Unified Coverage Database API User Guide

Version C-2009.06
June 2009

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of
>
> _____ and its employees. This is copy number_____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

1

Introducing UCAPI

The Unified Coverage Application Programming Interface, or UCAPI, provides a single, consistent, and extensible interface to access coverage data generated by Synopsys tools. With this API, you can create custom reports and tools for displaying and analyzing coverage data.

The UCAPI library provides a set of C functions that can be called to get handles to objects in the coverage database. Handles have properties that may be retrieved, such as name, coverage status, and source location information. Handles are connected with 1-to-1 and 1-to-many relations that allow you to traverse all of the data in the database.

UCAPI coverage data is represented using a small set of simple building blocks that can be used to model any type of coverage data. The objects in the database are the same regardless of the design language (for example, Verilog or VHDL), functional coverage

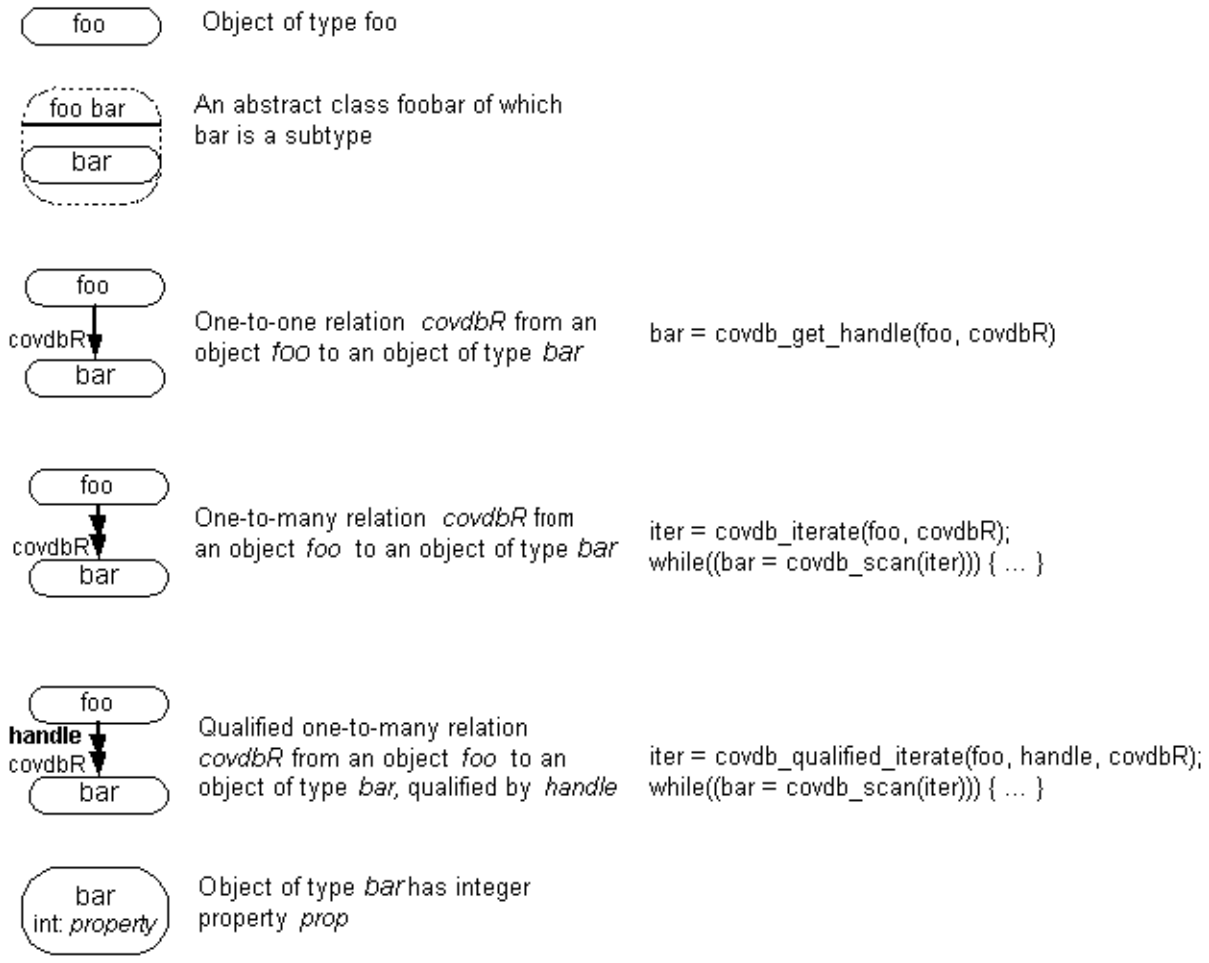
language, or coverage metric type. Different coverage metrics arrange the objects in different ways, but the same basic building blocks are used through UCAPI. This means you can write applications that can understand any type of coverage data without having to follow a set of rules for each different metric.

UCAPI does not provide a complete representation of the design – only what is necessary to present coverage information in proper context. For example, there is no representation of the control structure of code – only the aspects of the design that are monitored by each different metric. Data about the design that is independent of any specific coverage metric is called *unqualified* data.

Inside each region in the design, the coverage data for a given metric is made of UCAPI building blocks. These represent the objects that were monitored for coverage in that region. Since each metric has its own separate collection of such objects, this is called *metric-qualified* data.

Whether or not an object is covered depends on which test or collection of tests you have loaded. For example, a cross bin may have been covered in one simulation run, but not in another. Data that depends on a particular test or set of tests is called *test-qualified data*.

UCAPI uses a series of diagrams to represent object relationships. The following is a summary of the relation types.



Database Contents

A UCAPI database is loaded from one or more directories. The first directory loaded must always contain the unqualified design information - this is typically the data directory created when the design was compiled. Subsequent directories can be full directories (containing unqualified data) or test-only directories (containing only information produced by one or more test runs).

Persistence

UCAPI handles are either volatile or persistent. A persistent handle is guaranteed to remain valid until the application explicitly releases it. Volatile handles are guaranteed to be valid only until the next UCAPI operation is performed.

Some UCAPI functions automatically create persistent handles. But for performