  Definition of BRANCH_STATEMENT -->
  <xsd:complexType name="BRANCH_STATEMENT">
      <xsd:sequence>
        <xsd:element name="id" type="STATEMENT_ID"/>
        <xsd:element name="branch" type="BRANCH"
        minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="userAttr" type="USER_ATTR"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="branchExpr" type="xsd:string"/>
```

```
     <xsd:attribute name="statementType" type="xsd:string" use="required"/>
     <xsd:attributeGroup ref="objAttributes"/>
</xsd:complexType>
 <!--  Definition of BRANCH_COVERAGE -->
 <xsd:complexType name="BRANCH_COVERAGE">
  <xsd:sequence>
    <xsd:element name="statement" type="BRANCH_STATEMENT"
    minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="userAttr" type="USER_ATTR"
    minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="metricAttributes"/>
 </xsd:complexType>
 <!-- Definition of FSM_STATE -->
 <xsd:complexType name="FSM_STATE">
   <xsd:sequence>
     <xsd:element name="stateBin" type="BIN"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="stateName" type="xsd:string"/>
   <xsd:attribute name="stateValue" type="xsd:string"/>
 </xsd:complexType>
 <!-- Definition of FSM_TRANSITION -->
 <xsd:complexType name="FSM_TRANSITION">
   <xsd:sequence>
     <xsd:element name="state" type="xsd:string"
     minOccurs="2" maxOccurs="unbounded"/>
     <xsd:element name="transitionBin" type="BIN"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
 </xsd:complexType>
 <!--  Definition of FSM -->
 <xsd:complexType name="FSM">
     <xsd:sequence>
       <xsd:element name="state" type="FSM_STATE"
                        minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="stateTransition" type="FSM_TRANSITION"
                        minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="userAttr" type="USER_ATTR"
                    minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string"/>
     <xsd:attribute name="type" type="xsd:string"/>
     <xsd:attribute name="width" type="xsd:positiveInteger"/>
     <xsd:attributeGroup ref="objAttributes"/>
 </xsd:complexType>
 <!--  Definition of FSM_COVERAGE -->
 <xsd:complexType name="FSM_COVERAGE">
  <xsd:sequence>
    <xsd:element name="fsm" type="FSM"
    minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="userAttr" type="USER_ATTR"
    minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="metricAttributes"/>
 </xsd:complexType>
 <!--  Definition of ASSERTION -->
 <xsd:complexType name="ASSERTION">
     <xsd:sequence>
       <xsd:element name="coverBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
       <xsd:element name="passBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
       <xsd:element name="failBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
       <xsd:element name="vacuousBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
       <xsd:element name="disabledBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
       <xsd:element name="attemptBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
       <xsd:element name="activeBin" type="BIN"
```

```
            minOccurs="0" maxOccurs="1"/>
          <xsd:element name="peakActiveBin" type="BIN"
            minOccurs="0" maxOccurs="1"/>
          <xsd:element name="userAttr" type="USER_ATTR"
          minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="assertionKind" type="xsd:string" use="required"/>
        <xsd:attributeGroup ref="objAttributes"/>
  </xsd:complexType>
  <!--  Definition of ASSERTION_COVERAGE -->
  <xsd:complexType name="ASSERTION_COVERAGE">
   <xsd:sequence>
     <xsd:element name="assertion" type="ASSERTION"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attributeGroup ref="metricAttributes"/>
  </xsd:complexType>
  <!-- Definition of SEQUENCE -->
  <xsd:complexType name="SEQUENCE">
    <xsd:sequence>
      <xsd:element name="contents" type="BIN_CONTENTS"/>
      <xsd:element name="seqValue" type="xsd:integer"
      minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Definition of RANGE_VALUE -->
  <xsd:complexType name="RANGE_VALUE">
    <xsd:sequence>
      <xsd:element name="contents" type="BIN_CONTENTS"/>
    </xsd:sequence>
    <xsd:attribute name="from" type="xsd:integer" use="required"/>
    <xsd:attribute name="to" type="xsd:integer" use="required"/>
  </xsd:complexType>
  <!--  Definition of COVERPOINT_BIN -->
  <xsd:complexType name="COVERPOINT_BIN">
    <xsd:sequence>
       <xsd:choice>
         <xsd:element name="range" type="RANGE_VALUE"
         minOccurs="1" maxOccurs="unbounded"/>
         <xsd:element name="sequence" type="SEQUENCE"
         minOccurs="1" maxOccurs="unbounded"/>
       </xsd:choice>
       <xsd:element name="userAttr" type="USER_ATTR"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="type" type="xsd:string" use="required"/>
  </xsd:complexType>
  <!--  Definition of CROSS_BIN -->
  <xsd:complexType name="CROSS_BIN">
   <xsd:sequence>
     <xsd:element name="index" type="xsd:integer"
        minOccurs="1" maxOccurs="unbounded"/>
     <xsd:element name="contents" type="BIN_CONTENTS"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="type" type="xsd:string" default="default"/>
   <xsd:attribute name="alias" type="xsd:string"/>
  </xsd:complexType>
 <!--  Definition of CROSS_OPTIONS -->
  <xsd:complexType name="CROSS_OPTIONS">
      <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="goal" type="xsd:nonNegativeInteger" default="100"/>
      <xsd:attribute name="comment" type="xsd:string" default=""/>
      <xsd:attribute name="at_least" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="cross_num_print_missing" type="xsd:nonNegativeInteger"
default="0"/>
 </xsd:complexType>
  <!--  Definition of CROSS -->
  <xsd:complexType name="CROSS">
```

```
   <xsd:sequence>
     <xsd:element name="options" type="CROSS_OPTIONS"/>
     <xsd:element name="crossExpr" type="xsd:string"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="crossBin" type="CROSS_BIN"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use="required"/>
   <xsd:attribute name="alias" type="xsd:string"/>
  </xsd:complexType>
 <!--  Definition of COVERPOINT_OPTIONS -->
  <xsd:complexType name="COVERPOINT_OPTIONS">
      <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="goal" type="xsd:nonNegativeInteger" default="100"/>
      <xsd:attribute name="comment" type="xsd:string" default=""/>
      <xsd:attribute name="at_least" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="detect_overlap" type="xsd:boolean" default="false"/>
      <xsd:attribute name="auto_bin_max" type="xsd:nonNegativeInteger" default="64"/>
 </xsd:complexType>
  <!--  Definition of COVERPOINT -->
  <xsd:complexType name="COVERPOINT">
   <xsd:sequence>
     <xsd:element name="options" type="COVERPOINT_OPTIONS"/>
     <xsd:element name="coverpointBin" type="COVERPOINT_BIN"
     minOccurs="1" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use="required"/>
   <xsd:attribute name="alias" type="xsd:string"/>
   <xsd:attribute name="exprString" type="xsd:string"/>
  </xsd:complexType>
  <!--  Definition of CG_ID -->
  <xsd:complexType name="CG_ID">
     <xsd:sequence>
        <xsd:element name="cginstSourceId" type="STATEMENT_ID"/>
        <xsd:element name="cgSourceId" type="STATEMENT_ID"/>
     </xsd:sequence>
     <xsd:attribute name="cgName" type="xsd:string" use="required"/>
     <xsd:attribute name="moduleName" type="xsd:string" use="required"/>
  </xsd:complexType>
 <!--  Definition of CGINST_OPTIONS -->
  <xsd:complexType name="CGINST_OPTIONS">
      <xsd:attribute name="weight" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="goal" type="xsd:nonNegativeInteger" default="100"/>
      <xsd:attribute name="comment" type="xsd:string" default=""/>
      <xsd:attribute name="at_least" type="xsd:nonNegativeInteger" default="1"/>
      <xsd:attribute name="detect_overlap" type="xsd:boolean" default="false"/>
      <xsd:attribute name="auto_bin_max" type="xsd:nonNegativeInteger" default="64"/>
      <xsd:attribute name="cross_num_print_missing" type="xsd:nonNegativeInteger"
default="0"/>
      <xsd:attribute name="per_instance" type="xsd:boolean" default="false"/>
      <xsd:attribute name="merge_instances" type="xsd:boolean" default="false"/>
 </xsd:complexType>
  <!--  Definition of CGINSTANCE -->
  <xsd:complexType name="CGINSTANCE">
   <xsd:sequence>
     <xsd:element name="options" type="CGINST_OPTIONS"/>
     <xsd:element name="cgId" type="CG_ID"/>
     <xsd:element name="cgParms" type="NAME_VALUE"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="coverpoint" type="COVERPOINT"
     minOccurs="1" maxOccurs="unbounded"/>
     <xsd:element name="cross" type="CROSS"
     minOccurs="0" maxOccurs="unbounded"/>
     <xsd:element name="userAttr" type="USER_ATTR"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   <xsd:attribute name="key" type="xsd:string" use="required"/>
```

```
    <xsd:attribute name="alias" type="xsd:string"/>
    <xsd:attribute name="excluded" type="xsd:boolean" default="false"/>
    <xsd:attribute name="excludedReason" type="xsd:string"/>
   </xsd:complexType>
   <!--  Definition of COVERGROUP_COVERAGE -->
   <xsd:complexType name="COVERGROUP_COVERAGE">
    <xsd:sequence>
      <xsd:element name="cgInstance" type="CGINSTANCE"
      minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="userAttr" type="USER_ATTR"
      minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="metricAttributes"/>
   </xsd:complexType>
  <!-- Definition of INSTANCE_COVERAGE -->
   <xsd:complexType name="INSTANCE_COVERAGE">
     <xsd:sequence>
       <xsd:element name="designParameter" type="NAME_VALUE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="id" type="STATEMENT_ID"/>
       <xsd:element name="toggleCoverage" type="TOGGLE_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="blockCoverage" type="BLOCK_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="conditionCoverage" type="CONDITION_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="branchCoverage" type="BRANCH_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="fsmCoverage" type="FSM_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="assertionCoverage" type="ASSERTION_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="covergroupCoverage" type="COVERGROUP_COVERAGE"
       minOccurs="0" maxOccurs="unbounded"/>
       <xsd:element name="userAttr" type="USER_ATTR"
                          minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string" use="required"/>
     <xsd:attribute name="key" type="xsd:string" use="required"/>
     <xsd:attribute name="instanceId" type="xsd:integer"/>
     <xsd:attribute name="alias" type="xsd:string"/>
     <xsd:attribute name="moduleName" type="xsd:string"/>
     <xsd:attribute name="parentInstanceId" type="xsd:integer"/>
   </xsd:complexType>
   <!-- Definition of UCIS -->
   <xsd:element name="UCIS">
     <xsd:complexType>
       <xsd:sequence>
       <xsd:element name="sourceFiles" type="SOURCE_FILE"
                          minOccurs="1" maxOccurs="unbounded"/>
         <xsd:element name="historyNodes" type="HISTORY_NODE"
                          minOccurs="1" maxOccurs="unbounded"/>
         <xsd:element name="instanceCoverages" type="INSTANCE_COVERAGE"
                          minOccurs="1" maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="ucisVersion" type="xsd:string" use="required"/>
       <xsd:attribute name="writtenBy" type="xsd:string" use="required"/>
       <xsd:attribute name="writtenTime" type="xsd:dateTime" use="required"/>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Annex A Use case examples

This annex contains the following sections:

The material in this annex is not part of the UCIS.

A.1 UCIS use cases

Understanding the 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. This section treats how to use the API. It makes some assumptions about the data model. It also treats specific use scenarios.

The API has the following design requirements:

— Represents all kinds of coverage created by a simulator or a formal tool: covergroups, statement, block, branch, condition, expression, toggle, FSM, assertions, covers and formal verification coverage.
— Supports all coverage reports and GUIs in the simulator.
— Is useful for verification plan traceability.
— Is extensible: able to store data from third parties with arbitrary coverage hierarchies and arbitrary data.
— Uses space efficiently when written from a verification tool.
— Uses space efficiently when generating reports.

— Is time-efficient to retrieve summary coverage data from a file.
— Is usable in a standalone fashion to read, modify, or create coverage data.

This section starts with the easiest use models and progresses to the more complex. It is not exhaustive. It is illustrative, focusing on a number of important use models, with an aim to describing them concisely and conceptually.

A.2 UCISDB access modes

There are these ways to open a UCISDB file:

— **In-memory**: open the UCISDB file so that the entire UCISDB 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 file name, the file is opened and closed within an API function, but the user specifies 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 narrowly prescribed 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 UCISDB data structures, and so minimizes the memory profile of the writing application.

Of these modes, the first one to discuss is the easiest: in-memory mode. Others will be discussed in separate sections.

The database handle type of the UCIS is `ucisT` 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 (see Chapter 8, "UCIS API Functions"). Opening a database in-memory is trivially easy:

```
ucisT db = ucis_Open(filename);
```

If the database is non-NULL, the open succeeded and the database handle may be used to access all data in the database. Note the database is not tied in any way to the file on the file system. The database exists entirely in memory, and may be re-written 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 sub-sets or instance tree sub-sets or by coverage type sub-sets. Without worrying about sub-sets of the database, the basic write call is this one:

```
ucis_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.) See Chapter 8, "UCIS API Functions" for details.

Finally, the database in memory is de-allocated with this call:

```
ucis_Close(db);
```

A.3 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 UCIS error handler. If the API application is linked with a verification tool, installation of an error handler will not be allowed because the verification tool should already be linked with one.

The basic error handler looks something like this. All examples will have one.

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

The error-handler is installed as follows:

```
ucis_RegisterErrorHandler(error_handler, NULL);
```

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

A.4 Traversing a UCISDB in memory

A.4.1 Traversing scopes using callback method

Example from traverse-scopes.c

Note: Also refer to"traverse_scopes.c" on page 304.

```
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope;
    switch (cbdata->reason) {
    case UCIS_REASON_DU:
    case UCIS_REASON_SCOPE:
        scope = (ucisScopeT)(cbdata->obj);
        printf("%s\n",
               ucis_GetStringProperty(cbdata->db,scope,-
1,UCIS_STR_SCOPE_HIER_NAME));
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisname)
{
    ucisT db = ucis_Open(ucisname);
    if (db==NULL)
        return;
    ucis_CallBack(db,NULL,callback,NULL);
    ucis_Close(db);
}
```

This example illustrates a callback-based traversal, showing all UCIS scope types. The ucis_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 data 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.

The ucisCBDataT* 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 coveritems themselves.

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

There are many other pieces of information that can be acquired of a UCIS scope, see Chapter 8, "UCIS API Functions".

A.5 Reading coverage data

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

Example from read-coverage.c

Note: Also refer to"read_coverage.c, example 1" on page 297 and "read_coverage.c, example 2" on page 299.

```
/* Callback to report coveritem count */
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope = (ucisScopeT)(cbdata->obj);
    ucisT db = cbdata->db;
    char* name;
    ucisCoverDataT coverdata;
    ucisSourceInfoT sourceinfo;
    switch (cbdata->reason) {
    case UCIS_REASON_DU:
        /* Don't traverse data under a DU: see read-coverage2 */
        return UCIS_SCAN_PRUNE;
    case UCIS_REASON_CVBIN:
        scope = (ucisScopeT)(cbdata->obj);
        /* Get coveritem data from scope and coverindex passed in: */
        ucis_GetCoverData(db,scope,cbdata->coverindex,
                          &name,&coverdata,&sourceinfo);
        if (name!=NULL && name[0]!='\0') {
            /* Coveritem has a name, use it: */
            printf("%s%c%s: ",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   ucis_GetPathSeparator(db),name);
        } else {
            /* Coveritem has no name, use [file:line] instead: */
            printf("%s [%s:%d]: ",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   ucis_GetFileName(db,&sourceinfo.filehandle),
                   sourceinfo.line);
        }
        print_coverage_count(&coverdata);
        printf("\n");
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
```

This example skips the code coverage stored under a design unit – see the "read-coverage2" example for that, discussed below. If a design unit scope is encountered in the callback, UCIS_SCAN_PRUNE returns a value that instructs the callback generator to skip further callbacks for data structures underneath the design unit.

The callback prints something for the UCIS_REASON_CVBIN callback. This 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 ucis_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. More information is available below on how source files are stored in the UCISDB.

Coverage data is printed via this function

```
void
print_coverage_count(ucisCoverDataT* coverdata)
{
    if (coverdata->flags & UCIS_IS_32BIT) {
       /* 32-bit count: */
       printf("%d", coverdata->data.int32);
    } else if (coverdata->flags & UCIS_IS_64BIT) {
       /* 64-bit count: */
       printf("%lld", coverdata->data.int64);
    } else if (coverdata->flags & UCIS_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 shows how the coverage count must be printed. There currently are not any source inputs or tools that create the UCIS_IS_VECTOR type of coverage data, but 32-bit and 64-bit platforms each create counts of their respective integer sizes, and those must be handled gracefully.

The problem is the UCIS_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 UCIS design unit scope – and indeed the UCIS API allows that. 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.

Partial C callback example from read-coverage.c,example 2

Note: Also refer to"read_coverage.c, example 2" on page 299.

```
/* Structure type for the callback private data */
struct dustate {
    int underneath;
    int subscope_counter;
};
/* Callback to report coveritem count */
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope = (ucisScopeT)(cbdata->obj);
    ucisT db = cbdata->db;
    char* name;
    ucisCoverDataT coverdata;
    ucisSourceInfoT sourceinfo;
    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 UCIS_REASON_DU:
        du->underneath = 1; du->subscope_counter = 0; break;
    case UCIS_REASON_SCOPE:
        if (du->underneath) {
            du->subscope_counter++;
        }
        break;
```

```
case UCIS_REASON_ENDSCOPE:
    if (du->underneath) {
        if (du->subscope_counter)
            du->subscope_counter--;
        else
            du->underneath = 0;
    }
    break;
```

This requires some user data established for the callback function. 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.ucis
/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
```

A.6 Finding objects in a UCISDB

There are various methods to find named objects in a UCISDB. The most general method, available in read-streaming and in-memory modes, is to initiate a callback on all possible objects and test each one for the desired characteristics. This method is useful when multiple matches are possible, or for extended selection algorithms. For example, wildcard names, partial names, or hierarchical names rather than full Unique IDs, can match multiple objects, and the user can write callbacks for all of these cases.

The unique ID routines may be used to match Unique IDs for an in-memory database. These routines return either a single match or none (see "Scope management functions" on page 129). They will be faster than the callback methods as they parse the supplied name and therefore do not need to explore the full hierarchy to find matches.

The example below uses the callback method to match scope hierarchical names, obtained from the UCIS_STR_SCOPE_HIER_NAME property.

C example to find objects

```
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    switch (cbdata->reason) {
    case UCIS_REASON_SCOPE:
        if (strcmp(userdata,
                   ucis_GetStringProperty(cbdata->db, cbdata->obj, -1,
                                          UCIS_STR_SCOPE_HIER_NAME)) == 0)
            print_scope(cbdata->db,(ucisScopeT)(cbdata->obj));
        break;
    case UCIS_REASON_CVBIN:
        if (strcmp(userdata,
                   ucis_GetStringProperty(cbdata->db, cbdata->obj, -1,
                                          UCIS_STR_SCOPE_HIER_NAME)) == 0)
            print_coveritem(cbdata->db,(ucisScopeT)(cbdata->obj),
                            cbdata->coverindex);
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
```

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.

A.7 Incrementing coverage

The next example shows how to find a unique coveritem using the Unique ID routine, and increment its coverage. This is an in-memory example and does not use callbacks.

C example to increment coverage

```
void
example_code(const char* ucisname, const char* path)
{
    ucisT db = ucis_Open(ucisname);
    ucisScopeT scope;
    int coverindex;
    ucisCoverDataT coverdata;
    char *name;
    ucisSourceInfoT srcinfo;

    if (db==NULL)
        return;
    scope = ucis_MatchCoverByUniqueID(db, NULL,path,&coverindex);
    ucis_GetCoverData(db,scope,coverindex,&name, &coverdata, &srcinfo);
    printf("Coverbin %s count is %d - will now add 15 to it\n", name,
coverdata.data.int32);
    ucis_IncrementCover(db,scope,coverindex,15);
    ucis_GetCoverData(db,scope,coverindex,&name, &coverdata, &srcinfo);
    printf("New count is %d\n", coverdata.data.int32);
    ucis_Write(db,ucisname,
               NULL, /* save entire database */
               1, /* recurse: not necessary with NULL */
               -1); /* save all scope types */
    ucis_Close(db);
}
```

A.8 Removing data from a UCISDB

C example to remove data

```
void
example_code(const char* ucisname, const char* path)
{
    ucisT db = ucis_Open(ucisname);
    ucisScopeT scope;
    int coverindex = -1;
    int rc;
    char *name;

    if (db==NULL)
        return;

    scope = ucis_MatchScopeByUniqueID(db, NULL,path);
    if (scope) {
        printf("Removing scope %s\n",
               ucis_GetStringProperty(db,scope,
                                      coverindex,UCIS_STR_SCOPE_HIER_NAME));
        ucis_RemoveScope(db,scope);
    } else {
        scope = ucis_MatchCoverByUniqueID(db, NULL,path,&coverindex);
        if (scope) {
            ucis_GetCoverData(db,scope,coverindex,&name,NULL,NULL);
            printf("Removing cover %s/%s\n",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   name);
            rc = ucis_RemoveCover(db,scope,coverindex);
            if (rc!=0) {
                printf("Unable to remove cover %s/%s\n",
                     ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                       name);
            }
        }
    }
    if (scope == NULL) {
        printf("Unable to find object matching \"%s\"\n",path);
    } else {
        ucis_Write(db,ucisname,
                   NULL, /* save entire database */
                   1, /* recurse: not necessary with NULL */
                   -1); /* save all scope types */
    }
    ucis_Close(db);
}
```

The ucis_RemoveScope() and ucis_RemoveCover() functions are used to delete objects from the database. When a scope is removed, all its children are removed, too.

Some UCISDB scopes encode references to sibling scopes as part of their data model; for example FSM state and transition scopes are related. It is the responsibility of the caller to keep such related data models consistent when removing scopes or coveritems.

A.9 User-Defined Attributes and Tags in the UCISDB

Tags are names that are associated with scopes and/or test data records in the database. These names could be used for general purpose grouping in the database. There may be an enhancement in the future that allows a tag to reference another tag: that would pave ground for hierarchical groups of otherwise unrelated objects.

A.9.1 Tags in the UCISDB

C example to print tags

```
void
print_tags(ucisT db, ucisScopeT scope, int coverindex)
{
    ucisIteratorT iter;
    const char *tag;
    char* covername;

    iter = ucis_ObjectTagsIterate(db,scope,coverindex);

    printf("Tags for %s",ucis_GetStringProperty(db,scope,

coverindex,UCIS_STR_SCOPE_HIER_NAME));
    if (coverindex>=0) {
        ucis_GetCoverData(db,scope,coverindex,&covername,NULL,NULL);
        printf("%c%s:\n",ucis_GetPathSeparator(db),covername);
    } else {
        printf(":\n");
    }
    if (iter) {
        while (tag = ucis_ObjectTagsScan(db,iter)) {
            printf("   %s\n", tag);
        }
        ucis_FreeIterator(db, iter);
    }
}
```

This example demonstrates one of the iteration methods, in this case to iterate for tags on a single object. Note that the iterator is freed after use; this is necessary to prevent memory leaks.

A.9.2 User-Defined Attributes in the UCISDB

User-defined attributes are also names that can be associated with a UCISDB 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 ucisObjT handle to the API calls – that are called “global” or “UCIS” attributes. These are not associated with any particular object in the database but instead are associated with the database itself. (More on built-in attributes later.)
- User-defined attributes have values as well as names. The names are the so-called “key” for the values. In other words, 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.)

C example to print attributes

```
void
print_attrs(ucisT db, ucisScopeT scope, int coverindex)
{
    const char* attrname;
    ucisAttrValueT* attrvalue;
    char* covername;
    printf("Attributes for %s",ucis_GetStringProperty(db,scope,

coverindex,UCIS_STR_SCOPE_HIER_NAME));
    if (coverindex>=0) {
        ucis_GetCoverData(db,scope,coverindex,&covername,NULL,NULL);
        printf("%c%s:\n",ucis_GetPathSeparator(db),covername);
    } else {
        printf(":\n");
    }
    attrname = NULL;
    while ((attrname = ucis_AttrNext(db,(ucisObjT)scope,coverindex,
                                     attrname,&attrvalue))) {
        printf("\t%s: ", attrname);
        switch (attrvalue->type)
            {
            case UCIS_ATTR_INT:
                printf("int = %d\n", attrvalue->u.ivalue);
                break;
            case UCIS_ATTR_FLOAT:
                printf("float = %f\n", attrvalue->u.fvalue);
                break;
            case UCIS_ATTR_DOUBLE:
                printf("double = %lf\n", attrvalue->u.dvalue);
                break;
            case UCIS_ATTR_STRING:
                printf("string = '%s'\n",
                       attrvalue->u.svalue ? attrvalue->u.svalue : "(null)");
                break;
            case UCIS_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 (ucis_AttrNext(...)) */
}
```

This iterator requires a loop like this:

```
attrname = NULL;
while ((attrname = ucis_AttrNext(db,(ucisObjT)scope,coverindex,
                attrname,&attrvalue))) {
```

The assignment of attrname to NULL starts the iteration.

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

A.10 File Representation in the UCISDB

A.10.1 Creating a source file handle

A source file handle can be created, for example, through **ucis_CreateFileHandle** with appropriate arguments (as shown below). Optimizations like storing an offset value into a table of file names can be internal to UCIS implementation.

```
/*
 *  ucis_CreateFileHandle()
 *  Create and return a file handle.
 *  When the "filename" is absolute, "fileworkdir" is ignored.
 *  When the "filename" is relative, it is desirable to set "fileworkdir"
 *      to the directory path which it is relative to.
 *  Returns NULL for error.
*/
ucisFileHandleT
ucis_CreateFileHandle (
   ucisT        db,
   const char*  filename,
   const char*  fileworkdir);
```

A.11 Adding new data to a UCISDB

The single complex example “create-ucis/create_ucis.c” creates a hardcoded UCISDB from scratch. The code that it uses could be adapted – with variations – to add objects to an existing UCISDB. After all, even in the “create_ucis.c” example, the database exists: it starts out empty and is added to with each call. The subsections that follow discuss each type of object:

— “Adding a design unit to a UCISDB” on page 269
— “Adding a module instance to a UCISDB” on page 270
— “Adding a statement to a UCISDB” on page 271
— “Adding a toggle to a UCISDB” on page 272
— “Adding a covergroup to a UCISDB” on page 273

The example is not exhaustive. Statements, an enum toggle, and a covergroup are created as an illustration. To create other types of objects, consult the data model chapters:

— Chapter 4, “Introduction to the UCIS Data Model”
— Chapter 5, “Data Model Schema”
— Chapter 6, “Data Models”

It also may help to reverse-engineer UCISDB data created by a vended verification tool.

A.11.1 Adding a design unit to a UCISDB

Example from create-ucis.c

Note: Also refer to“create_ucis.c” on page 283.

```
ucisScopeT
create_design_unit(ucisT db,
                   const char* duname,
                   ucisFileHandleT file,
                   int line)
{
    ucisScopeT duscope;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = file;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    duscope = ucis_CreateScope(db,
                               NULL, /* DUs never have a parent */
                               duname,
                               &srcinfo,
                               1, /* weight */
                               UCIS_VLOG, /* source language */
                               UCIS_DU_MODULE, /* scope type */
                               /* flags: */
                               UCIS_ENABLED_STMT | UCIS_ENABLED_BRANCH |
                               UCIS_ENABLED_COND | UCIS_ENABLED_EXPR |
                               UCIS_ENABLED_FSM | UCIS_ENABLED_TOGGLE |
                               UCIS_INST_ONCE | UCIS_SCOPE_UNDER_DU);
    ucis_SetStringProperty(db,duscope,-1,UCIS_STR_DU_SIGNATURE,"FAKE DU
SIGNATURE");
    return duscope;
}
```

One cardinal rule is that design units must be created before their corresponding instances. Design units come in five types:

— UCIS_DU_MODULE: a Verilog or SystemVerilog module.
— UCIS_DU_ARCH: a VHDL architecture.
— UCIS_DU_PACKAGE: a Verilog, SystemVerilog or VHDL package.
— UCIS_DU_PROGRAM: a SystemVerilog program block.
— UCIS_DU_INTERFACE: a SystemVerlog interface.

One crucial fact about all these except packages is that differently parameterized versions of the same design unit could be merged together by the verification tool when saving a UCISDB. This is because different parameterizations may be created arbitrarily and capriciously by the optimizer.

This example does not show line numbers for its design units, but that is implementation-specific; the UCISDB may store complete source information for any object except toggles.

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

The UCIS_INST_ONCE flag is hardcoded in this case, but the user is 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 known a priori that the design unit will only ever have a single instance.

The flag UCIS_SCOPE_UNDER_DU may be required for certain tool-specific features to work correctly: it supplies the implementation for ucis_ScopeIsUnderDU() and has implications for ucis_CalcCoverageSummary(). If the flag is not set, some design-unit-oriented coverage may be mistaken as being per-instance.

The UCIS_STR_DU_SIGNATURE property enum is required to detect source code changes for the files associated with the design unit.

A.11.2 Adding a module instance to a UCISDB

There is little more to this than to use an API call.

Example from create-ucis.c

Note: Also refer to"create_ucis.c" on page 283.

```
ucis_CreateInstance(db,parent,instname,
   NULL,            /* source info: not used */
   1,               /* weight */
   UCIS_VLOG,       /* source language */
   UCIS_INSTANCE,   /* instance of module/architecture */
   duscope,         /* reference to design unit */
   UCIS_INST_ONCE); /* flags */
```

Because the UCISDB is a hierarchical data structure, the parent must be given. (If NULL, that creates the instance at the top-level, i.e., 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 UCISDB hierarchy. If the name contains odd characters, it is good practice to turn it into an escaped (or extended) identifier to allow path searching in a verification tool to work properly. The escaped identifier syntax will be VHDL style for instances under a VHDL parent, Verilog style for instances under a Verilog parent.

Source information may be given.

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

- UCIS_INSTANCE for design unit type of UCIS_DU_MODULE or UCIS_DU_ARCH.
- UCIS_PACKAGE for design unit type of UCIS_DU_PACKAGE.
- UCIS_INTERFACE for design unit type of UCIS_DU_INTERFACE.
- UCIS_PROGRAM for design unit type of UCIS_DU_PROGRAM.

The UCIS_INST_ONCE flag is set only for the case of the single instance of a given design unit. If adding an additional instance, the flag must be cleared explicitly by the user. Here is an example:

```
ucis_SetScopeFlag(db,scope,UCIS_INST_ONCE,0);
```

A.11.3 Adding a statement to a UCISDB

This has already been illustrated in the “filehandles” example. Here is a full discussion.

Example from create-ucis.c

Note: Also refer to“create_ucis.c” on page 283.

```
void
create_statement(ucisT db,
                 ucisScopeT parent,
                 ucisFileHandleT filehandle,
                 int fileno, /* 1-referenced wrt DU contributing files */
                 int line,   /* 1-referenced wrt file */
                int item,   /* 1-referenced wrt statements starting on the line */
                 int count)
{
    ucisCoverDataT coverdata;
    ucisSourceInfoT srcinfo;
    int coverindex;
    char name[25];
    /* UOR name generation */
    sprintf(name,"#stmt#%d#%d#%d#",fileno, line, item);

    coverdata.type = UCIS_STMTBIN;
    coverdata.flags = UCIS_IS_32BIT; /* data type flag */
    coverdata.data.int32 = count; /* must be set for 32 bit flag */
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 17;/* fake token # */
    coverindex = ucis_CreateNextCover(db,parent,
                                      name, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
    ucis_SetIntProperty(db,parent,coverindex,UCIS_INT_STMT_INDEX,item);
}
```

Like any object to be created in the design or test bench or verification plan hierarchy, this requires a parent. The third argument to ucis_CreateNextCover() is the name of the object.

The &coverdata argument is a pointer to the ucisCoverDataT 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 ucisCoverDataT; Chapter 8, “UCIS API Functions” has details. Statements require only the data field (the coverage count.)

The property enum, UCIS_INT_STMT_INDEX, is used to determine the ordering of the statement on a line. If the statement is the only one to appear on the line, UCIS_INT_STMT_INDEX is always 1. The second statement on a line would have value 2, etc.

A.11.4 Adding a toggle to a UCISDB

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

Example from create-ucis.c

Note: Also refer to"create_ucis.c" on page 283.

```
ucisScopeT
create_covergroup(ucisT db,
                  ucisScopeT parent,
                  const char* name,
                  ucisFileHandleT filehandle,
                  int line)
{
    ucisScopeT cvg;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvg = ucis_CreateScope(db,parent,name,
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERGROUP,
                           0); /* flags */
    /* Hardcoding attribute values for type_options: */
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_STROBE,0);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_MERGEINSTANCES,1);
    ucis_SetStringProperty(db,cvg,-1,UCIS_STR_COMMENT,"");
    return cvg;
}
```

This corresponds to a source toggle declared like so in SystemVerilog:

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

Note that toggles have only a name, not source information. Thus the NULL values passed to ucis_CreateNextCover().

The canonical name is used for wire (net) toggles, as described in the data model section for net toggles above. The exclusions flags may apply to the toggle, so those can be given, too.

Finally, 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.

A.11.5 Adding a covergroup to a UCISDB

The covergroup is created in various stages. The covergroup for the “create-ucis” example looks like this:

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

This requires creating a hierarchy like this:

- cg
- t
- a
- b

Example, top level code from create-ucis.c

Note: Also refer to“create_ucis.c” on page 283.

```
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, 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.

Example, create the covergroup from create-ucis.c

Note: Also refer to“create_ucis.c” on page 283.

```
/*
 * Create a covergroup of the given name under the given parent.
 * This hardcodes the type_options.
 */
ucisScopeT
create_covergroup(ucisT db,
                  ucisScopeT parent,
                  const char* name,
                  ucisFileHandleT filehandle,
                  int line)
{
    ucisScopeT cvg;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvg = ucis_CreateScope(db,parent,name,
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERGROUP,
                           0); /* flags */
    /* Hardcoding attribute values for type_options: */
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_STROBE,0);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_MERGEINSTANCES,1);
    ucis_SetStringProperty(db,cvg,-1,UCIS_STR_COMMENT,"");
    return cvg;
}
```

In ucis_CreateScope(), the scope type is UCIS_COVERGROUP and the source type is UCIS_VLOG. The source type could reasonably be UCIS_SV as well.

The attributes are required to have full report capability for the covergroup. Because this covergroup option.per_instance default is 0, the example creates type_option values only. Note that type_option.weight is provided directly as an argument to ucis_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 UCIS_COVERINSTANCE scopes in the covergroup subtree.

C example, create the coverpoint, from create-ucis.c

Note: Also refer to"create_ucis.c" on page 283.

```
/*
 * Create a coverpoint of the given name under the given parent.
 * This hardcodes the type_options.
 */
ucisScopeT
create_coverpoint(ucisT db,
                  ucisScopeT parent,
                  const char* name,
                  ucisFileHandleT filehandle,
                  int line)
{
    ucisScopeT cvp;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvp = ucis_CreateScope(db,parent,name,
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERPOINT,
                           0); /* flags */
    /* Hardcoding attribute values to defaults for type_options: */
    ucis_SetIntProperty(db,cvp,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvp,-1,UCIS_INT_CVG_ATLEAST,1);
    ucis_SetStringProperty(db,cvp,-1,UCIS_STR_COMMENT,"");
    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 ucisCreateScope() -- which derive from the default values for the type_option structure in coverpoint scope.

Finally, the bins are created as children of the coverpoint.

Example, creating bins as children of the coverpoint, from create-ucis.c

Note: Also refer to"create_ucis.c" on page 283.

```
/*
 * Create a coverpoint bin of the given name, etc., under the given
 * coverpoint.
 * Note: the right-hand-side value for a bin is the value(s) that can cause
 * the bin to increment if sampled.
 */
void
create_coverpoint_bin(ucisT db,
                      ucisScopeT parent,
                      const char* name,
                      ucisFileHandleT filehandle,
                      int line,
```

```
                        int at_least,
                        int count,
                        const char* binrhs) /* right-hand-side value */
{
    ucisSourceInfoT srcinfo;
    ucisCoverDataT coverdata;
    ucisAttrValueT attrvalue;
    int coverindex;
    coverdata.type = UCIS_CVGBIN;
    coverdata.flags = UCIS_IS_32BIT | UCIS_HAS_GOAL | UCIS_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 = ucis_CreateNextCover(db,parent,name,
                                      &coverdata,&srcinfo);
    /*
     * This uses a user-defined attribute, named BINRHS
     */
    attrvalue.type = UCIS_ATTR_STRING;
    attrvalue.u.svalue = binrhs;
    ucis_AttrAdd(db,parent,coverindex,"BINRHS",&attrvalue);
}
```

This is similar to previous examples, except for these data:

— UCIS_HAS_GOAL indicates that the “goal” field of ucisCoverDataT 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.

— UCIS_HAS_WEIGHT indicates that the “weight” field of the ucisCoverDataT 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 (e.g., statement bins.)

— The BINRHSVALUE attribute is one added by the verification tool that depends on knowledge of how the coverpoint is declared. The “bin rhs value” is the sampled value(s), on the right-hand-side of the “=” 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, whether it is a transition bin or not. It can be an enum value (as in the case illustrated above, if you look back at the top-level C code) or it can be another type of integral value, or transitions among those values.

Example, creating test data records for create-ucis.c

Note: Also refer to“create_ucis.c” on page 283.

```
/*
 * Create test data. For the most part, this is hardcoded.
 */
void
create_testdata(ucisT db,
                const char* ucisdb)
{
    ucisHistoryNodeT testnode;
    ucisTestDataT    testdata = {
        UCIS_TESTSTATUS_OK,  /* teststatus */
        0.0,                 /* simtime */
        "ns",                /* timeunit */
        "./",                /* runcwd */
        0.0,                 /* cputime */
        "0",                 /* seed */
        "toolname",          /* cmd */
        "command arguments", /* args */
        0,                   /* compulsory */
        "20110824143300",    /* date */
        "ucis_user",         /* username */
        0.0,                 /* cost */
```

```
        "UCIS:Simulator"     /* toolcategory */
    };

    testnode =  ucis_CreateHistoryNode(
                db,
                NULL,                /* no parent since it is the only one */
                "TestLogicalName",   /* primary key, never NULL */
                (char *) ucisdb,   /* optional physical name at creation */
                UCIS_HISTORYNODE_TEST);  /* It's a test history node */

    if (testnode) ucis_SetTestData(db, testnode, &testdata);
}
```

This example creates faked test data that is nearly identical to data created automatically by a simulator for the “create-ucis” example. The differences are in the date and userid, which cannot be reproduced since those will vary according to who runs the example when.

All of the test data attributes (arguments to the function above) correspond to attribute names that can be accessed using the UCIS attribute API (see “User-defined attribute functions” on page 126). One of the chief uses of the attribute data is to add user-defined attributes that can be added for any reason.

The format of the date is strict. Chapter 8, “UCIS API Functions” describes how it can be created (from a POSIX-compliant C library call, strftime().) The virtue of this format is that it can be sorted alphabetically – at least through the year 9999, which is likely to suffice for the lifetime of the UCIS API.

The “test script” argument to ucis_AddTest() is not used, though it could be. The verification tool arguments are created automatically and can be used to re-run the test. The verification tool arguments should be quoted such that the arguments could be passed to a shell for running with the verification tool.

The comment is typically not used, but of course can be set within the tool.

A.12 Creating a UCISDB from scratch in memory

Example from create-ucis.c

Note: Also refer to"create_ucis.c" on page 283.

```
/*
 * top-level example code
 */
void
example_code(const char* ucisdb)
{
    ucisFileHandleT filehandle;
    ucisScopeT instance, du, cvg, cvp;
    ucisT db = ucis_Open(NULL);
    create_testdata(db,ucisdb);
    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,1,6,1,17);
    create_statement(db,instance, filehandle,1,8,1,0);
    create_statement(db,instance, filehandle,1,9,2, 365);
    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 UCIS file '%s'\n", ucisdb);
    ucis_Write(db,ucisdb,NULL,1,-1);
    ucis_Close(db);
}
```

This is the top-level code that calls all the functions previously described in "Adding new data to a UCISDB" on page 269. This reproduces – with a few exceptions described in the header comment of create_ucis.c – the UCISDB is created by the verification tool from this source:

SystemVerilog example from create-ucis.c

Note: Also refer to"create_ucis.c" on page 283.

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

Many of the details have been discussed elsewhere. The only notable thing is the call to ucis_Open() with a NULL argument; this creates a completely empty UCISDB in memory, to which any data can be added.

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

The final ucis_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.

A.13 Read streaming mode

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

The layout on disk is broadly thus:

- Header with database version, etc.
- Global UCISDB 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 are written after the initial header for the scope or coveritem.
- Tail with summary data.

The presence of the tail is in some sense an implementation detail: the tail is loaded at the same time as the header.

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, etc., 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:

- ll ancestors of a given scope or coveritem.
- All design units.
- All global UCISDB attributes and other data global to the UCISDB.
- All test data records.
- The summary data used by functions described in Chapter 8, “UCIS API Functions” as pertaining to global coverage statistics.

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

The intuitive way to think of this is as read streaming mode maintaining a relatively small “window” into the data, that progresses through the file, with some global data available generally.

The following shows a simple example adapted from one of the previously discussed in-memory examples:

Example from read-streaming.c

Note: Also refer to“traverse_scopes.c, read streaming” on page 301.

```
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope;
    switch (cbdata->reason) {
    case UCIS_REASON_DU:
    case UCIS_REASON_SCOPE:
        scope = (ucisScopeT)(cbdata->obj);
```

```
        printf("%s\n",ucis_GetStringProperty(cbdata->db,scope,-
1,UCIS_STR_SCOPE_HIER_NAME));
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisname)
{
    ucis_OpenReadStream(ucisname,callback,NULL);
}
```

The read streaming mode is based on the same callback type functions as ucis_CallBack(). This example should look familiar: it is the “traverse-scopes” example. The “example_code” function is different. The database handle is only available through the callback. The path to the UCISDB file is given to the open call, and this calls the callback for each object in the database.

There is one interesting difference between the read-streaming callback of “read-streaming” and the in-memory callback of “traverse-scopes”, and that will be discussed in the next section.

A.13.1 Issues with the UCIS_INST_ONCE optimization

Recall that the UCIS_INST_ONCE optimization is for the case of a single instance of a design unit. Because code coverage is aggregated and stored in the design unit – the verification tool could do this for access to coverage reports without waiting on an aggregation algorithm – it saves significant space to store data only under the instance and not under the design unit for cases with only a single instance. The optimization is indicated with the UCIS_INST_ONCE flag on both the instance and the design unit.

To illustrate, contrast the behavior of “traverse-scopes/traverse_scopes” with “read-streaming/traverse_scopes”:

Example from traverse-scopes.c

Note: Also refer to“traverse_scopes.c” on page 304.

```
./traverse_scopes ../../data-models/fsm/test.ucis
work.top
/top/state
/top/state/states
/top/state/trans
/top
/top/state
/top/state/states
/top/state/trans
Example (“read-streaming”):
./traverse_scopes ../../data-models/fsm/test.ucis
work.top
/top
/top/state
/top/state/states
/top/state/trans
```

This illustrates both ways in which the UCIS_INST_ONCE optimization is not transparent. Note how the read-streaming example omits the scopes after the design unit, and how in the first example, the path to those scopes is relative to the instance. One of these could eventually be fixed, the other not. The two side-effects are these:

— Paths under a design unit are reported relative to the instance. In-memory, the API is “smart enough” to traverse to the instance's children when you ask for the design unit's children. However, it is not smart enough to know how you got there, so it reports the path relative to the instance. It is possible to create a different data structure – with “shadow” nodes under the design unit – to make the distinction. Eventually the UCIS API could create these nodes on demand and even compute design unit aggregation nodes for covergroups, cover directives, and assertions.

— The read-streaming mode reports only the nodes underneath the instance. This is unavoidable and can never be fixed. This is because read-streaming corresponds directly to the file format and the “stream” of data as it is read from the file. Since the data really does not exist under the design unit, the read-streaming mode application does not see it. This could be fixed by building nodes in memory, but that defeats the purpose of read-streaming mode, which is to reduce memory overhead as much as possible. So this side-effect will always exist.

A.14 Write streaming mode

Write streaming mode is a way of *writing* a UCISDB with optimally low memory overhead. This is the hardest of all use cases of the UCIS API. In general, it should be avoided unless you fall into one of the following circumstances:

— You are a professional tool developer for whom memory overhead is a crucial concern.

— You are linked with the verification tool kernel – as through PLI or VPI – and want to contribute your own data “on the fly” to a UCISDB being saved from the verification tool.

It is much easier to create a UCISDB from scratch in memory, as described earlier in this User Guide.

The “write-streaming” example shows the “create-ucis” example adapted to write streaming mode.

Example, create-ucis.c top-level code

Note: Also refer to“create_ucis.c” on page 283.

```
/*
 * top-level example code
 */
void
example_code(const char* ucisdb)
{
    ucisFileHandleT filehandle;
    ucisT db = ucis_OpenWriteStream(ucisdb);
    create_testdata(db,ucisdb);
    filehandle = create_filehandle(db,"test.sv");
    create_design_unit(db,"work.top",filehandle,1);
    create_instance(db,"top","work.top");
    create_statement(db,filehandle,1,6,1,17);
    create_statement(db,filehandle,1,8,1,0);
    create_statement(db,filehandle,1,9,2, 365);
    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");
    ucis_WriteStreamScope(db); /* terminate coverpoint */
    ucis_WriteStreamScope(db); /* terminate covergroup */
    ucis_WriteStreamScope(db); /* terminate instance */
    printf("Writing UCIS file '%s'\n", ucisdb);
    ucis_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 ucis_OpenWriteStream(), which gives the name of the output UCISDB. The concept of write streaming is that it writes to the database "as it goes along". Objects must be created in the same order as was previously explained for read-streaming mode. This imposes order-of-creation rules that must be clearly understood.

— 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 file handles must be used when needed.

— The ucis_WriteStream(db) call is used to terminate the creation of the current object. For scopes, this terminates the creation of the beginning of the scope. Literally, this creates the scope as a context, writes the name of the scope and other information to the database, so that subsequent objects are known to be created as children of that scope. ucis_WriteStream(db) is similar to a "flush" to disk. The API will flush the current object before writing the next one on any ucis_Create API function call; it calls ucbd_WriteStream() implicitly. The utility of having the explicit "flush" capability of ucis_WriteStream() is for memory re-use (as in creating

objects from a loop). For example, to set up string storage in advance of calling ucis_CreateNextCover(), the current object must be flushed before calling ucis_CreateNextCover(). The API does not always copy string storage; it makes use of the string value during ucis_Writestream(). After that, the value may be changed.

— The ucis_CreateInstanceByName() function must be used to create the instance. This is name-based for the design unit rather than using a ucisScopeT handle.

— The ucis_WriteStreamScope(db) call must be used to terminate the scope. More on this in a moment.

— The ucis_Close(db) function terminates the write to the non-volatile database as well as freeing the database handle.

In write-streaming mode, the nesting of calls creates the design hierarchy. This means that ucis_WriteStreamScope(db) is not optional. It terminates a scope. Write streaming mode maintains a "current scope". When a new scope is created, 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 ucis_WriteStreamScope(db) in the example are thus commented with the type of the scope they terminate.

Because write streaming mode has strict dependencies on order of creation, it is a difficult mode to use. It offers the most seamless mode of integration with the verification tool, when there is code linked into the verification tool through an interface like VPI.

This will require a tool-specific callback into the application code, but within that callback, UCIS API write streaming code can be written to add coverage to the current context being saved to the UCISDB.

A.15 Examples

These are complete compilable, linkable, working examples using the UCIS API, excerpted and discussed in this chapter.

A.15.1 create_ucis.c

```
/*****************************************************************************
* UCIS API Example
*
* Usage: create_ucis
*
* When built, the application creates a UCISDB, although the data in it
* has no reference to actual design data as a real database would
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
const char* UCISDBFILE = "test_API.ucis";
/*
 * Create a design unit of the given name.
 * Note: this hardcodes INST_ONCE and all code coverage enabled (without
 * extended toggle coverage).
 */
ucisScopeT
create_design_unit(ucisT db,
                   const char* duname,
                   ucisFileHandleT file,
                   int line)
{
    ucisScopeT duscope;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = file;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    duscope = ucis_CreateScope(db,
                               NULL, /* DUs never have a parent */
                               duname,
                               &srcinfo,
                               1, /* weight */
                               UCIS_VLOG, /* source language */
                               UCIS_DU_MODULE, /* scope type */
                               /* flags: */
                               UCIS_ENABLED_STMT | UCIS_ENABLED_BRANCH |
                               UCIS_ENABLED_COND | UCIS_ENABLED_EXPR |
                               UCIS_ENABLED_FSM | UCIS_ENABLED_TOGGLE |
                               UCIS_INST_ONCE | UCIS_SCOPE_UNDER_DU);
    ucis_SetStringProperty(db,duscope,-1,UCIS_STR_DU_SIGNATURE,"FAKE DU
SIGNATURE");
    return duscope;
}
/*
 * Create a filehandle from the given file in the current directory
 * (Works on UNIX variants only, because of the reliance on the PWD
 * environment variable.)
 */
ucisFileHandleT
create_filehandle(ucisT db,
                  const char* filename)
{
    ucisFileHandleT filehandle;
    const char* pwd = getenv("PWD");
    filehandle = ucis_CreateFileHandle(db,
                                       filename,
                                       pwd);
    return filehandle;
}
/*
```

```
 * Create test data. For the most part, this is hardcoded.
 */
void
create_testdata(ucisT db,
                const char* ucisdb)
{
    ucisHistoryNodeT testnode;
    ucisTestDataT    testdata = {
        UCIS_TESTSTATUS_OK,  /* teststatus */
        0.0,                 /* simtime */
        "ns",                /* timeunit */
        "./",                /* runcwd */
        0.0,                 /* cputime */
        "0",                 /* seed */
        "toolname",          /* cmd */
        "command arguments", /* args */
        0,                   /* compulsory */
        "20110824143300",    /* date */
        "ucis_user",         /* username */
        0.0,                 /* cost */
        "UCIS:Simulator"     /* toolcategory */
    };

    testnode =  ucis_CreateHistoryNode(
               db,
               NULL,                /* no parent since it is the only one */
               "TestLogicalName",   /* primary key, never NULL */
               (char *) ucisdb,   /* optional physical name at creation */
               UCIS_HISTORYNODE_TEST);  /* It's a test history node */

    if (testnode) ucis_SetTestData(db, testnode, &testdata);
}
/*
 * Create instance of the given design design unit.
 * This assumes INST_ONCE
 */
ucisScopeT
create_instance(ucisT db,
                const char* instname,
                ucisScopeT parent,
                ucisScopeT duscope)
{
    return
        ucis_CreateInstance(db,parent,instname,
                            NULL, /* source info: not used */
                            1, /* weight */
                            UCIS_VLOG, /* source language */
                            UCIS_INSTANCE, /* instance of module/architecture */
                            duscope, /* reference to design unit */
                            UCIS_INST_ONCE); /* flags */
}
/*
 * Create a statement bin under the given parent, at the given line number,
 * with the given count.
 */
void
create_statement(ucisT db,
                 ucisScopeT parent,
                 ucisFileHandleT filehandle,
                 int fileno, /* 1-referenced wrt DU contributing files */
                 int line,   /* 1-referenced wrt file */
                int item,   /* 1-referenced wrt statements starting on the line */
                 int count)
{
    ucisCoverDataT coverdata;
    ucisSourceInfoT srcinfo;
    int coverindex;
    char name[25];
    /* UOR name generation */
    sprintf(name,"#stmt#%d#%d#%d#",fileno, line, item);

    coverdata.type = UCIS_STMTBIN;
    coverdata.flags = UCIS_IS_32BIT; /* data type flag */
```

```
    coverdata.data.int32 = count; /* must be set for 32 bit flag */
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 17;/* fake token # */
    coverindex = ucis_CreateNextCover(db,parent,
                                      name, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
    ucis_SetIntProperty(db,parent,coverindex,UCIS_INT_STMT_INDEX,item);
}
/*
 * Create enum toggle
 * This hardcodes pretty much everything.
 */
void
create_enum_toggle(ucisT db,
                   ucisScopeT parent)
{
    ucisCoverDataT coverdata;
    ucisScopeT toggle;
    toggle = ucis_CreateToggle(db,parent,
                               "t", /* toggle name */
                               NULL, /* canonical name */
                               0, /* exclusions flags */
                               UCIS_TOGGLE_METRIC_ENUM, /* metric */
                               UCIS_TOGGLE_TYPE_REG,    /* type */
                               UCIS_TOGGLE_DIR_INTERNAL); /* toggle "direction" */
    coverdata.type = UCIS_TOGGLEBIN;
    coverdata.flags = UCIS_IS_32BIT; /* data type flag */
    coverdata.data.int32 = 0; /* must be set for 32 bit flag */
    ucis_CreateNextCover(db,toggle,
                         "a", /* enum name */
                         &coverdata,
                         NULL); /* no source data */
    coverdata.data.int32 = 1; /* must be set for 32 bit flag */
    ucis_CreateNextCover(db,toggle,
                         "b", /* enum name */
                         &coverdata,
                         NULL); /* no source data */
}
/*
 * Create a covergroup of the given name under the given parent.
 * This hardcodes the type_options.
 */
ucisScopeT
create_covergroup(ucisT db,
                  ucisScopeT parent,
                  const char* name,
                  ucisFileHandleT filehandle,
                  int line)
{
    ucisScopeT cvg;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvg = ucis_CreateScope(db,parent,name,
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERGROUP,
                           0); /* flags */
    /* Hardcoding attribute values for type_options: */
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_STROBE,0);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_MERGEINSTANCES,1);
    ucis_SetStringProperty(db,cvg,-1,UCIS_STR_COMMENT,"");
    return cvg;
}
/*
 * Create a coverpoint of the given name under the given parent.
 * This hardcodes the type_options.
 */
```

```
ucisScopeT
create_coverpoint(ucisT db,
                  ucisScopeT parent,
                  const char* name,
                  ucisFileHandleT filehandle,
                  int line)
{
    ucisScopeT cvp;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvp = ucis_CreateScope(db,parent,name,
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERPOINT,
                           0); /* flags */
    /* Hardcoding attribute values to defaults for type_options: */
    ucis_SetIntProperty(db,cvp,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvp,-1,UCIS_INT_CVG_ATLEAST,1);
    ucis_SetStringProperty(db,cvp,-1,UCIS_STR_COMMENT,"");
    return cvp;
}
/*
 * Create a coverpoint bin of the given name, etc., under the given
 * coverpoint.
 * Note: the right-hand-side value for a bin is the value(s) that can cause
 * the bin to increment if sampled.
 */
void
create_coverpoint_bin(ucisT db,
                      ucisScopeT parent,
                      const char* name,
                      ucisFileHandleT filehandle,
                      int line,
                      int at_least,
                      int count,
                      const char* binrhs) /* right-hand-side value */
{
    ucisSourceInfoT srcinfo;
    ucisCoverDataT coverdata;
    ucisAttrValueT attrvalue;
    int coverindex;
    coverdata.type = UCIS_CVGBIN;
    coverdata.flags = UCIS_IS_32BIT | UCIS_HAS_GOAL | UCIS_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 = ucis_CreateNextCover(db,parent,name,
                                      &coverdata,&srcinfo);
    /*
     * This uses a user-defined attribute, named BINRHS
     */
    attrvalue.type = UCIS_ATTR_STRING;
    attrvalue.u.svalue = binrhs;
    ucis_AttrAdd(db,parent,coverindex,"BINRHS",&attrvalue);
}
/*
 * top-level example code
 */
void
example_code(const char* ucisdb)
{
    ucisFileHandleT filehandle;
    ucisScopeT instance, du, cvg, cvp;
    ucisT db = ucis_Open(NULL);
    create_testdata(db,ucisdb);
    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,1,6,1,17);
    create_statement(db,instance, filehandle,1,8,1,0);
    create_statement(db,instance, filehandle,1,9,2, 365);
    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 UCIS file '%s'\n", ucisdb);
    ucis_Write(db,ucisdb,NULL,1,-1);
    ucis_Close(db);
}
/*
 * Error handler and main program
 */
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error %d: %s\n",
            errorInfo->msgno, errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(UCISDBFILE);
    return 0;
}
```

A.15.2 create_filehandles.c

```
/****************************************************************************
* UCIS API Example
*
* Usage: create_filehandles
*
* This adds some file handles to the UCIS file generated in this directory.
*
****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>

const char* UCISFILE = "test_API.ucis";
/*
 * Create a statement bin under the given parent, at the given line number,
 * with the given count.
 */
void
create_statement_with_filehandle(ucisT db,
                                 ucisScopeT parent,
                                 ucisFileHandleT filehandle,
                                 int line,
                                 int count)
{
    ucisCoverDataT coverdata;
    ucisSourceInfoT srcinfo;
    int coverindex;
    coverdata.type = UCIS_STMTBIN;
    coverdata.flags = UCIS_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 = ucis_CreateNextCover(db,parent,
                                      NULL, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
    ucis_SetIntProperty(db,parent,coverindex,UCIS_INT_STMT_INDEX,1);
}
/*
 * top-level example code
 */
void
example_code(const char* ucisfile)
{
    ucisT db = ucis_Open(ucisfile);
    const char* pwd = getenv("PWD"); /* works in UNIX shells */
    /*
     * Find the top level INSTANCE scope called top
     */
    ucisScopeT instance = ucis_MatchScopeByUniqueID(db,NULL,"/4:top");

    ucisFileHandleT filehandle;
    printf("Adding file handles and statements to UCIS file '%s'\n", ucisfile);
    filehandle = ucis_CreateFileHandle(db,
                                       "test.sv",
                                       pwd);
    create_statement_with_filehandle(db,instance,filehandle,3,1);
    ucis_Write(db,ucisfile,NULL,1,-1);
}
/*
 * Error handler and main program
 */
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error %d: %s\n",
            errorInfo->msgno, errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
```

```
        exit(1);
}
int
main(int argc, char* argv[])
{
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(UCISFILE);
    return 0;
}
```

A.15.3 dump_UIDs.c

```
/*****************************************************************************
* UCIS API Example
*
* Usage: dump_UIDs <ucisdb name> [-p <pathsep char>] [-o <output filename>]
*
* Generate a list of all the UIDs (scopes and coveritems) in the UCISDB
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>

static char ps = '/'; /* default path separator */

void error_handler(void *cr_data,
                   ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS error: %s\n", errorInfo->msgstr);

    if (errorInfo->severity == UCIS_MSG_ERROR) {
        exit(1);
    }
}

/*-----------------------------------------------------------------------------
*   cr_read_cb
*        - callback function for read streaming mode.
*----------------------------------------------------------------------------*/
ucisCBReturnT cr_read_cb(void          *userData,
                         ucisCBDataT  *cbData)
{
    ucisT db  = cbData->db;
    void* obj = cbData->obj;
    int coverindex = cbData->coverindex;

    FILE* outFile = (FILE*) userData;

    switch (cbData->reason)
    {
    case UCIS_REASON_CVBIN:
    case UCIS_REASON_DU:
    case UCIS_REASON_SCOPE:
        fprintf(outFile,
                ucis_GetStringProperty(db,obj,coverindex,UCIS_STR_UNIQUE_ID));
        fprintf(outFile, "\n");
        break;
    case UCIS_REASON_INITDB:
        ucis_SetPathSeparator(db, ps);
        break;
    default:
        break;
    }
    return UCIS_SCAN_CONTINUE;
}

/*-----------------------------------------------------------------------------
*    main
*       - process arguments
*       - register error handler
*       - open database in read streaming mode with callback
*----------------------------------------------------------------------------*/
int main(int    argc,
         char **argv)
{
    char* outFileName   = NULL;
    char* inputFileName = NULL;
    FILE* outFile       = NULL;
    int   i;
    /*-------------------------------------------------------------------
```

```
     * argument processing phase
     *------------------------------------------------------------------------*/

    for (i = 1; i < argc; i++)
    {
        switch (*argv[i])
        {
        case '-':
            switch (*(argv[i] + 1)) {
            case 'o':
                outFileName = argv[++i];
                outFile = fopen(outFileName,"w");
                if (outFile == NULL) {
                    fprintf(stderr, "Error opening %s\n", outFileName);
                    exit(1);
                }
                break;
            case 'p':
                i++;
                ps = *argv[i];
                fprintf(stderr, "Path separator used in UID will be %c\n", ps);
                break;
            default:
                fprintf(stderr, "Illegal option name %s\n", argv[i]);
                break;
            }
            break;
        default:
            inputFileName = argv[i];
            break;
        }
    }

    if (inputFileName == NULL) {
       fprintf(stderr,"Usage: dump_UIDs <ucisdb name> [-o <outfile>] [-p <pathsep
char>]]\n");
        exit(1);
    }

    if (outFile == NULL) {
        outFile = stdout;
    }

    /*------------------------------------------------------------------------
     * setup phase
     *------------------------------------------------------------------------*/

    ucis_RegisterErrorHandler(error_handler, NULL);

    /*------------------------------------------------------------------------
     * data collection phase
     *------------------------------------------------------------------------*/

    ucis_OpenReadStream(inputFileName, cr_read_cb, outFile);

    /*------------------------------------------------------------------------
     * cleanup phase
     *------------------------------------------------------------------------*/

    if (outFile != stdout) {
        fclose(outFile);
    }

    return 0;
}
```

A.15.4 find_object.c

```
/*****************************************************************************
 * UCIS API Example
 *
 * Find the given scope or coveritem in a UCIS from a hierarchical name
 *
 *****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void
print_scope(ucisT db, ucisScopeT scope)
{
    ucisSourceInfoT sourceinfo;
    ucis_GetScopeSourceInfo(db,scope,&sourceinfo);
    printf("Found scope '%s': type=%08x line=%d\n",
           ucis_GetStringProperty(db, scope, -1, UCIS_STR_SCOPE_HIER_NAME),
           ucis_GetScopeType(db,scope),
           sourceinfo.line);
}
void
print_coveritem(ucisT db, ucisScopeT scope, int coverindex)
{
    ucisSourceInfoT sourceinfo;
    ucisCoverDataT coverdata;
    char* covername;
    ucis_GetCoverData(db,scope,coverindex,&covername,&coverdata,&sourceinfo);
    printf("Found cover '%s/%s': types=%08x/%08x line=%d\n",
           ucis_GetStringProperty(db, scope, -1, UCIS_STR_SCOPE_HIER_NAME),
           covername,
           ucis_GetScopeType(db,scope),
           coverdata.type,
           sourceinfo.line);
}
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    switch (cbdata->reason) {
    case UCIS_REASON_SCOPE:
        if (strcmp(userdata,
                   ucis_GetStringProperty(cbdata->db, cbdata->obj, -1,
                                          UCIS_STR_SCOPE_HIER_NAME)) == 0)
            print_scope(cbdata->db,(ucisScopeT)(cbdata->obj));
        break;
    case UCIS_REASON_CVBIN:
        if (strcmp(userdata,
                   ucis_GetStringProperty(cbdata->db, cbdata->obj, -1,
                                          UCIS_STR_SCOPE_HIER_NAME)) == 0)
            print_coveritem(cbdata->db,(ucisScopeT)(cbdata->obj),
                            cbdata->coverindex);
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisdb, const char* path)
{
    ucisT db = ucis_Open(ucisdb);
    if (db==NULL)
        return;
    ucis_CallBack(db,NULL,callback, (void *) path);
    ucis_Close(db);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
```

```
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<3) {
        printf("Usage: %s <ucisdb name> <path to scope or cover>\n",argv[0]);
        printf("Purpose: Find a scope or coveritem by path.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1],argv[2]);
    return 0;
}
```

A.15.5 increment_cover.c

```
/*****************************************************************************
* UCIS API Example
*
* Usage: increment_cover ucisname cover_UID
*
* Increment the given coveritem in a UCISDB.
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
void
example_code(const char* ucisname, const char* path)
{
    ucisT db = ucis_Open(ucisname);
    ucisScopeT scope;
    int coverindex;
    ucisCoverDataT coverdata;
    char *name;
    ucisSourceInfoT srcinfo;

    if (db==NULL)
        return;
    scope = ucis_MatchCoverByUniqueID(db, NULL,path,&coverindex);
    ucis_GetCoverData(db,scope,coverindex,&name, &coverdata, &srcinfo);
    printf("Coverbin %s count is %d - will now add 15 to it\n", name,
coverdata.data.int32);
    ucis_IncrementCover(db,scope,coverindex,15);
    ucis_GetCoverData(db,scope,coverindex,&name, &coverdata, &srcinfo);
    printf("New count is %d\n", coverdata.data.int32);
    ucis_Write(db,ucisname,
               NULL, /* save entire database */
               1, /* recurse: not necessary with NULL */
               -1); /* save all scope types */
    ucis_Close(db);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<3) {
        printf("Usage: %s <ucisdb name> <cover UID>\n",argv[0]);
        printf("Purpose: Increment the given coveritem in a UCISDB.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1],argv[2]);
    return 0;
}
```

A.15.6 read_attrtags.c

```
/****************************************************************************
* UCIS API Example
*
* Usage: <program> ucisname UID
*
* Read attributes and tags for the given object(s) in a UCISDB.
*
****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>

void
print_tags(ucisT db, ucisScopeT scope, int coverindex)
{
    ucisIteratorT iter;
    const char *tag;
    char* covername;

    iter = ucis_ObjectTagsIterate(db,scope,coverindex);

    printf("Tags for %s",ucis_GetStringProperty(db,scope,

coverindex,UCIS_STR_SCOPE_HIER_NAME));
    if (coverindex>=0) {
        ucis_GetCoverData(db,scope,coverindex,&covername,NULL,NULL);
        printf("%c%s:\n",ucis_GetPathSeparator(db),covername);
    } else {
        printf(":\n");
    }
    if (iter) {
        while (tag = ucis_ObjectTagsScan(db,iter)) {
            printf("   %s\n", tag);
        }
        ucis_FreeIterator(db, iter);
    }
}
void
print_attrs(ucisT db, ucisScopeT scope, int coverindex)
{
    const char* attrname;
    ucisAttrValueT* attrvalue;
    char* covername;
    printf("Attributes for %s",ucis_GetStringProperty(db,scope,

coverindex,UCIS_STR_SCOPE_HIER_NAME));
    if (coverindex>=0) {
        ucis_GetCoverData(db,scope,coverindex,&covername,NULL,NULL);
        printf("%c%s:\n",ucis_GetPathSeparator(db),covername);
    } else {
        printf(":\n");
    }
    attrname = NULL;
    while ((attrname = ucis_AttrNext(db,(ucisObjT)scope,coverindex,
                                     attrname,&attrvalue))) {
        printf("\t%s: ", attrname);
        switch (attrvalue->type)
            {
            case UCIS_ATTR_INT:
                printf("int = %d\n", attrvalue->u.ivalue);
                break;
            case UCIS_ATTR_FLOAT:
                printf("float = %f\n", attrvalue->u.fvalue);
                break;
            case UCIS_ATTR_DOUBLE:
                printf("double = %lf\n", attrvalue->u.dvalue);
                break;
            case UCIS_ATTR_STRING:
                printf("string = '%s'\n",
```

```
                        attrvalue->u.svalue ? attrvalue->u.svalue : "(null)");
                    break;
                case UCIS_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 (ucis_AttrNext(...)) */
}
void
example_code(const char* ucisname, const char* path)
{
    ucisT db = ucis_Open(ucisname);
    ucisScopeT scope;
    int coverindex = -1;

    if (db==NULL)
        return;

    scope = ucis_MatchScopeByUniqueID(db, NULL,path);
    if (scope) {
        print_tags(db, scope,coverindex);
        print_attrs(db,scope,coverindex);
    } else {
        scope = ucis_MatchCoverByUniqueID(db, NULL,path,&coverindex);
        print_tags(db, scope,coverindex);
        print_attrs(db,scope,coverindex);
    }
    ucis_Close(db);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<3) {
        printf("Usage: %s <ucis name> <path to scope or cover>\n",argv[0]);
        printf("Purpose: Read attributes and tags for the given object in a
UCISDB.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1],argv[2]);
    return 0;
}
```

A.15.7 read_coverage.c, example 1

```
/*****************************************************************************
* UCIS API Example
*
* Usage: <program> ucisname
*
* Read coveritem counts in a UCISDB.
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
/*
* This function prints the coverage count or coverage vector to stdout.
*/
void
print_coverage_count(ucisCoverDataT* coverdata)
{
    if (coverdata->flags & UCIS_IS_32BIT) {
        /* 32-bit count: */
        printf("%d", coverdata->data.int32);
    } else if (coverdata->flags & UCIS_IS_64BIT) {
        /* 64-bit count: */
        printf("%lld", coverdata->data.int64);
    } else if (coverdata->flags & UCIS_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]);
        }
    }
}
/* Callback to report coveritem count */
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope = (ucisScopeT)(cbdata->obj);
    ucisT db = cbdata->db;
    char* name;
    ucisCoverDataT coverdata;
    ucisSourceInfoT sourceinfo;
    switch (cbdata->reason) {
    case UCIS_REASON_DU:
        /* Don't traverse data under a DU: see read-coverage2 */
        return UCIS_SCAN_PRUNE;
    case UCIS_REASON_CVBIN:
        scope = (ucisScopeT)(cbdata->obj);
        /* Get coveritem data from scope and coverindex passed in: */
        ucis_GetCoverData(db,scope,cbdata->coverindex,
                          &name,&coverdata,&sourceinfo);
        if (name!=NULL && name[0]!='\0') {
            /* Coveritem has a name, use it: */
            printf("%s%c%s: ",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   ucis_GetPathSeparator(db),name);
        } else {
            /* Coveritem has no name, use [file:line] instead: */
            printf("%s [%s:%d]: ",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   ucis_GetFileName(db,&sourceinfo.filehandle),
                   sourceinfo.line);
        }
        print_coverage_count(&coverdata);
        printf("\n");
        break;
    default: break;
```

```
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisname)
{
    ucisT db = ucis_Open(ucisname);
    if (db==NULL)
        return;
    ucis_CallBack(db,NULL,callback,NULL);
    ucis_Close(db);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<2) {
        printf("Usage: %s <ucis name>\n",argv[0]);
        printf("Purpose: Read coveritem counts in a UCISDB.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1]);
    return 0;
}
```

A.15.8 read_coverage.c, example 2

```
/*****************************************************************************
* UCIS API Example
*
* Usage: <program> ucisname
*
* Read coveritem counts in a UCISDB.
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
/*
* This function prints the coverage count or coverage vector to stdout.
*/
void
print_coverage_count(ucisCoverDataT* coverdata)
{
    if (coverdata->flags & UCIS_IS_32BIT) {
        /* 32-bit count: */
        printf("%d", coverdata->data.int32);
    } else if (coverdata->flags & UCIS_IS_64BIT) {
        /* 64-bit count: */
        printf("%lld", coverdata->data.int64);
    } else if (coverdata->flags & UCIS_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]);
        }
    }
}
/* Structure type for the callback private data */
struct dustate {
    int underneath;
    int subscope_counter;
};
/* Callback to report coveritem count */
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope = (ucisScopeT)(cbdata->obj);
    ucisT db = cbdata->db;
    char* name;
    ucisCoverDataT coverdata;
    ucisSourceInfoT sourceinfo;
    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 UCIS_REASON_DU:
        du->underneath = 1; du->subscope_counter = 0; break;
    case UCIS_REASON_SCOPE:
        if (du->underneath) {
            du->subscope_counter++;
        }
        break;
    case UCIS_REASON_ENDSCOPE:
        if (du->underneath) {
            if (du->subscope_counter)
                du->subscope_counter--;
            else
                du->underneath = 0;
```

```
        }
        break;
    case UCIS_REASON_CVBIN:
        scope = (ucisScopeT)(cbdata->obj);
        /* Get coveritem data from scope and coverindex passed in: */
        ucis_GetCoverData(db,scope,cbdata->coverindex,
                          &name,&coverdata,&sourceinfo);
        if (name!=NULL && name[0]!='\0') {
            /* Coveritem has a name, use it: */
            printf("%s%c%s: ",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   ucis_GetPathSeparator(db),name);
        } else {
            /* Coveritem has no name, use [file:line] instead: */
            printf("%s [%s:%d]: ",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   ucis_GetFileName(db,&sourceinfo.filehandle),
                   sourceinfo.line);
        }
        print_coverage_count(&coverdata);
        /* This is sometimes needed to disambiguate DU roll-up data: */
        if (du->underneath) {
            printf(" (FROM DU)");
        }
        printf("\n");
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisname)
{
    struct dustate du;
    ucisT db = ucis_Open(ucisname);
    if (db==NULL)
        return;
    ucis_CallBack(db,NULL,callback,&du);
    ucis_Close(db);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<2) {
        printf("Usage: %s <ucis name>\n",argv[0]);
        printf("Purpose: Read coveritem counts in a UCISDB.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1]);
    return 0;
}
```

A.15.9 traverse_scopes.c, read streaming

```
/****************************************************************************
* UCIS API Example
*
* Usage: <program> ucisname
*
* Traverse all scopes in a UCISDB in READ STREAMING.
*
****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope;
    switch (cbdata->reason) {
    case UCIS_REASON_DU:
    case UCIS_REASON_SCOPE:
        scope = (ucisScopeT)(cbdata->obj);
        printf("%s\n",ucis_GetStringProperty(cbdata->db,scope,-
1,UCIS_STR_SCOPE_HIER_NAME));
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisname)
{
    ucis_OpenReadStream(ucisname,callback,NULL);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<2) {
        printf("Usage: %s <ucis name>\n",argv[0]);
        printf("Purpose: Traverse all scopes in a UCISDB in READ STREAMING.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1]);
    return 0;
}
```

A.15.10 remove_data.c

```
/*****************************************************************************
* UCIS API Example
*
* Usage: <program> ucisname UID
*
* Remove the named objects in a UCISDB.
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>

void
example_code(const char* ucisname, const char* path)
{
    ucisT db = ucis_Open(ucisname);
    ucisScopeT scope;
    int coverindex = -1;
    int rc;
    char *name;

    if (db==NULL)
        return;

    scope = ucis_MatchScopeByUniqueID(db, NULL,path);
    if (scope) {
        printf("Removing scope %s\n",
               ucis_GetStringProperty(db,scope,
                                      coverindex,UCIS_STR_SCOPE_HIER_NAME));
        ucis_RemoveScope(db,scope);
    } else {
        scope = ucis_MatchCoverByUniqueID(db, NULL,path,&coverindex);
        if (scope) {
            ucis_GetCoverData(db,scope,coverindex,&name,NULL,NULL);
            printf("Removing cover %s/%s\n",
                   ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                   name);
            rc = ucis_RemoveCover(db,scope,coverindex);
            if (rc!=0) {
                printf("Unable to remove cover %s/%s\n",
                     ucis_GetStringProperty(db,scope,-1,UCIS_STR_SCOPE_HIER_NAME),
                       name);
            }
        }
    }
    if (scope == NULL) {
        printf("Unable to find object matching \"%s\"\n",path);
    } else {
        ucis_Write(db,ucisname,
                   NULL, /* save entire database */
                   1, /* recurse: not necessary with NULL */
                   -1); /* save all scope types */
    }
    ucis_Close(db);
}

void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
}
int
main(int argc, char* argv[])
{
    if (argc<3) {
        printf("Usage: %s <ucis name> <UID>\n",argv[0]);
        printf("Purpose: Remove the identified objects in a UCISDB.\n");
        return 1;
    }
```

```
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1],argv[2]);
    return 0;
}
```

A.15.11 traverse_scopes.c

```
/*****************************************************************************
* UCIS API Example
*
* Usage: <program> ucisname
*
* Traverse all scopes in a UCISDB.
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
ucisCBReturnT
callback(
         void* userdata,
         ucisCBDataT* cbdata)
{
    ucisScopeT scope;
    switch (cbdata->reason) {
    case UCIS_REASON_DU:
    case UCIS_REASON_SCOPE:
        scope = (ucisScopeT)(cbdata->obj);
        printf("%s\n",
               ucis_GetStringProperty(cbdata->db,scope,-
1,UCIS_STR_SCOPE_HIER_NAME));
        break;
    default: break;
    }
    return UCIS_SCAN_CONTINUE;
}
void
example_code(const char* ucisname)
{
    ucisT db = ucis_Open(ucisname);
    if (db==NULL)
        return;
    ucis_CallBack(db,NULL,callback,NULL);
    ucis_Close(db);
}
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error: %s\n", errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    if (argc<2) {
        printf("Usage: %s <ucis name>\n",argv[0]);
        printf("Purpose: Traverse all scopes in a UCISDB.\n");
        return 1;
    }
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(argv[1]);
    return 0;
}
```

A.15.12 test_bin_assoc.c

```
/*****************************************************************************
* UCIS API Example
*
* Usage: test_bin_assoc
*
*****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
const char* UCISDB = "assoc_API.ucis";
/*
 * Basic UCISDB creation requires design unit and filehandle
 */
ucisScopeT
create_design_unit(ucisT db,
                   const char* duname,
                   ucisFileHandleT file,
                   int line)
{
    ucisScopeT duscope;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = file;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    duscope = ucis_CreateScope(db,
                               NULL, /* DUs never have a parent */
                               duname,
                               &srcinfo,
                               1, /* weight */
                               UCIS_VLOG, /* source language */
                               UCIS_DU_MODULE, /* scope type */
                               /* flags: */
                               UCIS_ENABLED_STMT | UCIS_ENABLED_BRANCH |
                               UCIS_ENABLED_COND | UCIS_ENABLED_EXPR |
                               UCIS_ENABLED_FSM | UCIS_ENABLED_TOGGLE |
                               UCIS_INST_ONCE | UCIS_SCOPE_UNDER_DU);
    ucis_SetStringProperty(db,duscope,-1,UCIS_STR_DU_SIGNATURE,"FAKE DU
SIGNATURE");
    return duscope;
}
ucisFileHandleT
create_filehandle(ucisT db,
                  const char* filename)
{
    ucisFileHandleT filehandle;
    const char* pwd = getenv("PWD");
    filehandle = ucis_CreateFileHandle(db,
                                       filename,
                                       pwd);
    return filehandle;
}
/*
 * Create history nodes. We will create 10 separate test records, as if
 * this UCIS file had included 10 tests, plus a merge record to record the
 * merge process
 * The 10 handles are loaded into the TRarray for the caller.
 * This example does not show setting the testdata
 * The logical names must differ as they are the primary keys,
 * in a real environment these names should have some meaning.
 */
void
create_testdata(ucisT db,
                const char* ucisdb,
                ucisHistoryNodeT *TRarray)
{
    char logical_name[10];
    int i;
    ucisHistoryNodeT mergenode;
```

```
    /* one merge record (all the test records will be children of this record) */
    mergenode = ucis_CreateHistoryNode(db,
                                       NULL,
                                       "TopHistoryNode",
                                       (char *) ucisdb,
                                       UCIS_HISTORYNODE_MERGE);

    /*
     * Create ten test records named Test1 through Test10
     * Handles to them are stored in TRarray[1] through TRarray[10]
     */
    for (i=1; i<=10; i++) {
       sprintf(logical_name, "Test%d", i); /* force a new logical name each time */
        TRarray[i] =  ucis_CreateHistoryNode(
                                            db,
                                            mergenode,
                                            logical_name,
                                            (char *) ucisdb,
                                            UCIS_HISTORYNODE_TEST);
    }
}
ucisScopeT
create_instance(ucisT db,
                const char* instname,
                ucisScopeT parent,
                ucisScopeT duscope)
{
    return
        ucis_CreateInstance(db,parent,instname,
                            NULL, /* source info: not used */
                            1, /* weight */
                            UCIS_VLOG, /* source language */
                            UCIS_INSTANCE, /* instance of module/architecture */
                            duscope, /* reference to design unit */
                            UCIS_INST_ONCE); /* flags */
}
/*
 * Create a statement bin under the given parent, at the given line number,
 * with the given count.
 */
void
create_statement(ucisT db,
                 ucisScopeT parent,
                 ucisFileHandleT filehandle,
                 int line,
                 int count)
{
    ucisCoverDataT coverdata;
    ucisSourceInfoT srcinfo;
    int coverindex;
    coverdata.type = UCIS_STMTBIN;
    coverdata.flags = UCIS_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 = ucis_CreateNextCover(db,parent,
                                      NULL, /* name: statements have none */
                                      &coverdata,
                                      &srcinfo);
    ucis_SetIntProperty(db,parent,coverindex,UCIS_INT_STMT_INDEX,1);
}
/*
 * top-level example code
 */
void
example_code(const char* ucisdb)
{
    ucisFileHandleT filehandle;
    ucisScopeT instance, du;
    ucisT db = ucis_Open(NULL);
    ucisHistoryNodeT TRarray[11]; /* test records, TRarray[0] is unused */
    ucisHistoryNodeT test;
```

```
ucisHistoryNodeListT testlist;
ucisIteratorT bin_iterator, test_iterator;
int i, index;

create_testdata(db,ucisdb, TRarray); /* load test record handles into array */
filehandle = create_filehandle(db,"test.sv");
du = create_design_unit(db,"work.top",filehandle,0);
instance = create_instance(db,"top",NULL,du);
/*
 * Create 10 statement bins under the same instance
 * We can assume that these will be coverindexes 0-19
 */
for (i=1; i<=10; i++) {
    create_statement(db,instance,filehandle,i,1);
}

/*
 * The UCISDB now has 10 test records and 10 statement bins.
 * The history node list association routines allow definition
 * of any many-to-many association between bins and tests
 * We will demonstrate associating 3 tests with the statement bin
 * coverindex 4, and 4 tests with bin 6
 */

/*
 * First we must create the list of test nodes we want to associate with a bin
 * The steps are
 * 1. create the list object
 * 2. add each desired test object to the list
 */
testlist = ucis_CreateHistoryNodeList(db);
printf("Add tests \"%s\" \"%s\" and \"%s\" to the list\n",
       ucis_GetStringProperty(db, TRarray[3], -1, UCIS_STR_HIST_LOG_NAME),
       ucis_GetStringProperty(db, TRarray[8], -1, UCIS_STR_HIST_LOG_NAME),
       ucis_GetStringProperty(db, TRarray[9], -1, UCIS_STR_HIST_LOG_NAME)
       );
ucis_AddToHistoryNodeList(db,testlist,TRarray[3]);
ucis_AddToHistoryNodeList(db,testlist,TRarray[8]);
ucis_AddToHistoryNodeList(db,testlist,TRarray[9]);

/*
 * We have constructed a test list of three tests.
 * Next, associate this list with the chosen statement bin, coverindex 4
 * The meaning of the association is "tests that hit the bin". This
 * is a pre-defined semantic association.
 */

ucis_SetHistoryNodeListAssoc(db, instance, 4, testlist, UCIS_ASSOC_TESTHIT);

/*
 * Add another test and associate it with bin coverindex 6
 */

printf("Add test \"%s\" to the list\n",
       ucis_GetStringProperty(db, TRarray[2], -1, UCIS_STR_HIST_LOG_NAME));
ucis_AddToHistoryNodeList(db,testlist,TRarray[2]);
ucis_SetHistoryNodeListAssoc(db, instance, 6, testlist, UCIS_ASSOC_TESTHIT);

/*
 * Free the testlist; the data has been applied to the database
 * and the list itself is no longer needed
 */
ucis_FreeHistoryNodeList(db, testlist);

/*
 * Verification phase.
 * Iterate all the bins and report the associated tests
 * (we expect to discover tests associated with the 4th and 6th bins)
 */

bin_iterator = ucis_CoverIterate(db, instance, -1);
for (index = ucis_CoverScan(db,bin_iterator);
     index >= 0;
```

```
         index = ucis_CoverScan(db,bin_iterator)) {
        printf("Cover item %s Index %d tests:\n",
               ucis_GetStringProperty(db, instance, -1, UCIS_STR_SCOPE_HIER_NAME),
               index);
        testlist = ucis_GetHistoryNodeListAssoc(db, instance, index,
UCIS_ASSOC_TESTHIT);
        if (testlist) {
            test_iterator = ucis_HistoryNodeListIterate(db, testlist);
            while ((test = ucis_HistoryScan(db,test_iterator))) {
                /*
                 * 'test' is now a test history node handleUCIS_STR_HIST_LOG_NAME
                 */
                printf("   %s\n",
                       ucis_GetStringProperty(db, test, -1,
UCIS_STR_HIST_LOG_NAME));
            }
            ucis_FreeIterator(db, test_iterator);
        } else {
            printf("   0 tests found\n");
        }
    }
    ucis_FreeIterator(db, bin_iterator);
    printf("Writing UCISDB file '%s'\n", ucisdb);
    ucis_Write(db,ucisdb,NULL,1,-1);
    ucis_Close(db);
}
/*
 * Error handler and main program
 */
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error %d: %s\n",
            errorInfo->msgno, errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(UCISDB);
    return 0;
}
```

A.15.13 create_ucis.c, write streaming

```
/*****************************************************************************
 * UCIS API Example
 *
 * Usage: create_ucis
 *
 * This creates a UCISDB from scratch.
 * THIS IS A WRITE STREAMING EXAMPLE ADAPTED FROM THE SIBLING
 * "create-ucis" EXAMPLE.
 *
 *****************************************************************************/
#include "ucis.h"
#include <stdio.h>
#include <stdlib.h>
const char* UCISDB = "test_ws.ucis";
/*
 * Create a design unit of the given name.
 * Note: this hardcodes INST_ONCE and all code coverage enabled (without
 * extended toggle coverage).
 */
void
create_design_unit(ucisT db,
                   const char* duname,
                   ucisFileHandleT file,
                   int line)
{
    ucisScopeT duscope;
    ucisSourceInfoT srcinfo;
    srcinfo.filehandle = file;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    duscope = ucis_CreateScope(db,
                               NULL, /* DUs never have a parent */
                               duname,
                               &srcinfo,
                               1, /* weight */
                               UCIS_VLOG, /* source language */
                               UCIS_DU_MODULE, /* scope type */
                               /* flags: */
                               UCIS_ENABLED_STMT | UCIS_ENABLED_BRANCH |
                               UCIS_ENABLED_COND | UCIS_ENABLED_EXPR |
                               UCIS_ENABLED_FSM | UCIS_ENABLED_TOGGLE |
                               UCIS_INST_ONCE | UCIS_SCOPE_UNDER_DU);
    ucis_SetStringProperty(db,duscope,-1,UCIS_STR_DU_SIGNATURE,"FAKE DU
SIGNATURE");
    ucis_WriteStreamScope(db); /* terminate scope object write */
}
/*
 * Create a filehandle from the given file in the current directory
 * (Works on UNIX variants only, because of the reliance on the PWD
 * environment variable.)
 */
ucisFileHandleT
create_filehandle(ucisT db,
                  const char* filename)
{
    ucisFileHandleT filehandle;
    const char* pwd = getenv("PWD");
    filehandle = ucis_CreateFileHandle(db,
                                       filename,
                                       pwd);
    return filehandle;
}
/*
 * Create test data. For the most part, this is hardcoded.
 */
void
create_testdata(ucisT db,
                const char* ucisdb)
{
```

```
    ucisHistoryNodeT testnode;
    ucisTestDataT    testdata = {
        UCIS_TESTSTATUS_OK,  /* teststatus */
        0.0,                 /* simtime */
        "ns",                /* timeunit */
        "./",                /* runcwd */
        0.0,                 /* cputime */
        "0",                 /* seed */
        "toolname",          /* cmd */
        "command arguments", /* args */
        0,                   /* compulsory */
        "20110824143300",    /* date */
        "ucis_user",         /* username */
        0.0,                 /* cost */
        "UCIS:Simulator"     /* toolcategory */
    };

    testnode =  ucis_CreateHistoryNode(
               db,
               NULL,               /* no parent since it is the only one */
               "TestLogicalName",  /* primary key, never NULL */
               (char *) ucisdb,    /* optional physical name at creation */
               UCIS_HISTORYNODE_TEST);  /* It's a test history node */

    ucis_WriteStream(db); /* terminate test data write */
}
/*
 * Create instance of the given design design unit.
 * This assumes INST_ONCE
 */
void
create_instance(ucisT db,
                const char* instname,
                const char* duname)
{
    ucis_CreateInstanceByName(db,NULL, /* parent must be NULL! */
                              instname,
                              NULL, /* source info: not used */
                              1, /* weight */
                              UCIS_VLOG, /* source language */
                              UCIS_INSTANCE, /* instance of module/architecture */
                              (char*)duname, /* name of design unit */
                              UCIS_INST_ONCE); /* flags */
    ucis_WriteStream(db); /* terminate start scope write */
}
/*
 * Create a statement bin under the given parent, at the given line number,
 * with the given count.
 */
void
create_statement(ucisT db,
                 ucisFileHandleT filehandle,
                 int fileno, /* 1-referenced wrt DU contributing files */
                 int line,   /* 1-referenced wrt file */
                int item,   /* 1-referenced wrt statements starting on the line */
                 int count)
{
    ucisCoverDataT coverdata;
    ucisSourceInfoT srcinfo;
    int coverindex;
    char name[25];
    /* UOR name generation */
    sprintf(name,"#stmt#%d#%d#%d#",fileno, line, item);

    coverdata.type = UCIS_STMTBIN;
    coverdata.flags = UCIS_IS_32BIT; /* data type flag */
    coverdata.data.int32 = count; /* must be set for 32 bit flag */
    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 17;/* fake token # */
    coverindex = ucis_CreateNextCover(db,NULL, /* parent must be NULL */
                                      name,
                                      &coverdata,
```

```
                                        &srcinfo);
    ucis_SetIntProperty(db,NULL,coverindex,UCIS_INT_STMT_INDEX,item);
    ucis_WriteStream(db); /* terminate coveritem write */
}
/*
 * Create enum toggle
 * This hardcodes pretty much everything.
 */
void
create_enum_toggle(ucisT db)
{
    ucisCoverDataT coverdata;
    ucis_CreateToggle(db,NULL,
                      "t", /* toggle name */
                      NULL, /* canonical name */
                      0, /* exclusions flags */
                      UCIS_TOGGLE_METRIC_ENUM, /* metric */
                      UCIS_TOGGLE_TYPE_REG,    /* type */
                      UCIS_TOGGLE_DIR_INTERNAL); /* toggle "direction" */
    ucis_WriteStream(db); /* terminate toggle start write */
    coverdata.type = UCIS_TOGGLEBIN;
    coverdata.flags = UCIS_IS_32BIT; /* data type flag */
    coverdata.data.int32 = 0; /* must be set for 32 bit flag */
    ucis_CreateNextCover(db,NULL, /* parent must be NULL */
                         "a", /* enum name */
                         &coverdata,
                         NULL); /* no source data */
    ucis_WriteStream(db); /* terminate coveritem write */
    coverdata.data.int32 = 1; /* must be set for 32 bit flag */
    ucis_CreateNextCover(db,NULL, /* parent must be NULL */
                         "b", /* enum name */
                         &coverdata,
                         NULL); /* no source data */
    ucis_WriteStream(db); /* terminate coveritem write */
    ucis_WriteStreamScope(db); /* terminate toggle scope write */
}
/*
 * Create a covergroup of the given name under the given parent.
 * This hardcodes the type_options to the defaults.
 */
void
create_covergroup(ucisT db,
                  const char* name,
                  ucisFileHandleT filehandle,
                  int line)
{
    ucisScopeT cvg;
    ucisSourceInfoT srcinfo;

    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvg = ucis_CreateScope(db,NULL,name, /* parent must be NULL */
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERGROUP,
                           0); /* flags */
    /* Hardcoding attribute values to defaults for type_options: */
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_STMT_INDEX,1);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvg,-1,UCIS_INT_CVG_STROBE,0);
    ucis_SetStringProperty(db,cvg,-1,UCIS_STR_COMMENT,"");
    ucis_WriteStream(db); /* terminate start scope write */
}
/*
 * Create a covergroup of the given name under the given parent.
 * This hardcodes the type_options to the defaults.
 */
void
create_coverpoint(ucisT db,
                  const char* name,
                  ucisFileHandleT filehandle,
```

```
                  int line)
{
    ucisScopeT cvp;
    ucisSourceInfoT srcinfo;

    srcinfo.filehandle = filehandle;
    srcinfo.line = line;
    srcinfo.token = 0; /* fake token # */
    cvp = ucis_CreateScope(db,NULL,name, /* parent must be NULL */
                           &srcinfo,
                           1, /* from type_option.weight */
                           UCIS_VLOG, /* source language type */
                           UCIS_COVERPOINT,
                           0); /* flags */
    /* Hardcoding attribute values to defaults for type_options: */
    ucis_SetIntProperty(db,cvp,-1,UCIS_INT_SCOPE_GOAL,100);
    ucis_SetIntProperty(db,cvp,-1,UCIS_INT_CVG_ATLEAST,1);
    ucis_SetStringProperty(db,cvp,-1,UCIS_STR_COMMENT,"");
    ucis_WriteStream(db); /* terminate start scope write */
}
/*
 * Create a coverpoint bin of the given name, etc., under the given
 * coverpoint.
 * Note: the right-hand-side value for a bin is the value(s) that can cause
 * the bin to increment if sampled.
 */
void
create_coverpoint_bin(ucisT db,
                      const char* name,
                      ucisFileHandleT filehandle,
                      int line,
                      int at_least,
                      int count,
                      const char* binrhs) /* right-hand-side value */
{
    ucisSourceInfoT srcinfo;
    ucisCoverDataT coverdata;
    int coverindex;

    coverdata.type = UCIS_CVGBIN;
    coverdata.flags = UCIS_IS_32BIT | UCIS_HAS_GOAL | UCIS_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 = ucis_CreateNextCover(db,NULL,name,
                                      &coverdata,&srcinfo);
    ucis_WriteStream(db); /* terminate start scope write */
}
/*
 * top-level example code
 */
void
example_code(const char* ucisdb)
{
    ucisFileHandleT filehandle;
    ucisT db = ucis_OpenWriteStream(ucisdb);
    create_testdata(db,ucisdb);
    filehandle = create_filehandle(db,"test.sv");
    create_design_unit(db,"work.top",filehandle,1);
    create_instance(db,"top","work.top");
    create_statement(db,filehandle,1,6,1,17);
    create_statement(db,filehandle,1,8,1,0);
    create_statement(db,filehandle,1,9,2, 365);
    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");
    ucis_WriteStreamScope(db); /* terminate coverpoint */
    ucis_WriteStreamScope(db); /* terminate covergroup */
```

```
    ucis_WriteStreamScope(db); /* terminate instance */
    printf("Writing UCIS file '%s'\n", ucisdb);
    ucis_Close(db);
}
/*
 * Error handler and main program
 */
void
error_handler(void *data,
              ucisErrorT *errorInfo)
{
    fprintf(stderr, "UCIS Error %d: %s\n",
            errorInfo->msgno, errorInfo->msgstr);
    if (errorInfo->severity == UCIS_MSG_ERROR)
        exit(1);
}
int
main(int argc, char* argv[])
{
    ucis_RegisterErrorHandler(error_handler, NULL);
    example_code(UCISDB);
    return 0;
}
```

A.15.14 formal.c, formal example

```
/*****************************************************************************
* UCIS API Example
*
* Usage: formal example
*
*****************************************************************************/

#include <vector>
#include <stdlib.h>
#include <stdio.h>
#include "ucis.h"

using namespace std;

int main(int argc, char* argv[]) // Single argument is the UCISDB name
{

  /* Formal example part 1: Open UCIS DB in memory mode */

    ucisT db = NULL;
    if (argc == 2) {
        db = ucis_Open(argv[1]);
    }
    if (db == NULL) {
        fprintf(stderr,"Unable to open UCISDB, or none provided\n");
        exit(1);
    }

  /* Formal example part 2: identity the formal test runs */
    ucisHistoryNodeT test;
    vector<ucisHistoryNodeT>::iterator itest;
    ucisIteratorT iterator;
    vector<ucisHistoryNodeT> formalTests; /* put formal tests in here */

    iterator = ucis_HistoryIterate(db,NULL,UCIS_HISTORYNODE_TEST);
    while (test = ucis_HistoryScan(db, iterator)) {
      ucisFormalToolInfoT* toolInfo=NULL;
      ucisFormalEnvT formalEnv=NULL;
      char *coverageContext=NULL;
      ucis_FormalTestGetInfo(db,test,
               &toolInfo,&formalEnv,&coverageContext);
      printf("Test \"%s\"\n",
         ucis_GetStringProperty(db, test, -1, UCIS_STR_HIST_LOG_NAME));
      if (toolInfo && formalEnv) { formalTests.push_back(test); }
    }
    ucis_FreeIterator (db, iterator);

    /* Formal example part 3: get the formal status info for all assert scopes */
    ucisScopeT scope;
    iterator = ucis_ScopeIterate(db,NULL,UCIS_ASSERT);
    while ((scope = ucis_ScopeScan(db, iterator))) {
      printf("Visiting scope: %s\n", ucis_GetStringProperty(db,scope,-1,
             UCIS_STR_SCOPE_NAME));
        ucisFormalStatusT formalStatus;
        for (itest=formalTests.begin(); itest<formalTests.end(); itest++) {
            int none = ucis_GetFormalStatus(db, *itest, scope, &formalStatus);
            if (none) { continue; }
       char *status_str = "?";
            switch (formalStatus) {
            case UCIS_FORMAL_PROOF:
          status_str = "Proven";
                break;
            case UCIS_FORMAL_FAILURE:
          status_str = "Failed";
                break;
            case UCIS_FORMAL_NONE:
          status_str = "None";
                break;
            case UCIS_FORMAL_INCONCLUSIVE:
          status_str = "Inconclusive";
```

```
                break;
             case UCIS_FORMAL_VACUOUS:
           status_str = "Vacuous";
                break;
             case UCIS_FORMAL_ASSUMPTION:
           status_str = "Assumption";
                break;
             case UCIS_FORMAL_CONFLICT:
           status_str = "Conflict";
                break;
             } // end switch
         printf("Assertion %s in test %s is %s!\n",
           ucis_GetStringProperty(db,scope,-1, UCIS_STR_SCOPE_NAME),
           ucis_GetStringProperty(db, *itest, -1, UCIS_STR_TEST_NAME),
           status_str);
          } // end for all formal tests
     } // end for all assert scopes
     ucis_FreeIterator (db, iterator);
     return(0);
}
```

Annex B Header file - normative reference

The following header file is a suggested working UCIS C-language header file. This file is not part of the UCIS.

```
/************************************************************************
 *
 * Copyright© 2012 by Accellera Systems Initiative All rights reserved.
 *
 ************************************************************************/


/*------------------------------------------------------------------------
 *  General API Behavior - please read
 *
 *  The following notes describe default API behavior which may be assumed unless
 *  the description of a particular function notes otherwise.
 *
 *  Functions that return a handle will return NULL (or invalid handle)
 *  on error, the handle otherwise. Search functions returning handles
 *  return NULL if the object was not found; this is not an error and will
 *  not trigger the error reporting subsystem.
 *
 *  Functions that return a status int will return non-zero on error, 0 otherwise.
 *
 *  Functions that return strings guarantee only to keep the return string valid
 *  until the next call to the function, or the database is closed, whichever
 *  comes first. Applications must make their own copy of any string data
 *  needed after this.
 *
 *  The iteration routines may cause memory to be allocated to hold the iteration
 *  state. The ucis_FreeIterator() routine must be called when the caller is
 *  finished with the iteration, to release any memory so allocated.
 *  The iterator object is invalid after this call and must not be used again, even
 *  if an iteration was terminated before all possible objects had been returned.
 *
 *  Where number sets are used as identifiers, all numbers below 1000 are
 *  reserved for future UCIS standardization extensions unless noted.
 *  This rule applies to both enumerated and #define definitions.
 *
 *  In some routines, the combination of a scope handle and a coverindex is
 *  used to identify either a scope or a coveritem associated with the scope
 *  Unless noted, the following rules apply to these routines:
 *  - a NULL scope pointer implies the top level database scope
 *  - a coverindex value of -1 identifies the scope
 *  - a coverindex value 0 or greater identifies a coveritem
 *
 *  Removal routines immediately invalidate the handle to the removed object
 *  Removal of a scope (in-memory) removes all the children of the scope
 *
 * ------------------------------------------------------------------------*/

#ifndef UCIS_API_H
#define UCIS_API_H

#ifdef __cplusplus
extern "C" {
#endif

/* Ensure that size-critical types are defined on all OS platforms. */
#if defined (_MSC_VER)
typedef unsigned __int64 uint64_t;
typedef unsigned __int32 uint32_t;
typedef unsigned __int16 uint16_t;
typedef unsigned __int8 uint8_t;
typedef signed __int64 int64_t;
typedef signed __int32 int32_t;
typedef signed __int16 int16_t;
typedef signed __int8 int8_t;
#elif defined(__MINGW32__)
#include <stdint.h>
```

```
#elif defined(__linux)
#include <inttypes.h>
#else
#include <sys/types.h>
#endif

/*
 * Macros to control literal size warnings cross-compiler
 */

#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

/*
 * 64-bit one-hot typing
 */

typedef uint64_t ucisObjTypeT;
typedef ucisObjTypeT ucisScopeTypeT;
typedef ucisObjTypeT ucisCoverTypeT;
typedef uint64_t ucisCoverMaskTypeT;
typedef uint64_t ucisScopeMaskTypeT;



#ifndef DEFINE_UCIST
#define DEFINE_UCIST
typedef void* ucisT;          /* generic handle to a UCIS */
#endif

typedef void* ucisScopeT;       /* scope handle */
typedef void* ucisObjT;         /* either ucisScopeT or ucisHistoryNodeT */
typedef void* ucisFileHandleT;  /* file handle */
typedef void* ucisIteratorT;    /* opaque iterator object */
typedef void* ucisHistoryNodeT;

/*----------------------------------------------------------------------------
 *  Source information management
 *
 *--------------------------------------------------------------------------*/

typedef struct {
    ucisFileHandleT     filehandle;
    int                 line;
    int                 token;
} ucisSourceInfoT;

/*
 *  ucis_GetFileName()
 *  Given a file handle, return the actual file name.
 *
 */
const char*
ucis_GetFileName(
    ucisT               db,
    ucisFileHandleT     filehandle);

/*
 *  ucis_CreateFileHandle()
 *  Create and return a file handle.
 *  When the "filename" is absolute, "fileworkdir" is ignored.
 *  When the "filename" is relative, it is desirable to set "fileworkdir"
 *  to the directory path which it is relative to.
 *  Returns NULL for error.
 */
```

```
ucisFileHandleT
ucis_CreateFileHandle (
    ucisT db,
    const char* filename,
    const char* fileworkdir);

/*----------------------------------------------------------------------------
 *  Error-handling
 *
 *  Optionally use ucis_RegisterErrorHandler() before any UCIS calls.
 *  The user's error callback, a function pointer of type ucis_ErrorHandler,
 *  is called for any errors produced by the system.
 *
 *--------------------------------------------------------------------------*/
typedef enum {
    UCIS_MSG_INFO,
    UCIS_MSG_WARNING,
    UCIS_MSG_ERROR
} ucisMsgSeverityT;

typedef struct ucisErr_s {
    int msgno;                    /* Message identifier */
    ucisMsgSeverityT severity;    /* Message severity */
    const char* msgstr;           /* Raw message string */
} ucisErrorT;

typedef void (*ucis_ErrorHandler) (void* userdata, ucisErrorT* errdata);

/*
 *  ucis_RegisterErrorHandler()
 *  The registered function, if set, will be called for all API errors.
 */
void
ucis_RegisterErrorHandler(
    ucis_ErrorHandler errHandle,
    void*             userdata);

/*----------------------------------------------------------------------------
 *  Database creation and file management
 *
 *  Notes on callback order:
 *
 *      * INITDB is always the first callback.
 *      * all TEST callbacks follow; after the next non-TEST callback
 *        there will be no more TEST callbacks.
 *      * DU callbacks must precede their first associated instance SCOPE
 *        callbacks, but need not immediately precede them.
 *      * SCOPE and DU and CVBIN callbacks can occur in any order
 *        (excepting the DU before first instance rule) -- though
 *        nesting level is implied by the order of callbacks.
 *      * ENDSCOPE callbacks correspond to SCOPE and DU callbacks and
 *        imply a "pop" in the nesting of scopes and design units.
 *      * ENDDB callbacks can be used to access UCIS attributes written at the
 *        end of the file, if created in write streaming modes.
 *
 *--------------------------------------------------------------------------*/

/*
 *  ucis_Open()
 *  Create an in-memory database, optionally populating it from the given file.
 *  Returns a handle with success, NULL with failure.
 */
ucisT
ucis_Open(
    const char*     name        /* file system path */
);

/*
* ucis_OpenFromInterchangeFormat()
* Create an in-memory database, optionally populating it from the given
* interchange format file.
* Returns a handle with success, NULL with failure.
*/
ucisT
```

```
ucis_OpenFromInterchangeFormat(
     const char* name /* file system path */
);


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


/* Enum type for different callback reasons */
typedef enum {
    UCIS_REASON_INITDB,      /* Start of the database, apply initial settings */
    UCIS_REASON_DU,          /* Start of a design unit scope */
    UCIS_REASON_TEST,        /* Testdata history object */
    UCIS_REASON_SCOPE,       /* Start of a scope object */
    UCIS_REASON_CVBIN,       /* Cover item */
    UCIS_REASON_ENDSCOPE,    /* End of a scope, including design units */
    UCIS_REASON_ENDDB,       /* End of database (database handle still valid) */
    UCIS_REASON_MERGEHISTORY /* Merge history object */
} ucisCBReasonT;


/* Enum type for return type for of ucis_CBFuncT function */
typedef enum {
    UCIS_SCAN_CONTINUE = -1,    /* Continue to scan ucis file */
    UCIS_SCAN_STOP     = -2,    /* Stop scanning ucis file */
    UCIS_SCAN_PRUNE    = -3     /* Prune the scanning of the ucis file at this
                                 * node. Scanning continues but ignores
                                 * processing of this node's children.
                                 * NOTE: This enum value is currently
                                 *       disallowed in read streaming mode.
                                 */
} ucisCBReturnT;

/* Data type for read callback */
typedef struct ucisCBDataS {
    ucisCBReasonT reason;       /* Reason for this callback */
    ucisT         db;           /* Database handle, to use in APIs */
    ucisObjT      obj;          /* ucisScopeT or ucisHistoryNodeT */
    int           coverindex;   /* if UCIS_REASON_CVBIN, index of coveritem */
} ucisCBDataT;

/* Function type to use in the ucis_OpenReadStream() read API */
typedef ucisCBReturnT (*ucis_CBFuncT) (void* userdata, ucisCBDataT* cbdata);

/*
 *  ucis_OpenReadStream()
 *  Open database for streaming read mode.
 *  Returns 0 with success, -1 with failure.
 */
int
ucis_OpenReadStream(
    const char*  name,      /* file system path */
    ucis_CBFuncT cbfunc,
    void*        userdata);


/*
 *  ucis_OpenWriteStream()
 *  Open data in write streaming mode.
 *  Implementations may impose ordering restrictions in write-streaming mode.
 *  Returns a restricted database handle with success, NULL with error.
 */
ucisT
ucis_OpenWriteStream(
    const char* name);      /* file system path */

/*
 *  ucis_WriteStream()
 *  Finishes a write of current object to the persistent database file in
 *  write streaming mode.
 *  Note: this operation is like a "flush", just completing the write of
```

```
 *  whatever was most recently created in write streaming mode.  In particular,
 *  there is no harm in doing multiple ucis_WriteStream() calls, because if the
 *  current object has already been written, it will not be written again.
 *  The database handle "db" must have been previously opened with
 *  ucis_OpenWriteStream().
 *  UCIS_WRITESTREAMING mode. Returns 0 for success, and -1 for any error.
 */
int
ucis_WriteStream(
    ucisT       db);

/*
 *  ucis_WriteStreamScope()
 *  Similar to ucis_WriteStream, except that it finishes the current scope and
 *  "pops" the stream to the parent scope.  Objects created after this belong
 *  to the parent scope of the scope just ended.
 *  The database handle "db" must have been previously opened with
 *  ucis_OpenWriteStream().
 *  Returns 0 for success, and -1 for any error.
 */
int
ucis_WriteStreamScope(
    ucisT       db);

/*
 *  ucis_Write()
 *  Copy in-memory database or a subset of the in-memory database to a
 *  persistent form stored in the given file name.
 *  Use the coverflags argument to select coverage types to be saved, use -1
 *  to save everything.
 *  Note that file system path will be overwritten if it contains existing
 *  data; write permissions must exist.
 *  The database handle "db" cannot have been opened for one of the streaming
 *  modes.
 *  Returns 0 for success, and -1 for any error.
 */
int
ucis_Write(
    ucisT       db,
    const char* file,
    ucisScopeT  scope,      /* write objects under given scope, write all
                               objects if scope==NULL */
    int         recurse,    /* if true, recurse under given scope, ignored if
                               scope==NULL */
    /* ucisCoverTypeT: */
    int         covertype); /* selects coverage types to save */

/*
* ucis_WriteToInterchangeFormat()
* Copy in-memory database or a subset of the in-memory database to a
* persistent form stored in the given file name in the interchange format.
* Use the coverflags argument to select coverage types to be saved, use -1
* to save everything.
* Note that file system path will be overwritten if it contains existing
* data; write permissions must exist.
* The database handle "db" cannot have been opened for one of the
* streaming modes.
* Returns 0 for success, and -1 for any error.
*/
int
ucis_WriteToInterchangeFormat(
     ucisT        db,
     const char*  file,
     ucisScopeT   scope,       /* write objects under given scope, write all
                                  objects if scope==NULL */
     int          recurse,     /* if true, recurse under given scope, ignored if
                                  /* ucisCoverTypeT: */
     int          covertype);  /* selects coverage types to save */


/*
 *  ucis_Close()
 *  Invalidate the database handle.  Frees all memory associated with the
 *  database handle, including the in-memory image of the database if not in
```

```
 *  one of the streaming modes.  If opened with ucis_OpenWriteStream(), this
 *  also has the side-effect of closing the output file.
 *  Returns 0 with success, non-zero with failure.
 */
int
ucis_Close(
    ucisT       db);

/*
 *  ucis_SetPathSeparator()
 *  ucis_GetPathSeparator()
 *  Set or get path separator used with the UCIS.
 *  The path separator is associated with the database, different databases may
 *  have different path separators; the path separator is stored with the
 *  persistent form of the database.
 *  Both routines return -1 with error, SetPathSeparator returns 0 if OK.
 */
int
ucis_SetPathSeparator(
    ucisT           db,
    char            separator);

char
ucis_GetPathSeparator(
    ucisT           db);


/*----------------------------------------------------------------------------
 *  Property APIs
 *
 *  The property routines provide access to pre-defined properties associated
 *  with database objects.
 *  Some properties may be read-only, for example the UCIS_STR_UNIQUE_ID property
 *  may be queried but not set. An attempt to set a read-only property will
 *  result in an error.
 *  Some properties may be in-memory only, for example the UCIS_INT_IS_MODIFIED
 *  property applies only to an in-memory database and indicates that the
 *  data has been modified since it was last saved. This property is not
 *  stored in the database.
 *  See also the attribute routines for extended data management
 *--------------------------------------------------------------------------*/

/*----------------------------------------------------------------------------
 *  Integer Properties
 *--------------------------------------------------------------------------*/

typedef enum {
    UCIS_INT_IS_MODIFIED,/* Modified since opening stored UCISDB (In-memory and read
only) */
    UCIS_INT_MODIFIED_SINCE_SIM,/* Modified since end of simulation run (In-memory and
read only) */
    UCIS_INT_NUM_TESTS,/* Number of test history nodes (UCIS_HISTORYNODE_TEST) in UCISDB
*/
    UCIS_INT_SCOPE_WEIGHT,/* Scope weight */
    UCIS_INT_SCOPE_GOAL,/* Scope goal */
    UCIS_INT_SCOPE_SOURCE_TYPE,/* Scope source type (ucisSourceT) */
    UCIS_INT_NUM_CROSSED_CVPS,/* Number of coverpoints in a cross (read only) */
    UCIS_INT_SCOPE_IS_UNDER_DU,/* Scope is underneath design unit scope (read only) */
    UCIS_INT_SCOPE_IS_UNDER_COVERINSTANCE,/* Scope is underneath covergroup instance
(read only) */
    UCIS_INT_SCOPE_NUM_COVERITEMS,/* Number of coveritems underneath scope (read only)
*/
    UCIS_INT_SCOPE_NUM_EXPR_TERMS,/* Number of input ordered expr term strings delimited
by '#' */
    UCIS_INT_TOGGLE_TYPE,/* Toggle type (ucisToggleTypeT) */
    UCIS_INT_TOGGLE_DIR,/* Toggle direction (ucisToggleDirT) */
    UCIS_INT_TOGGLE_COVERED,/* Toggle object is covered */
    UCIS_INT_BRANCH_HAS_ELSE,/* Branch has an 'else' coveritem */
    UCIS_INT_BRANCH_ISCASE,/* Branch represents 'case' statement */
    UCIS_INT_COVER_GOAL,/* Coveritem goal */
    UCIS_INT_COVER_LIMIT,/* Coverage count limit for coveritem */
    UCIS_INT_COVER_WEIGHT,/* Coveritem weight */
    UCIS_INT_TEST_STATUS,/* Test run status (ucisTestStatusT) */
    UCIS_INT_TEST_COMPULSORY,/* Test run is compulsory */
```

```
    UCIS_INT_STMT_INDEX,/* Index or number of statement on a line */
    UCIS_INT_BRANCH_COUNT,/* Total branch execution count */
    UCIS_INT_FSM_STATEVAL,/* FSM state value */
    UCIS_INT_CVG_ATLEAST,/* Covergroup at_least option */
    UCIS_INT_CVG_AUTOBINMAX,/* Covergroup auto_bin_max option */
    UCIS_INT_CVG_DETECTOVERLAP,/* Covergroup detect_overlap option */
    UCIS_INT_CVG_NUMPRINTMISSING,/* Covergroup cross_num_print_missing option */
    UCIS_INT_CVG_STROBE,/* Covergroup strobe option */
    UCIS_INT_CVG_PERINSTANCE,/* Covergroup per_instance option */
    UCIS_INT_CVG_GETINSTCOV,/* Covergroup get_inst_coverage option */
    UCIS_INT_CVG_MERGEINSTANCES/* Covergroup merge_instances option */
} ucisIntPropertyEnumT;

int
ucis_GetIntProperty(
    ucisT                   db,
    ucisObjT                obj,
    int                     coverindex,
    ucisIntPropertyEnumT    property);

int
ucis_SetIntProperty(
    ucisT                   db,
    ucisObjT                obj,
    int                     coverindex,
    ucisIntPropertyEnumT    property,
    int                     value);


/*----------------------------------------------------------------------------
 *  String Properties
 *----------------------------------------------------------------------------*/
typedef enum {
    UCIS_STR_FILE_NAME,/* UCISDB file/directory name (read only) */
    UCIS_STR_SCOPE_NAME,/* Scope name */
    UCIS_STR_SCOPE_HIER_NAME,/* Hierarchical scope name */
    UCIS_STR_INSTANCE_DU_NAME,/* Instance' design unit name */
    UCIS_STR_UNIQUE_ID,/* Scope or coveritem unique-id (read only) */
    UCIS_STR_VER_STANDARD,/* Standard (Currently fixed to be "UCIS") */
    UCIS_STR_VER_STANDARD_VERSION,/* Version of standard (e.g. "2011", etc) */
    UCIS_STR_VER_VENDOR_ID,/* Vendor id (e.g. "CDNS", "MENT", "SNPS", etc) */
    UCIS_STR_VER_VENDOR_TOOL,/* Vendor tool (e.g. "Incisive", "Questa", "VCS", etc) */
    UCIS_STR_VER_VENDOR_VERSION,/* Vendor tool version (e.g. "6.5c", "Jun-12-2009",
etc) */
    UCIS_STR_GENERIC,/* Miscellaneous string data */
    UCIS_STR_ITH_CROSSED_CVP_NAME,/* Ith coverpoint name of a cross */
    UCIS_STR_HIST_CMDLINE,/* Test run command line */
    UCIS_STR_HIST_RUNCWD,/* Test run working directory */
    UCIS_STR_COMMENT,/* Comment */
    UCIS_STR_TEST_TIMEUNIT,/* Test run simulation time unit */
    UCIS_STR_TEST_DATE,/* Test run date */
    UCIS_STR_TEST_SIMARGS,/* Test run simulator arguments */
    UCIS_STR_TEST_USERNAME,/* Test run user name */
    UCIS_STR_TEST_NAME,/* Test run name */
    UCIS_STR_TEST_SEED,/* Test run seed */
    UCIS_STR_TEST_HOSTNAME,/* Test run hostname */
    UCIS_STR_TEST_HOSTOS,/* Test run hostOS */
    UCIS_STR_EXPR_TERMS,/* Input ordered expr term strings delimited by '#' */
    UCIS_STR_TOGGLE_CANON_NAME,/* Toggle object canonical name */
    UCIS_STR_UNIQUE_ID_ALIAS,/* Scope or coveritem unique-id alias */
    UCIS_STR_DESIGN_VERSION_ID,/* Version of the design or elaboration-id */
    UCIS_STR_DU_SIGNATURE,
    UCIS_STR_HIST_TOOLCATEGORY,
    UCIS_STR_HIST_LOG_NAME,
    UCIS_STR_HIST_PHYS_NAME
} ucisStringPropertyEnumT;


const char*
ucis_GetStringProperty(
    ucisT                   db,
    ucisObjT                obj,
    int                     coverindex,
    ucisStringPropertyEnumT property);
```

```
int
ucis_SetStringProperty(
    ucisT                   db,
    ucisObjT                obj,
    int                     coverindex,
    ucisStringPropertyEnumT property,
    const char*             value);

/*----------------------------------------------------------------------------
 *  Real-valued Properties
 *--------------------------------------------------------------------------*/
typedef enum {
    UCIS_REAL_HIST_CPUTIME,/* Test run CPU time */
    UCIS_REAL_TEST_SIMTIME,/* Test run simulation time */
    UCIS_REAL_TEST_COST
    UCIS_REAL_CVG_INST_AVERAGE
} ucisRealPropertyEnumT;


double
ucis_GetRealProperty(
    ucisT                   db,
    ucisObjT                obj,
    int                     coverindex,
    ucisRealPropertyEnumT   property);



int
ucis_SetRealProperty(
    ucisT                   db,
    ucisObjT                obj,
    int                     coverindex,
    ucisRealPropertyEnumT   property,
    double                  value);

/*----------------------------------------------------------------------------
 *  Object Handle Properties
 *--------------------------------------------------------------------------*/

typedef enum {
    UCIS_HANDLE_SCOPE_PARENT,/* Parent scope */
    UCIS_HANDLE_SCOPE_TOP,/* Top (root) scope */
    UCIS_HANDLE_INSTANCE_DU,/* Instance' design unit scope */
    UCIS_HANDLE_HIST_NODE_PARENT,/* Parent history node */
    UCIS_HANDLE_HIST_NODE_ROOT/* Top (root) history node */
} ucisHandleEnumT;

ucisObjT
ucis_GetHandleProperty(
    ucisT                   db,
    ucisObjT                obj,       /* scope or history node */
    ucisHandleEnumT         property);

int
ucis_SetHandleProperty(
    ucisT                   db,
    ucisObjT                obj,       /* scope or history node */
    ucisHandleEnumT         property,
    ucisObjT                value);

/*----------------------------------------------------------------------------
 *  Version query API
 *
 *--------------------------------------------------------------------------*/

typedef void* ucisVersionHandleT;


ucisVersionHandleT ucis_GetAPIVersion();
ucisVersionHandleT ucis_GetDBVersion(ucisT db);
ucisVersionHandleT ucis_GetFileVersion(char* filename_or_directory_name);
ucisVersionHandleT ucis_GetHistoryNodeVersion(ucisT db, ucisHistoryNodeT hnode);
```

```
const char*
ucis_GetVersionStringProperty (
    ucisVersionHandleT versionH,
    ucisStringPropertyEnumT property);


/*----------------------------------------------------------------------------
 *  Traversal API
 *
 *  These routines provide access to the database history node, scope and
 *  coveritem objects
 *  Iteration and callback mechanisms provide for multiple object returns.
 *  The iteration routines are only available for objects that are
 *  currently in memory.
 *  The callback mechanism is available for both in-memory and
 *  read-streaming applications (see the ucis_OpenReadStream() routine).
 *
 *  Objects may also be found from in-memory scope hierarchies by matching
 *  the Unique ID.
 *  These routines return 0 or 1 objects.
 *
 *  Generally, returned items are of known object kind (e.g. scopes, coveritems,
 *  history nodes), but the ucis_TaggedObjIterate()/ucis_TaggedObjScan mechanism
 *  may return mixed handles, i.e. a mix of history nodes, scopes, and coveritems.
 *  The ucis_ObjKind() routine may be used to indicate the typing scheme of the
 *  returned handle.
 *
 *----------------------------------------------------------------------------*/

/*
 * Enum type for different object kinds. This is a bit mask for the different
 * kinds of objects which may be tagged. Mask values may be and'ed and or'ed
 * together.
 */
typedef enum {
    UCIS_OBJ_ERROR       = 0x00000000,    /* Start of the database, apply initial
settings */
    UCIS_OBJ_HISTORYNODE = 0x00000001,    /* History node object */
    UCIS_OBJ_SCOPE       = 0x00000002,    /* Scope object */
    UCIS_OBJ_COVERITEM   = 0x00000004,    /* Coveritem object */
    UCIS_OBJ_ANY         = -1             /* All taggable types */
} ucisObjMaskT;


/*
 *  ucis_ObjKind()
 *  Given 'obj' return the kind of object that it is.
 */

ucisObjMaskT
ucis_ObjKind(ucisT db, ucisObjT obj);

ucisIteratorT
ucis_ScopeIterate(
    ucisT               db,
    ucisScopeT          scope,
    ucisScopeMaskTypeT  scopemask);

ucisScopeT
ucis_ScopeScan(
    ucisT               db,
    ucisIteratorT       iterator);

void
ucis_FreeIterator(
    ucisT               db,
    ucisIteratorT       iterator);

ucisIteratorT
ucis_CoverIterate(
    ucisT               db,
    ucisScopeT          scope,
    ucisCoverMaskTypeT  covermask);
```

```
int
ucis_CoverScan(
    ucisT               db,
    ucisIteratorT       iterator);

ucisIteratorT
ucis_TaggedObjIterate(
    ucisT               db,
    const char*         tagname);

ucisObjT
ucis_TaggedObjScan(
    ucisT               db,
    ucisIteratorT       iterator,
    int*                coverindex_p);

ucisIteratorT
ucis_ObjectTagsIterate(
    ucisT               db,
    ucisObjT            obj,
    int                 coverindex);

const char* ucis_ObjectTagsScan (
     ucis db,
     ucisIteratorT iterator );

ucisScopeT
ucis_MatchScopeByUniqueID(
    ucisT db,
    ucisScopeT scope,
    const char *uniqueID);

ucisScopeT
ucis_CaseAwareMatchScopeByUniqueID(
    ucisT db,
    ucisScopeT scope,
    const char *uniqueID);

/*
 * ucis_MatchCoverByUniqueID() and ucis_CaseAwareMatchCoverByUniqueID()
 *
 * A coveritem is defined by a combination of parental
 * scope (the returned handle) plus coverindex (returned in
 * the index pointer)
 */
ucisScopeT
ucis_MatchCoverByUniqueID(
    ucisT db,
    ucisScopeT scope,
    const char *uniqueID,
    int *index);

ucisScopeT
ucis_CaseAwareMatchCoverByUniqueID(
    ucisT db,
    ucisScopeT scope,
    const char *uniqueID,
    int *index);

/*
 *  ucis_CallBack()
 *  Visit that portion of the database rooted at and below the given starting
 *  scope. Issue calls to the callback function (cbfunc) along the way. When
 *  the starting scope (start) is NULL the entire database will be walked.
 *  Returns 0 with success, -1 with failure.
 *  In-memory mode only.
 */

int
ucis_CallBack(
    ucisT        db,
    ucisScopeT   start,             /* NULL traverses entire database */
    ucis_CBFuncT cbfunc,
```

```
    void*        userdata);

/*----------------------------------------------------------------------------
 *  Attributes (key/value pairs)
 *  Attribute key names shall be unique for the parental object
 *  Adding an attribute with a key name that already exists shall overwrite
 *  the existing attribute value.
 *--------------------------------------------------------------------------*/

typedef enum {
    UCIS_ATTR_INT,
    UCIS_ATTR_FLOAT,
    UCIS_ATTR_DOUBLE,
    UCIS_ATTR_STRING,
    UCIS_ATTR_MEMBLK,
    UCIS_ATTR_INT64
} ucisAttrTypeT;

typedef struct {
    ucisAttrTypeT 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;
    } u;
} ucisAttrValueT;

const char*
ucis_AttrNext(
    ucisT           db,
    ucisObjT        obj,        /* ucisScopeT, ucisHistoryNodeT, or NULL */
    int             coverindex, /* obj is ucisScopeT: -1 for scope, valid index
                                   for coveritem */
    const char*     key,        /* NULL to get the first one */
    ucisAttrValueT**value);

int
ucis_AttrAdd(
    ucisT           db,
    ucisObjT        obj,        /* ucisScopeT, ucisHistoryNodeT, or NULL */
    int             coverindex, /* obj is ucisScopeT: -1 for scope, valid index
                                   for coveritem */
    const char*     key,        /* New attribute key */
    ucisAttrValueT* value);     /* New attribute value */

int
ucis_AttrRemove(
    ucisT           db,
    ucisObjT        obj,        /* ucisScopeT, ucisHistoryNodeT, or NULL */
    int             coverindex, /* obj is ucisScopeT: -1 for scope, valid index
                                   for coveritem */
    const char*     key);       /* NULL to get the first one */

ucisAttrValueT*
ucis_AttrMatch(
    ucisT           db,
    ucisObjT        obj,        /* ucisScopeT, ucisHistoryNodeT, or NULL */
    int             coverindex, /* obj is ucisScopeT: -1 for scope, valid index
                                   for coveritem */
    const char*     key);

/*----------------------------------------------------------------------------
 *  History Nodes
 *
 *  History nodes are attribute collections that record the historical
 *  construction process for the database
 *--------------------------------------------------------------------------*/
```

```
typedef int   ucisHistoryNodeKindT; /* takes the #define'd types below */

/*
 * History node kinds.
 */
#define UCIS_HISTORYNODE_NONE    -1  /* no node or error */
#define UCIS_HISTORYNODE_ALL      0  /* valid only in iterate-all request */
                                     /* (no real object gets this value)  */
#define UCIS_HISTORYNODE_TEST     1  /* test leaf node (primary database) */
#define UCIS_HISTORYNODE_MERGE    2  /* merge node */

/*
 * Pre-defined tool category attribute strings
 */
#define UCIS_SIM_TOOL        "UCIS:Simulator"
#define UCIS_FORMAL_TOOL     "UCIS:Formal"
#define UCIS_ANALOG_TOOL     "UCIS:Analog"
#define UCIS_EMULATOR_TOOL   "UCIS:Emulator"
#define UCIS_MERGE_TOOL      "UCIS:Merge"


/*
 *  ucis_CreateHistoryNode()
 *  Create a HistoryNode handle of the indicated kind
 *  linked into the database db.
 *  Returns NULL for error or the history node already exists.
 *
 */
ucisHistoryNodeT
ucis_CreateHistoryNode(
    ucisT                   db,
    ucisHistoryNodeT        parent,
    char*                   logicalname,   /* primary key, never NULL */
    char*                   physicalname,
    ucisHistoryNodeKindT    kind);         /* takes predefined values, above */

/*
 *  ucis_RemoveHistoryNode()
 *  This function removes the given history node and all its descendants.
 */
int
ucis_RemoveHistoryNode(
    ucisT db,
    ucisHistoryNodeT historynode);

/*
 * UCIS_HISTORYNODE_TEST specialization
 *
 * ucisTestStatusT type is an enumerated type. When accessed directly as an attribute,
 * its attribute typing is UCIS_ATTR_INT and the returned value can be cast to
 * the type defined below.
 */

typedef enum {
    UCIS_TESTSTATUS_OK,
    UCIS_TESTSTATUS_WARNING,    /* test warning ($warning called) */
    UCIS_TESTSTATUS_ERROR,      /* test error ($error called) */
    UCIS_TESTSTATUS_FATAL,      /* fatal test error ($fatal called) */
    UCIS_TESTSTATUS_MISSING,    /* test not run yet */
    UCIS_TESTSTATUS_MERGE_ERROR /* testdata record was merged with
                                   inconsistent data values */
} ucisTestStatusT;


typedef struct {
    ucisTestStatusT teststatus;
    double simtime;
    const char* timeunit;
    const char* runcwd;
    double cputime;
    const char* seed;
    const char* cmd;
    const char* args;
    int compulsory;
```

```
    const char* date;
    const char* username;
    double cost;
    const char* toolcategory;
} ucisTestDataT;

int
ucis_SetTestData(
    ucisT db,
    ucisHistoryNodeT testhistorynode,
    ucisTestDataT* testdata);

int
ucis_GetTestData( ucisT db,
    ucisHistoryNodeT testhistorynode,
    ucisTestDataT* testdata);


ucisIteratorT
ucis_HistoryIterate (
    ucisT db,
    ucisHistoryNodeT historynode,
    ucisHistoryNodeKindT kind);

ucisHistoryNodeT
ucis_HistoryScan (
    ucisT db,
    ucisIteratorT iterator);

/*
 * Note that ucis_SetHistoryNodeParent() overwrites any existing setting
 */
int
ucis_SetHistoryNodeParent(
    ucisT                db,
    ucisHistoryNodeT     childnode,
    ucisHistoryNodeT     parentnode);

ucisHistoryNodeT
ucis_GetHistoryNodeParent(
    ucisT                    db,
    ucisHistoryNodeT         childnode);

/*
 *    ucis_GetHistoryKind (alias ucis_GetObjType)
 *
 *    This is a polymorphic function for acquiring an object type.
 *    This may return multiple bits set (specifically
 *    for history data objects). The return value MUST NOT be used as a mask.
 */
#define ucis_GetHistoryKind(db,obj) (ucisHistoryNodeKindT)ucis_GetObjType(db,obj)

/*----------------------------------------------------------------------------
 *  Scopes
 *
 *  The database is organized hierarchically, as per the design database: i.e.,
 *  there is a tree of module instances, each of a given module type.
 *
 *--------------------------------------------------------------------------*/

/* One-hot bits for ucisScopeTypeT: */
#define UCIS_TOGGLE          /* cover scope- toggle coverage scope: */ \
                             INT64_LITERAL(0x0000000000000001)
#define UCIS_BRANCH          /* cover scope- branch coverage scope: */ \
                             INT64_LITERAL(0x0000000000000002)
#define UCIS_EXPR            /* cover scope- expression coverage scope: */ \
                             INT64_LITERAL(0x0000000000000004)
#define UCIS_COND            /* cover scope- condition coverage scope: */ \
                             INT64_LITERAL(0x0000000000000008)
#define UCIS_INSTANCE        /* HDL scope- Design hierarchy instance: */ \
                             INT64_LITERAL(0x0000000000000010)
#define UCIS_PROCESS         /* HDL scope- process: */ \
                             INT64_LITERAL(0x0000000000000020)
#define UCIS_BLOCK           /* HDL scope- vhdl block, vlog begin-end: */ \
```

```
                              INT64_LITERAL(0x0000000000000040)
#define UCIS_FUNCTION         /* HDL scope- function: */ \
                              INT64_LITERAL(0x0000000000000080)
#define UCIS_FORKJOIN         /* HDL scope- Verilog fork-join block: */ \
                              INT64_LITERAL(0x0000000000000100)
#define UCIS_GENERATE         /* HDL scope- generate block: */ \
                              INT64_LITERAL(0x0000000000000200)
#define UCIS_GENERIC          /* cover scope- generic scope type: */ \
                              INT64_LITERAL(0x0000000000000400)
#define UCIS_CLASS            /* HDL scope- class type scope: */ \
                              INT64_LITERAL(0x0000000000000800)
#define UCIS_COVERGROUP       /* cover scope- covergroup type scope: */ \
                              INT64_LITERAL(0x0000000000001000)
#define UCIS_COVERINSTANCE    /* cover scope- covergroup instance scope: */ \
                              INT64_LITERAL(0x0000000000002000)
#define UCIS_COVERPOINT       /* cover scope- coverpoint scope: */ \
                              INT64_LITERAL(0x0000000000004000)
#define UCIS_CROSS            /* cover scope- cross scope: */ \
                              INT64_LITERAL(0x0000000000008000)
#define UCIS_COVER            /* cover scope- directive(SVA/PSL) cover: */ \
                              INT64_LITERAL(0x0000000000010000)
#define UCIS_ASSERT           /* cover scope- directive(SVA/PSL) assert: */ \
                              INT64_LITERAL(0x0000000000020000)
#define UCIS_PROGRAM          /* HDL scope- SV program instance: */ \
                              INT64_LITERAL(0x0000000000040000)
#define UCIS_PACKAGE          /* HDL scope- package instance: */ \
                              INT64_LITERAL(0x0000000000080000)
#define UCIS_TASK             /* HDL scope- task: */ \
                              INT64_LITERAL(0x0000000000100000)
#define UCIS_INTERFACE        /* HDL scope- SV interface instance: */ \
                              INT64_LITERAL(0x0000000000200000)
#define UCIS_FSM              /* cover scope- FSM coverage scope: */ \
                              INT64_LITERAL(0x0000000000400000)
#define UCIS_DU_MODULE        /* design unit- for instance type: */ \
                              INT64_LITERAL(0x0000000001000000)
#define UCIS_DU_ARCH          /* design unit- for instance type: */ \
                              INT64_LITERAL(0x0000000002000000)
#define UCIS_DU_PACKAGE       /* design unit- for instance type: */ \
                              INT64_LITERAL(0x0000000004000000)
#define UCIS_DU_PROGRAM       /* design unit- for instance type: */ \
                              INT64_LITERAL(0x0000000008000000)
#define UCIS_DU_INTERFACE     /* design unit- for instance type: */ \
                              INT64_LITERAL(0x0000000010000000)
#define UCIS_FSM_STATES       /* cover scope- FSM states coverage scope: */ \
                              INT64_LITERAL(0x0000000020000000)
#define UCIS_FSM_TRANS        /* cover scope- FSM transition coverage scope: */\
                              INT64_LITERAL(0x0000000040000000)
#define UCIS_COVBLOCK         /* cover scope- block coverage scope: */ \
                              INT64_LITERAL(0x0000000080000000)
#define UCIS_CVGBINSCOPE      /* cover scope- SV cover bin scope: */ \
                              INT64_LITERAL(0x0000000100000000)
#define UCIS_ILLEGALBINSCOPE /* cover scope- SV illegal bin scope: */ \
                              INT64_LITERAL(0x0000000200000000)
#define UCIS_IGNOREBINSCOPE   /* cover scope- SV ignore bin scope: */ \
                              INT64_LITERAL(0x0000000400000000)
#define UCIS_RESERVEDSCOPE    INT64_LITERAL(0xFF00000000000000)
#define UCIS_SCOPE_ERROR      /* error return code: */ \
                              INT64_LITERAL(0x0000000000000000)

#define UCIS_FSM_SCOPE      ((ucisScopeMaskTypeT)(UCIS_FSM |\
                                                  UCIS_FSM_STATES |\
                                                  UCIS_FSM_TRANS))

#define UCIS_CODE_COV_SCOPE ((ucisScopeMaskTypeT)(UCIS_BRANCH |\
                                                  UCIS_EXPR |\
                                                  UCIS_COND |\
                                                  UCIS_TOGGLE |\
                                                  UCIS_FSM_SCOPE |\
                                                  UCIS_BLOCK))

#define UCIS_DU_ANY ((ucisScopeMaskTypeT)(UCIS_DU_MODULE |\
                                          UCIS_DU_ARCH |\
                                          UCIS_DU_PACKAGE |\
                                          UCIS_DU_PROGRAM |\
```

```
                                        UCIS_DU_INTERFACE))

#define UCIS_CVG_SCOPE      ((ucisScopeMaskTypeT)(UCIS_COVERGROUP |\
                                                  UCIS_COVERINSTANCE |\
                                                  UCIS_COVERPOINT |\
                                                  UCIS_CVGBINSCOPE |\
                                                  UCIS_ILLEGALBINSCOPE |\
                                                  UCIS_IGNOREBINSCOPE |\
                                                  UCIS_CROSS))

#define UCIS_FUNC_COV_SCOPE ((ucisScopeMaskTypeT)(UCIS_CVG_SCOPE |\
                                                  UCIS_COVER))

#define UCIS_COV_SCOPE ((ucisScopeMaskTypeT)(UCIS_CODE_COV_SCOPE |\
                                             UCIS_FUNC_COV_SCOPE))



#define UCIS_VERIF_SCOPE ((ucisScopeMaskTypeT)(UCIS_COV_SCOPE |\
                                               UCIS_ASSERT |\
                                               UCIS_GENERIC))


#define UCIS_HDL_SUBSCOPE ((ucisScopeMaskTypeT)(UCIS_PROCESS |\
                                                UCIS_BLOCK |\
                                                UCIS_FUNCTION |\
                                                UCIS_FORKJOIN |\
                                                UCIS_GENERATE |\
                                                UCIS_TASK |\
                                                UCIS_CLASS))

#define UCIS_HDL_INST_SCOPE ((ucisScopeMaskTypeT)(UCIS_INSTANCE |\
                                                  UCIS_PROGRAM |\
                                                  UCIS_PACKAGE |\
                                                  UCIS_INTERFACE))

#define UCIS_HDL_DU_SCOPE ((ucisScopeMaskTypeT)(UCIS_DU_ANY))

#define UCIS_HDL_SCOPE ((ucisScopeMaskTypeT)(UCIS_HDL_SUBSCOPE |\
                                             UCIS_HDL_INST_SCOPE |\
                                             UCIS_HDL_DU_SCOPE))

#define UCIS_NO_SCOPES  ((ucisScopeMaskTypeT)INT64_ZERO)
#define UCIS_ALL_SCOPES ((ucisScopeMaskTypeT)INT64_NEG1)

typedef unsigned int ucisFlagsT;

/*
 *  Flags for scope data
 *  32-bits are allocated to flags.
 *  Flags are partitioned into four groups: GENERAL, TYPED, MARK and USER
 *  The general flags apply to all scopes
 *  The type-qualified flags have a meaning that can only be understood
 *  with respect to the scope type
 *  A single mark flag is reserved for temporary (volatile) use only
 *  The user flags are reserved for user applications
 */

#define UCIS_SCOPEMASK_GENERAL     0x0000FFFF;  /* 16 flags for general use   */
#define UCIS_SCOPEMASK_TYPED       0x07FF0000;  /* 11 flags for typed use     */
#define UCIS_SCOPEMASK_MARK        0x08000000;  /* 1 flag for mark (temporary) use */
#define UCIS_SCOPEMASK_USER        0xF0000000;  /* 4 flags for user extension */

/* General flags 0x0000FFFF (apply to all scope types) */
#define UCIS_INST_ONCE             0x00000001   /* An instance is instantiated only
                                                   once; code coverage is stored only in
                                                     the instance */
#define UCIS_ENABLED_STMT          0x00000002   /* statement coverage collected for
scope */
#define UCIS_ENABLED_BRANCH        0x00000004    /* branch coverage coverage collected
                                                    for scope */
#define UCIS_ENABLED_COND          0x00000008    /* condition coverage coverage
                                                    collected for scope */
#define UCIS_ENABLED_EXPR          0x00000010    /* expression coverage coverage
```

```
                                                collected for scope */
#define UCIS_ENABLED_FSM           0x00000020   /* FSM coverage coverage collected for
                                                   scope */
#define UCIS_ENABLED_TOGGLE        0x00000040   /* toggle coverage coverage collected
                                                   for scope */

#define UCIS_SCOPE_UNDER_DU        0x00000100   /* is scope under a design unit? */
#define UCIS_SCOPE_EXCLUDED        0x00000200
#define UCIS_SCOPE_PRAGMA_EXCLUDED 0x00000400
#define UCIS_SCOPE_PRAGMA_CLEARED  0x00000800
#define UCIS_SCOPE_SPECIALIZED     0x00001000   /* Specialized scope may have data
                                                   restrictions */
#define UCIS_UOR_SAFE_SCOPE        0x00002000   /* Scope construction is UOR
                                                   compliant*/
#define UCIS_UOR_SAFE_SCOPE_ALLCOVERS0x00004000 /* Scope child coveritems are all
                                                    UOR compliant */

/* Type-qualified flags x07FF0000 - flag locations may be reused for non-intersecting
type sets */
#define UCIS_IS_TOP_NODE           0x00010000   /* UCIS_TOGGLE for top level toggle
                                                  node */
#define UCIS_IS_IMMEDIATE_ASSERT   0x00010000   /* UCIS_ASSERTION for SV immediate
                                                   assertions */
#define UCIS_SCOPE_CVG_AUTO        0x00010000   /* UCIS_COVERPOINT, UCIS_CROSS auto bin
                                                   */
#define UCIS_SCOPE_CVG_SCALAR      0x00020000   /* UCIS_COVERPOINT, UCIS_CROSS SV scalar
                                                   bin */
#define UCIS_SCOPE_CVG_VECTOR      0x00040000   /* UCIS_COVERPOINT, UCIS_CROSS SV vector
                                                   bin */
#define UCIS_SCOPE_CVG_TRANSITION  0x00080000   /* UCIS_COVERPOINT, UCIS_CROSS
                                                   transition bin */

#define UCIS_SCOPE_IFF_EXISTS      0x00100000   /* UCIS_COVERGROUP, UCIS_COVERINSTANCE
                                                  has 'iff' clause */
#define UCIS_SCOPE_SAMPLE_TRUE     0x00200000   /* UCIS_COVERGROUP, UCIS_COVERINSTANCE
                                                  has runtime samples */


#define UCIS_ENABLED_BLOCK         0x00800000   /* Block coverage collected for scope */
#define UCIS_SCOPE_BLOCK_ISBRANCH  0x01000000   /* UCIS_BBLOCK scope contributes to
                                                  both Block and Branch coverage */


#define UCIS_SCOPE_EXPR_ISHIERARCHICAL 0x02000000  /* UCIS_EXPR, UCIS_COND represented
                                                      hierarchically */

/* The temporary mark flag */
#define UCIS_SCOPEFLAG_MARK        0x08000000   /* flag for temporary mark */

/* The reserved flags */
#define UCIS_SCOPE_INTERNAL        0xF0000000

/*
 *  ucisSourceT
 *  Enumerated type to encode the source type of a scope, if needed.  Note that
 *  scope type can have an effect on how the system regards escaped identifiers
 *  within design hierarchy.
 */
typedef enum {
    UCIS_VHDL,
    UCIS_VLOG,          /* Verilog */
    UCIS_SV,            /* SystemVerilog */
    UCIS_SYSTEMC,
    UCIS_PSL_VHDL,      /* assert/cover in PSL VHDL */
    UCIS_PSL_VLOG,      /* assert/cover in PSL Verilog */
    UCIS_PSL_SV,        /* assert/cover in PSL SystemVerilog */
    UCIS_PSL_SYSTEMC,   /* assert/cover in PSL SystemC */
    UCIS_E,
    UCIS_VERA,
    UCIS_NONE,          /* if not important */
    UCIS_OTHER,         /* to refer to user-defined attributes */
    UCIS_SOURCE_ERROR   /* for error cases */
}   ucisSourceT;
```

```
/*
 *  ucis_CreateScope()
 *  Create the given scope underneath the given parent scope; if parent is
 *  NULL, create as root scope.
 *  Returns scope pointer or NULL if error.
 *  Note: use ucis_CreateInstance() for UCIS_INSTANCE scopes.
 *  Note: in write streaming mode, "name" is not copied; it needs to be
 *  preserved unchanged until the next ucis_WriteStream* call or the next
 *  ucis_Create* call.
 */
ucisScopeT
ucis_CreateScope(
    ucisT               db,
    ucisScopeT          parent,
    const char*         name,
    ucisSourceInfoT*    srcinfo,
    int                 weight,         /* negative for none */
    ucisSourceT         source,
    ucisScopeTypeT      type,           /* type of scope to create */
    ucisFlagsT          flags);
/*
 * Specialized constructors
 *
 * Specialized constructors are used to ensure data coherence by
 * enforcing the collection of necessary data in an atomic operation,
 * at construction time.
 *
 * Specialized constructors add constraints to database object *data*
 * but still operate within the database *schema*.
 *
 * The data consistency enforced by the specialization may be a
 * pre-requisite to certain optimizations, and these optimizations may be
 * vendor-specific.
 *
 * For example, a specialized scope may have a forced bin collection shape
 * and default attributes populated from a default list of values.
 * It may also require that certain attributes be present.
 * The intent is to support common data constructs that depend on
 * certain assumptions.
 *
 * Specialized constructors may enforce attribute and bin shape data, but
 * they do not imply the presence of data that is not accessible to
 * the API routines
 *
 * The UCIS_SCOPE_SPECIALIZED is set on a scope if a specialized constructor
 * was used.
 *
 * The specialized scope constructors are:
 *
 * ucis_CreateInstance() - enforce the rule that an instance has a DU
 * ucis_CreateInstanceByName() - enforce the rule that an instance has a DU
 * ucis_CreateCross()    - enforce the cross relationship to coverpoints
 * ucis_CreateCrossByName() - enforce the cross relationship to coverpoints
 * ucis_CreateToggle()   - add properties to identify the basic toggle metrics
 */

/*
 * ucis_CreateInstance() specialized constructor
 *
 * This constructor enforces the addition of the DU relationship to an
 * instance scope
 * It is used to construct scopes of types:
 * - UCIS_INSTANCE which must have a du_scope of one of the UCIS_DU_* types
 * - UCIS_COVERINSTANCE, which must have a du_scope of UCIS_COVERGROUP type
 */

ucisScopeT
ucis_CreateInstance(
    ucisT               db,
    ucisScopeT          parent,
    const char*         name,
    ucisSourceInfoT*    fileinfo,
    int                 weight,         /* negative for none */
    ucisSourceT         source,
```

```
    ucisScopeTypeT      type,           /* UCIS_INSTANCE, UCIS_COVERINSTANCE
                                           type of scope to create */
    ucisScopeT          du_scope,       /* if type==UCIS_INSTANCE, this is a
                                           scope of type UCIS_DU_*; if
                                           type==UCIS_COVERINSTANCE, this is a
                                           scope of type UCIS_COVERGROUP */
    ucisFlagsT          flags);

/*
 * ucis_CreateInstanceByName() specialized constructor
 *
 * This is the write-streaming version of ucis_CreateInstance() where the
 * design unit or covergroup is identified by name (because in-memory pointer
 * identification is not possible). Note that this routine marshalls data for
 * write-streaming but it is not flushed until a ucis_WriteStream* or ucis_Create*
 * call is made
 * The strings pointed to by "name" and "du_name" must be retained
 * until then.
 *
 * When called in write-streaming mode, the parent is ignored, however the routine
 * may also be used in in-memory mode. In this case, the du_name is converted
 * to a ucisScopeT and the routine behaves as ucis_CreateInstance().
 */

ucisScopeT
ucis_CreateInstanceByName(
    ucisT               db,
    ucisScopeT          parent,
    const char*         name,
    ucisSourceInfoT*    fileinfo,
    int                 weight,
    ucisSourceT         source,
    ucisScopeTypeT      type,           /* UCIS_INSTANCE, UCIS_COVERINSTANCE
                                           type of scope to create */
    char*               du_name,        /* name for instance's design unit, or
                                           coverinstance's covergroup type */
    int                 flags);

/*
 * ucis_CreateCross() specialized constructor
 *
 * This constructor uses the data from existing parent covergroup or
 * coverinstance to infer the basic cross scope construction
 * The parental coverpoints must exist.
 */

ucisScopeT
ucis_CreateCross(
    ucisT               db,
    ucisScopeT          parent,         /* covergroup or cover instance */
    const char*         name,
    ucisSourceInfoT*    fileinfo,
    int                 weight,
    ucisSourceT         source,
    int                 num_points,     /* number of crossed coverpoints */
    ucisScopeT*         points);        /* array of coverpoint scopes */

/*
 * ucis_CreateCrossByName() specialized constructor
 *
 * This is the write-streaming version of ucis_CreateCross() where the
 * crossed coverpoints are identified by name (because in-memory pointer
 * identification is not possible)
 * Note that this routine marshalls data for
 * write-streaming but it is not flushed until a ucis_WriteStream* or ucis_Create*
 * call is made
 * The strings pointed to by "name" and "du_name" must be retained
 * until then.
 *
 */

ucisScopeT
ucis_CreateCrossByName(
    ucisT               db,
```

```
    ucisScopeT          parent,         /* covergroup or cover instance */
    const char*         name,
    ucisSourceInfoT*    fileinfo,
    int                 weight,
    ucisSourceT         source,
    int                 num_points,     /* number of crossed coverpoints */
    char**              point_names);   /* array of coverpoint names */

/*
 * ucis_CreateToggle() specialized constructor
 *
 * This constructor enforces type-dependent bin shapes and associated attributes
 * onto a toggle scope.
 *
 * The bins under this scope must be created after the scope has been created.
 *
 * The canonical_name attribute is used to associate the toggle name, which
 * is a local name, with a global net name (similar to the Verilog simnet concept)
 * The canonical_name is recovered with ucis_GetStringProperty() for the
 * UCIS_STR_TOGGLE_CANON_NAME property
 *
 * The toggle_metric value is used to identify the specific shape specialization
 * The toggle_metric is recovered with ucis_GetIntProperty() for the
 * UCIS_INT_TOGGLE_METRIC property
 *
 * The toggle_type value is used to identify whether the toggling object is a
 * net or register.
 * The toggle_type is recovered with ucis_GetIntProperty() for the
 * UCIS_INT_TOGGLE_TYPE property
 *
 * The toggle_dir value is used to identify the toggle direction for a port toggle.
 * The toggle_dir is recovered with ucis_GetIntProperty() for the
 * UCIS_INT_TOGGLE_DIR property
 */

typedef enum {                          /* Implicit toggle local bins metric is:  */
    UCIS_TOGGLE_METRIC_NOBINS = 1,      /* Toggle scope has no local bins         */
    UCIS_TOGGLE_METRIC_ENUM,            /* UCIS:ENUM                              */
    UCIS_TOGGLE_METRIC_TRANSITION,      /* UCIS:TRANSITION                        */
    UCIS_TOGGLE_METRIC_2STOGGLE,        /* UCIS:2STOGGLE                          */
    UCIS_TOGGLE_METRIC_ZTOGGLE,         /* UCIS:ZTOGGLE                           */
    UCIS_TOGGLE_METRIC_XTOGGLE          /* UCIS:XTOGGLE                           */
} ucisToggleMetricT;

typedef enum {
    UCIS_TOGGLE_TYPE_NET = 1,
    UCIS_TOGGLE_TYPE_REG = 2
} ucisToggleTypeT;

typedef enum {
    UCIS_TOGGLE_DIR_INTERNAL = 1,   /* non-port: internal wire or variable */
    UCIS_TOGGLE_DIR_IN,             /* input port */
    UCIS_TOGGLE_DIR_OUT,            /* output port */
    UCIS_TOGGLE_DIR_INOUT           /* inout port */
} ucisToggleDirT;

ucisScopeT
ucis_CreateToggle(
    ucisT               db,
    ucisScopeT          parent,         /* toggle will be created in this scope */
    const char*         name,           /* name of toggle object                */
    const char*         canonical_name, /* canonical_name of toggle object      */
    ucisFlagsT          flags,
    ucisToggleMetricT   toggle_metric,
    ucisToggleTypeT     toggle_type,
    ucisToggleDirT      toggle_dir);

/*
 *  Utilities for parsing and composing design unit scope names.  These are of
 *  the form "library.primary(secondary)#instance_num"
 *
 *  Note: these utilities each employ a static dynamic string (one for the
 *  "Compose" function, one for the "Parse" function).  That means that values
 *  are only valid until the next call to the respective function; if you need
```

```
 *  to hold the memory across separate calls you must copy it.
 */
const char*                              /* Return value is in temp storage */
ucis_ComposeDUName(
    const char*         library_name,  /* input names */
    const char*         primary_name,
    const char*         secondary_name);

void
ucis_ParseDUName(
    const char*         du_name,       /* input to function */
    const char**        library_name,  /* output names */
    const char**        primary_name,
    const char**        secondary_name);

int
ucis_RemoveScope(
    ucisT               db,     /* database context */
    ucisScopeT          scope); /* scope to remove */

/*
 *  ucis_GetScopeType()
 *  Get type of scope.
 *  Returns UCIS_SCOPE_ERROR if error.
 */
ucisScopeTypeT
ucis_GetScopeType(
    ucisT               db,
    ucisScopeT          scope);

/*
 *  ucis_GetObjType (alias ucis_GetHistoryKind)
 *
 *  Returns UCIS_SCOPE_ERROR if error.
 *  Returns UCIS_HISTORYNODE_TEST when obj is a test data record.
 *  Returns UCIS_HISTORYNODE_MERGE when obj is a merge record.
 *  Otherwise, returns the scope type ucisScopeTypeT:
 *      [See ucisScopeTypeT ucis_GetScopeType(ucisT db, ucisScopeT scope)]
 *
 *  This is a polymorphic function for acquiring an object type.
 *  History node types, unlike scope and coveritem types, are not one-hot,
 *  and should therefore not be used as a mask
 */
ucisObjTypeT
ucis_GetObjType(
    ucisT           db,
    ucisObjT        obj);

/*
 *  ucis_GetScopeFlags()
 *  Get scope flags.
 */
ucisFlagsT
ucis_GetScopeFlags(
    ucisT               db,
    ucisScopeT          scope);

/*
 *  ucis_SetScopeFlags()
 *  Set scope flags.
 */
void
ucis_SetScopeFlags(
    ucisT               db,
    ucisScopeT          scope,
    ucisFlagsT          flags);

/*
 *  ucis_GetScopeFlag()
 *  Fancier interface for getting a single flag bit.
 *  Return 1 if specified scope's flag bit matches the given mask.
 */
int
ucis_GetScopeFlag(
```

```
    ucisT               db,
    ucisScopeT          scope,
    ucisFlagsT          mask);

/*
 *  ucis_SetScopeFlag()
 *  Set bits in the scope's flag field with respect to the given mask --
 *  set all bits to 0 or 1.
 */
void
ucis_SetScopeFlag(
    ucisT               db,
    ucisScopeT          scope,
    ucisFlagsT          mask,
    int                 bitvalue);  /* 0 or 1 only */

/*
 *  ucis_GetScopeSourceInfo()
 *  Gets source information (file/line/token) for the given scope.
 *  Note: does not apply to toggle nodes.
 */
int
ucis_GetScopeSourceInfo(
    ucisT               db,
    ucisScopeT          scope,
    ucisSourceInfoT*    sourceinfo);

/*
 *  ucis_SetScopeSourceInfo()
 *  Sets source information (file/line/token) for the given scope.
 *  Returns 0 on success, non-zero on failure.
 */
int
ucis_SetScopeSourceInfo(
    ucisT               db,
    ucisScopeT          scope,
    ucisSourceInfoT*    sourceinfo);

/*
 *  ucis_GetIthCrossedCvp()
 *  Get crossed coverpoint of scope specified by the index, if scope is a
 *  cross scope.
 *  Returns 0 if success, non-zero if error.
 */
int
ucis_GetIthCrossedCvp(
    ucisT               db,
    ucisScopeT          scope,
    int                 index,
    ucisScopeT*         point_scope);

/*
 *  ucis_MatchDU()
 *  Given a design unit name, get the design unit scope in the database.
 *  Returns NULL if no match is found.
 */
ucisScopeT
ucis_MatchDU(
    ucisT               db,
    const char*         name);


/*----------------------------------------------------------------------------
 *  Creating coveritems (bins, with an associated count.)
 *--------------------------------------------------------------------------*/

/* One-hot bits for ucisCoverTypeT: */
#define UCIS_CVGBIN          /* For SV Covergroups: */ \
                             INT64_LITERAL(0x0000000000000001)
#define UCIS_COVERBIN        /* For cover directives- pass: */ \
                             INT64_LITERAL(0x0000000000000002)
#define UCIS_ASSERTBIN       /* For assert directives- fail: */ \
                             INT64_LITERAL(0x0000000000000004)
#define UCIS_STMTBIN         /* For Code coverage(Statement): */ \
```

```
                              INT64_LITERAL(0x0000000000000020)
#define UCIS_BRANCHBIN       /* For Code coverage(Branch): */ \
                              INT64_LITERAL(0x0000000000000040)
#define UCIS_EXPRBIN          /* For Code coverage(Expression): */ \
                              INT64_LITERAL(0x0000000000000080)
#define UCIS_CONDBIN          /* For Code coverage(Condition): */ \
                              INT64_LITERAL(0x0000000000000100)
#define UCIS_TOGGLEBIN        /* For Code coverage(Toggle): */ \
                              INT64_LITERAL(0x0000000000000200)
#define UCIS_PASSBIN          /* For assert directives- pass count: */ \
                              INT64_LITERAL(0x0000000000000400)
#define UCIS_FSMBIN           /* For FSM coverage: */ \
                              INT64_LITERAL(0x0000000000000800)
#define UCIS_USERBIN          /* User-defined coverage: */ \
                              INT64_LITERAL(0x0000000000001000)
#define UCIS_GENERICBIN       UCIS_USERBIN
#define UCIS_COUNT            /* user-defined count, not in coverage: */ \
                              INT64_LITERAL(0x0000000000002000)
#define UCIS_FAILBIN          /* For cover directives- fail count: */ \
                              INT64_LITERAL(0x0000000000004000)
#define UCIS_VACUOUSBIN       /* For assert- vacuous pass count: */ \
                              INT64_LITERAL(0x0000000000008000)
#define UCIS_DISABLEDBIN      /* For assert- disabled count: */ \
                              INT64_LITERAL(0x0000000000010000)
#define UCIS_ATTEMPTBIN       /* For assert- attempt count: */ \
                              INT64_LITERAL(0x0000000000020000)
#define UCIS_ACTIVEBIN        /* For assert- active thread count: */ \
                              INT64_LITERAL(0x0000000000040000)
#define UCIS_IGNOREBIN        /* For SV Covergroups: */ \
                              INT64_LITERAL(0x0000000000080000)
#define UCIS_ILLEGALBIN       /* For SV Covergroups: */ \
                              INT64_LITERAL(0x0000000000100000)
#define UCIS_DEFAULTBIN       /* For SV Covergroups: */ \
                              INT64_LITERAL(0x0000000000200000)
#define UCIS_PEAKACTIVEBIN    /* For assert- peak active thread count: */ \
                              INT64_LITERAL(0x0000000000400000)
#define UCIS_BLOCKBIN         /* For Code coverage(Block): */ \
                              INT64_LITERAL(0x0000000001000000)
#define UCIS_USERBITS         /* For user-defined coverage: */ \
                              INT64_LITERAL(0x00000000FE000000)
#define UCIS_RESERVEDBIN      INT64_LITERAL(0xFF00000000000000)

/* Coverage item types */

#define UCIS_COVERGROUPBINS \
            ((ucisCoverMaskTypeT)(UCIS_CVGBIN | \
                                  UCIS_IGNOREBIN | \
                                  UCIS_ILLEGALBIN | \
                                  UCIS_DEFAULTBIN))

#define UCIS_FUNC_COV \
            ((ucisCoverMaskTypeT)(UCIS_COVERGROUPBINS | \
                                  UCIS_COVERBIN | \
                                  UCIS_SCBIN))

#define UCIS_CODE_COV \
            ((ucisCoverMaskTypeT)(UCIS_STMTBIN | \
                                  UCIS_BRANCHBIN | \
                                  UCIS_EXPRBIN | \
                                  UCIS_CONDBIN | \
                                  UCIS_TOGGLEBIN | \
                                  UCIS_FSMBIN |\
                                  UCIS_BLOCKBIN))

#define UCIS_ASSERTIONBINS \
            ((ucisCoverMaskTypeT)(UCIS_ASSERTBIN |\
                                  UCIS_PASSBIN |\
                                  UCIS_VACUOUSBIN |\
                                  UCIS_DISABLEDBIN |\
                                  UCIS_ATTEMPTBIN |\
                                  UCIS_ACTIVEBIN |\
                                  UCIS_PEAKACTIVEBIN))

#define UCIS_COVERDIRECTIVEBINS \
```

```
            ((ucisCoverMaskTypeT)(UCIS_COVERBIN |\
                                  UCIS_FAILBIN))
#define UCIS_NO_BINS  ((ucisCoverMaskTypeT)INT64_ZERO)
#define UCIS_ALL_BINS ((ucisCoverMaskTypeT)INT64_NEG1)

/*--------------------------------------------------------------------------
 *  Creating and manipulating coveritems
 *--------------------------------------------------------------------------*/

/*
 *  Flags for coveritem data
 *  32-bits are allocated to flags.
 *  Flags are partitioned into four groups: GENERAL, TYPED, MARK and USER
 *  The general flags apply to all scopes
 *  The type-determined flags have a meaning that can only be understood
 *  with respect to the scope type
 *  The user flags are reserved for user applications
 */

#define UCIS_COVERITEMMASK_GENERAL   0x0000FFFF;  /* 16 flags for general use   */
#define UCIS_COVERITEMMASK_TYPED     0x07FF0000;  /* 11 flags for typed use     */
#define UCIS_COVERITEMMASK_MARK      0x08000000;  /* 1 flag for mark (temporary) use */
#define UCIS_COVERITEMMASK_USER      0xF0000000;  /* 4 flags for user extension */

/* General flags 0x0000FFFF (apply to all coveritem types */
#define UCIS_IS_32BIT        0x00000001  /* data is 32 bits */
#define UCIS_IS_64BIT        0x00000002  /* data is 64 bits */
#define UCIS_IS_VECTOR       0x00000004  /* data is a vector */
#define UCIS_HAS_GOAL        0x00000008  /* goal included */
#define UCIS_HAS_WEIGHT      0x00000010  /* weight included */

#define UCIS_CLEAR_PRAGMA    0x00004000
#define UCIS_EXCLUDE_PRAGMA  0x00000020  /* excluded by pragma  */
#define UCIS_EXCLUDE_FILE    0x00000040  /* excluded by file; does not
                                            count in total coverage */
#define UCIS_EXCLUDE_INST    0x00000080  /* for instance-specific exclusions */
#define UCIS_EXCLUDE_AUTO    0x00000100  /* for automatic exclusions */

#define UCIS_ENABLED         0x00000200  /* generic enabled flag; if disabled,
                                            still counts in total coverage */
#define UCIS_HAS_LIMIT       0x00000400  /* for limiting counts */
#define UCIS_HAS_COUNT       0x00000800  /* has count in ucisCoverDataValueT? */
#define UCIS_IS_COVERED      0x00001000  /* set if object is covered */
#define UCIS_UOR_SAFE_COVERITEM        0x00002000   /* Coveritem construction is UOR
compliant */

/* Type-qualified flags 0x07FF0000 - flag locations may be reused for non-intersecting
type sets */
#define UCIS_HAS_ACTION      0x00010000  /* UCIS_ASSERTBIN */
#define UCIS_IS_TLW_ENABLED  0x00020000  /* UCIS_ASSERTBIN */
#define UCIS_LOG_ON          0x00040000  /* UCIS_COVERBIN, UCIS_ASSERTBIN */
#define UCIS_IS_EOS_NOTE     0x00080000  /* UCIS_COVERBIN, UCIS_ASSERTBIN */

#define UCIS_IS_FSM_RESET    0x00010000  /* UCIS_FSMBIN */
#define UCIS_IS_FSM_TRAN     0x00020000  /* UCIS_FSMBIN */
#define UCIS_IS_BR_ELSE      0x00010000  /* UCIS_BRANCHBIN */

#define UCIS_BIN_IFF_EXISTS  0x00010000  /* UCIS_CVGBIN UCIS_IGNOREBIN UCIS_ILLEGALBIN
                                            UCIS_DEFAULTBIN */
#define UCIS_BIN_SAMPLE_TRUE 0x00020000  /* UCIS_CVGBIN UCIS_IGNOREBIN UCIS_ILLEGALBIN
                                             UCIS_DEFAULTBIN */

#define UCIS_IS_CROSSAUTO    0x00040000  /* UCIS_CROSS */

/* The temporary mark flag */
#define UCIS_COVERFLAG_MARK  0x08000000  /* flag for temporary mark */

/* The reserved user flags */
#define UCIS_USERFLAGS       0xF0000000  /* reserved for user flags */

#define UCIS_FLAG_MASK       0xFFFFFFFF

#define UCIS_EXCLUDED        (UCIS_EXCLUDE_FILE | UCIS_EXCLUDE_PRAGMA |
UCIS_EXCLUDE_INST | UCIS_EXCLUDE_AUTO)
```

```
/*
 *  Type representing coveritem data.
 */
typedef union {
    uint64_t            int64;      /* if UCIS_IS_64BIT */
    uint32_t            int32;      /* if UCIS_IS_32BIT */
    unsigned char*      bytevector; /* if UCIS_IS_VECTOR */
} ucisCoverDataValueT;

typedef struct {
    ucisCoverTypeT      type;       /* type of coveritem */
    ucisFlagsT          flags;      /* as above, validity of fields below */
    ucisCoverDataValueT data;
    /*
     *  This "goal" value is used to determine whether an individual bin is
     *  covered; it corresponds to "at_least" in a covergroup:
     */
    int                 goal;       /* if UCIS_HAS_GOAL */
    int                 weight;     /* if UCIS_HAS_WEIGHT */
    int                 limit;      /* if UCIS_HAS_LIMIT */
    int                 bitlen;     /* length of data.bytevector in bits,
                                     * extra bits are lower order bits of the
                                     * last byte in the byte vector
                                     */
} ucisCoverDataT;

/*
 *  ucis_CreateNextCover()
 *  Create the next coveritem in the given scope.
 *  Returns the index number of the created coveritem, -1 if error.
 *  Note: in write streaming mode, "name" is not copied; it needs to be
 *  preserved unchanged until the next ucis_WriteStream* call or the next
 *  ucis_Create* call.
 */
int
ucis_CreateNextCover(
    ucisT               db,
    ucisScopeT          parent,     /* coveritem will be created in this scope */
    const char*         name,       /* name of coveritem, can be NULL */
    ucisCoverDataT*     data,       /* associated data for coverage */
    ucisSourceInfoT*    sourceinfo);

/*
 *  ucis_RemoveCover()
 *  This function removes the given cover from its parent.
 *  There is no effect of this function in streaming modes.
 *  A successful operation will return 0, and -1 for error.
 *  Note: Coveritems can not be removed from scopes of type UCIS_ASSERT,
 *  need to remove the whole scope instead.
 *  Similarly, coveritems from scopes of type UCIS_TOGGLE which has toggle kind
 *  UCIS_TOGGLE_SCALAR, or UCIS_TOGGLE_SCALAR_EXT, or UCIS_TOGGLE_REG_SCALAR,
 *  or UCIS_TOGGLE_REG_SCALAR_EXT cannot be removed. The scope needs to be
 *  removed in this case too.
 *  Also, this function should be used carefully if it is
 *  used during iteration of coveritems using any coveritem
 *  iteration API, otherwise the iteration may not be complete.
 */
int
ucis_RemoveCover(
    ucisT               db,
    ucisScopeT          parent,
    int                 coverindex);

/*
 *  ucis_GetCoverFlags()
 *  Get coveritem's flag.
 */
ucisFlagsT
ucis_GetCoverFlags(
    ucisT               db,
    ucisScopeT          parent,      /* parent scope of coveritem */
    int                 coverindex); /* index of coveritem in parent */
```

```
/*
 *  ucis_GetCoverFlag()
 *  Return 1 if specified coveritem's flag bit matches the given mask.
 */
int
ucis_GetCoverFlag(
    ucisT               db,
    ucisScopeT          parent,     /* parent scope of coveritem */
    int                 coverindex, /* index of coveritem in parent */
    ucisFlagsT          mask);

/*
 *  ucis_SetCoverFlag()
 *  Set bits in the coveritem's flag field with respect to the given mask --
 *  set all bits to 0 or 1.
 */
void
ucis_SetCoverFlag(
    ucisT               db,
    ucisScopeT          parent,     /* parent scope of coveritem */
    int                 coverindex, /* index of coveritem in parent */
    ucisFlagsT          mask,
    int                 bitvalue);  /* 0 or 1 only */

/*
 *  ucis_GetCoverData()
 *  Get all the data for a coverage item, returns 0 for success, and non-zero
 *  for any error.  It is the user's responsibility to save the returned data,
 *  the next call to this function will invalidate the returned data.
 *  Note: any of the data arguments may be NULL, in which case data is not
 *  retrieved.
 */
int
ucis_GetCoverData(
    ucisT               db,
    ucisScopeT          parent,     /* parent scope of coveritem */
    int                 coverindex, /* index of coveritem in parent */
    char**              name,
    ucisCoverDataT*     data,
    ucisSourceInfoT*    sourceinfo);

/*
 *  ucis_SetCoverData()
 *  Set the data for a coverage item, returns 0 for success, and non-zero
 *  for any error.  It is the user's responsibility to make all the data
 *  fields valid.
 */
int
ucis_SetCoverData(
    ucisT               db,
    ucisScopeT          parent,     /* parent scope of coveritem */
    int                 coverindex, /* index of coveritem in parent */
    ucisCoverDataT*     data);

/*
 *  ucis_IncrementCover()
 *  Increment the data count for a coverage data item, if not a vector item.
 */
int
ucis_IncrementCover(
    ucisT               db,
    ucisScopeT          parent,     /* parent scope of coveritem */
    int                 coverindex, /* index of coveritem in parent */
    int64_t             increment); /* added to current count */

/*----------------------------------------------------------------------------
 * Test traceability support
 * Test traceability is based on a data handle type that encapsulates a list
 * of history node records in an opaque handle type - ucisHistoryNodeListT
 *
 * The history node list is a data type in its own right but lists are not stored
 * in the database as standalone objects, only in association with coveritems.
 *
 * A history node list may be iterated with ucis_HistoryNodeListIterate
```

```
 * and ucis_HistoryScan.
 * Note though that the iterator object is distinct from the list.
 * List and iterator memory management must both be individually considered
 * when using history node list iterators.
 *
 * A list of history nodes may be associated with any number of coveritems in
 * the hierarchy, for example to express the relationship between tests and
 * the target coveritem(s) that were incremented when the test was run.
 *
 * If the application constructs a list, and associates the list with one or
 * more coveritems, the original list must be freed when the associations have
 * been made.
 *
 * Each association between a list and a coveritem may be labeled with an
 * integral 'association type' to indicate the semantic of the association.
 * For example the UCIS_ASSOC_TESTHIT semantic
 * is pre-defined to represent the association between a list of
 * tests and the coveritems that were incremented by the test
 *
 *------------------------------------------------------------------------*/

#define UCIS_ASSOC_TESTHIT                0
#define UCIS_ASSOC_FORMALLY_UNREACHABLE   1

typedef void * ucisHistoryNodeListT; /* conceptually a list of history nodes */
typedef int    ucisAssociationTypeT; /* see UCIS_ASSOC #defines above */

/*
 * History node list management
 * - list construction
 * - add and delete history nodes
 * - list deletion (to free memory associated with list, not the history nodes)
 * - list query (by iteration).
 * - list association to coveritem(s)
 *
 * The model for this functionality is that history node lists are
 * data-marshalling constructs created by the application, or returned
 * by ucis_GetHistoryNodeListAssoc()
 *
 * The list itself is required to be non-duplicative and non-ordered
 * If a list-element is re-added, it is not an error, although nothing is added
 * The list key (to determine duplication) is the ucisHistoryNodeT handle.
 * If a non-existent element is removed, it is not an error, nothing is removed
 * No list ordering is guaranteed or implied. Applications must not rely
 * on list order.
 *
 * An existing list may be associated with one or more coveritems
 * (one association is made per call to ucis_SetHistoryNodeListAssoc()
 * but the call may be repeated on multiple coveritems using the same list)
 *
 * Events that change or invalidate either the history nodes on the list or
 * the coveritems, may also invalidate the lists and the iterations on
 * them.
 *
 * The memory model for this process is that the call to
 * ucis_SetHistoryNodeListAssoc() causes the replacement of any pre-existing
 * association.
 * ucis_SetHistoryNodeListAssoc() causes database *copies* of the list data
 * to be constructed.
 * A call to ucis_GetHistoryNodeListAssoc() returns a read-only list, valid
 * until another call to ucis_GetHistoryNodeListAssoc() or another
 * invalidating event.
 * An empty list and a NULL pointer are both representations of zero
 * history nodes associated with a coveritem. These are equivalent when
 * presented to ucis_SetHistoryNodeListAssoc() (both will delete any existing
 * association), but ucis_GetHistoryNodeListAssoc() returns a NULL list pointer
 * in both cases. That is, the database does not distinguish between an
 * empty list and the absence of a list.
 *
 * Consequences of these design assumptions are:
 * - the application copy of a list created by ucis_CreateHistoryNodeList() and
 *   used in a ucis_SetHistoryNodeListAssoc() call can (though need not)
 *   immediately be freed if desired
 * - Updating an existing list associated with a coveritem is a
```

```
 *   read-duplicate-modify-replace operation (the list associated with a coveritem
 *   cannot be directly edited)
 * - Either an empty list or a NULL list pointer deletes the existing association
 */

ucisHistoryNodeListT
ucis_CreateHistoryNodeList(ucisT db);

/*
 * ucis_FreeHistoryNodeList frees only the memory associated with the list,
 * it does not affect the history nodes on the list.
 *
 * This routine must only be used on a list created by the user with
 * ucis_CreateHistoryNodeList(). Lists returned from the kernel should
 * not be freed. Generally, lists returned from the kernel are valid only
 * until the next list-management call, or the database is closed, whichever
 * is first. Lists are also invalidated by coveritem removal events.
 * Any iterators on a list that is invalidated by one of these
 * events, also immediately become invalid.
 */
int
ucis_FreeHistoryNodeList(ucisT db,
                         ucisHistoryNodeListT historynode_list);

int
ucis_AddToHistoryNodeList(ucisT db,
                          ucisHistoryNodeListT historynode_list,
                          ucisHistoryNodeT historynode);

int
ucis_RemoveFromHistoryNodeList(ucisT db,
                               ucisHistoryNodeListT historynode_list,
                               ucisHistoryNodeT historynode);

ucisIteratorT
ucis_HistoryNodeListIterate(ucisT db,
                            ucisHistoryNodeListT historynode_list);


/*
 * Association of history node list with coveritem - set and query
 */

int
ucis_SetHistoryNodeListAssoc(ucisT db,
                     ucisScopeT scope,
                     int coverindex,
                     ucisHistoryNodeListT historynode_list,
                     ucisAssociationTypeT assoc_type);

ucisHistoryNodeListT
ucis_GetHistoryNodeListAssoc(ucisT db,
                     ucisScopeT scope,
                     int coverindex,
                     ucisAssociationTypeT assoc_type);


/*----------------------------------------------------------------------------
 *  Summary coverage statistics.
 *
 *  This interface allows quick access to aggregated coverage and statistics
 *  for different kinds of coverage, and some overall statistics for the
 *  database.
 *--------------------------------------------------------------------------*/

#define UCIS_CVG_INST    0x00000001 /* same as $get_coverage in SystemVerilog */
#define UCIS_CVG_DU      0x00000002 /* Covergroup coverage, per design unit */
#define UCIS_BLOCK_INST  0x00000004 /* Block coverage, per design instance */
#define UCIS_BLOCK_DU    0x00000008 /* Block coverage, per design unit */
#define UCIS_STMT_INST   0x00000010 /* Statement coverage, per design instance */
#define UCIS_STMT_DU     0x00000020 /* Statement coverage, per design unit */
#define UCIS_BRANCH_INST 0x00000040 /* Branch coverage, per design instance */
#define UCIS_BRANCH_DU   0x00000080 /* Branch coverage, per design unit */
#define UCIS_EXPR_INST   0x00000100 /* Expression coverage, per design instance */
```

```
#define UCIS_EXPR_DU     0x00000200 /* Expression coverage, per design unit */
#define UCIS_COND_INST   0x00000400 /* Condition coverage, per design instance */
#define UCIS_COND_DU     0x00000800 /* Condition coverage, per design unit */
#define UCIS_TOGGLE_INST 0x00001000 /* Toggle coverage, per design instance */
#define UCIS_TOGGLE_DU   0x00002000 /* Toggle coverage, per design unit */
#define UCIS_FSM_ST_INST 0x00004000 /* FSM state coverage, per design instance */
#define UCIS_FSM_ST_DU   0x00008000 /* FSM state coverage, per design unit */
#define UCIS_FSM_TR_INST 0x00010000 /* FSM transition coverage, per design instance */
#define UCIS_FSM_TR_DU   0x00020000 /* FSM transition coverage, per design unit */
#define UCIS_USER_INST   0x00040000 /* User-defined coverage, per design instance */
#define UCIS_USER_DU     0x00080000 /* User-defined coverage, per design unit */
#define UCIS_ASSERT_PASS_INST      0x00100000 /* Assertion directive passes, per design
                                                 instance */
#define UCIS_ASSERT_FAIL_INST      0x00200000 /* Assertion directive failures, per
                                                 design instance */
#define UCIS_ASSERT_VPASS_INST     0x00400000 /* Assertion directive vacuous passes,
                                                 per design instance */
#define UCIS_ASSERT_DISABLED_INST  0x00800000 /* Assertion directive disabled,per
                                                 design instance */
#define UCIS_ASSERT_ATTEMPTED_INST 0x01000000 /* Assertion directive attempted, per
                                                 design instance */
#define UCIS_ASSERT_ACTIVE_INST    0x02000000 /* Assertion directive active, per design
                                                 instance */
#define UCIS_ASSERT_PACTIVE_INST   0x04000000 /* Assertion directive peakactive, per
                                                 design instance */
#define UCIS_COVER_INST  0x08000000 /* Cover directive, per design instance*/
#define UCIS_COVER_DU    0x10000000 /* Cover directive, per design instance*/

/* ucisCoverageT stores values for a particular #define */
typedef struct {
    int     num_coveritems;/* total number of coveritems (bins) */
    int     num_covered;/* number of coveritems (bins) covered */
    float   coverage_pct;/* floating point coverage value, percentage */
    int64_t weighted_numerator;
    int64_t weighted_denominator;
} ucisCoverageT;

#define UCIS_SCORE_DEFAULT 0 /* vendor-specific default scoring algorithm */

float ucis_CoverageScore(
    ucisT db,
    ucisScopeT scope,
    int recursive,
    int scoring_algorithm,
    uint64_t metrics_mask,
    ucisCoverageSummaryT* data /* list only of types enabled in metrics_mask */
                               /* Ordering is lsb-first of metrics mask bits */
);

/*----------------------------------------------------------------------------
 *  Tags
 *
 *  A tag is a string associated with an object. An object may have multiple
 *  tags. The tag semantic is that things that share the same
 *  tag are associated together.
 *
 *  Tags may be added to scopes, coveritems, and history nodes.
 *  See the ucis_ObjectTagsIterate()/ucis_ObjectTagsScan() routines for obtaining all
 *  the tags an object has.
 *  See the ucis_TaggedObjIterate()/ucis_TaggedObjScan() routines for iterating
 *  objects that have been tagged with a given tag.
 *
 *----------------------------------------------------------------------------*/

/*
 *  ucis_AddObjTag()
 *  Add a tag to a given obj.
 *  Returns 0 with success, non-zero with error.
 *  Error includes null tag or tag with '\n' character.
 */
int
ucis_AddObjTag(
    ucisT               db,
    ucisObjT            obj, /* ucisScopeT or ucisHistoryNodeT */
```

```
    const char*         tag);

/*
 *  ucis_RemoveObjTag()
 *  Remove the given tag from the obj.
 *  Returns 0 with success, non-zero with error.
 */
int
ucis_RemoveObjTag(
    ucisT               db,
    ucisObjT            obj, /* ucisScopeT or ucisHistoryNodeT */
    const char*         tag);

/*----------------------------------------------------------------------------
 *  Formal
 *
 *----------------------------------------------------------------------------*/

typedef void* ucisFormalEnvT;

/*
 * Formal Results enum
 *
 * The following enum is used in the API functions to indicate a formal result
 * for an assertion:
 */
typedef enum {
    UCIS_FORMAL_NONE,         /* No formal info (default) */
    UCIS_FORMAL_FAILURE,      /* Fails */
    UCIS_FORMAL_PROOF,        /* Proven to never fail */
    UCIS_FORMAL_VACUOUS,      /* Assertion is vacuous as defined by the
                                 assertion language */
    UCIS_FORMAL_INCONCLUSIVE, /* Proof failed to complete */
    UCIS_FORMAL_ASSUMPTION,   /* Assertion is an assume */
    UCIS_FORMAL_CONFLICT      /* Data merge conflict */
} ucisFormalStatusT;

typedef struct ucisFormalToolInfoS {
    char* formal_tool;
    char* formal_tool_version;
    char* formal_tool_setup;
    char* formal_tool_db;
    char* formal_tool_rpt;
    char* formal_tool_log;
} ucisFormalToolInfoT;

#define UCIS_FORMAL_COVERAGE_CONTEXT_STIMULUS \
    "UCIS_FORMAL_COVERAGE_CONTEXT_STIMULUS"
#define UCIS_FORMAL_COVERAGE_CONTEXT_RESPONSE \
    "UCIS_FORMAL_COVERAGE_CONTEXT_REPONSE"
#define UCIS_FORMAL_COVERAGE_CONTEXT_TARGETED \
    "UCIS_FORMAL_COVERAGE_CONTEXT_TARGETED"
#define UCIS_FORMAL_COVERAGE_CONTEXT_ANCILLARY \
    "UCIS_FORMAL_COVERAGE_CONTEXT_ANCILLARY"
#define UCIS_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS \
    "UCIS_FORMAL_COVERAGE_CONTEXT_INCONCLUSIVE_ANALYSIS"

int
ucis_SetFormalStatus(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT assertscope,
    ucisFormalStatusT formal_status);

int
ucis_GetFormalStatus(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT assertscope,
    ucisFormalStatusT* formal_status);

int
ucis_SetFormalRadius(
    ucisT db,
```

```
    ucisHistoryNodeT test,
    ucisScopeT assertscope,
    int radius,
    char* clock_name);

int
ucis_GetFormalRadius(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT assertscope,
    int* radius,
    char** clock);

int
ucis_SetFormalWitness(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT assertscope,
    char* witness_waveform_file_or_dir_name);

int
ucis_GetFormalWitness(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT assertscope,
    char** witness_waveform_file_or_dir_name);

int
ucis_SetFormallyUnreachableCoverTest(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT coverscope,
    int coverindex);

int
ucis_ClearFormallyUnreachableCoverTest(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT coverscope,
    int coverindex);

int
ucis_GetFormallyUnreachableCoverTest(
    ucisT db,
    ucisHistoryNodeT test,
    ucisScopeT coverscope,
    int coverindex,
    int* unreachable_flag);

ucisFormalEnvT
ucis_AddFormalEnv(
    ucisT db,
    const char* name,
    ucisScopeT scope);

int
ucis_FormalEnvGetData(
    ucisT db,
    ucisFormalEnvT formal_environment,
    const char** name,
    ucisScopeT* scope);

ucisFormalEnvT
ucis_NextFormalEnv(
    ucisT db,
    ucisFormalEnvT formal_environment);

int
ucis_AssocFormalInfoTest(
    ucisT db,
    ucisHistoryNodeT test,
    ucisFormalToolInfoT* formal_tool_info,
    ucisFormalEnvT formal_environment,
    char* formal_coverage_context);
```

```
int
ucis_FormalTestGetInfo(
    ucisT db,
    ucisHistoryNodeT test,
    ucisFormalToolInfoT** formal_tool_info,
    ucisFormalEnvT* formal_environment,
    char ** formal_coverage_context);

int
ucis_AssocAssumptionFormalEnv(
    ucisT db,
    ucisFormalEnvT formal_env,
    ucisScopeT assumption_scope);

ucisScopeT
ucis_NextFormalEnvAssumption(
    ucisT db,
    ucisFormalEnvT formal_env,
    ucisScopeT assumption_scope);

/*----------------------------------------------------------------------------
 *  Miscellaneous
 *
 *--------------------------------------------------------------------------*/
/*
 *  ucis_GetFSMTransitionStates()
 *  Given a UCIS_FSM_TRANS coveritem, return the corresponding state coveritem
 *  indices and corresponding UCIS_FSM_STATES scope.
 *  This API removes the need to parse transition names in order to access
 *  the states.
 *  Returns the related scope of type UCIS_FSM_STATES on success, NULL on failure
 *  On success, the transition_index_list points to an integer array of
 *  UCIS_FSM_STATES coverindexes in the transition order. The array is terminated
 *  with -1 and the array memory is valid until a subsequent call to this routine
 *  or closure of the database, whichever occurs first. Callers should not
 *  attempt to free this list.
 */

ucisScopeT
ucis_GetFSMTransitionStates(
    ucisT      db,
    ucisScopeT trans_scope,             /* input handle for UCIS_FSM_TRANS scope */
    int        trans_index,             /* input coverindex for transition */
    int *      transition_index_list);  /* output array of int, -1 termination */

/*
 *  ucis_CreateNextTransition()
 *  This is a specialized version of ucis_CreateNextCover() to create a
 *  transition coveritem. It records the source and destination state
 *  coveritem indices along with the transition.
 *  The parent of the state coveritems is assumed to be a sibling of the parent
 *  of type UCIS_FSM_STATES.
 *  Returns the index number of the created coveritem, -1 if error.
 */
int
ucis_CreateNextTransition(
    ucisT db,
    ucisScopeT parent,            /* UCIS_FSM_TRANS scope */
    const char* name,             /* name of coveritem, can be NULL */
    ucisCoverDataT* data,         /* associated data for coverage */
    ucisSourceInfoT* sourceinfo,  /* can be NULL */
    int*  transition_index_list); /* input array of int, -1 termination */

#ifdef __cplusplus
}
#endif

#endif
```

Function Index

Alphabetical index of UCIS API functions.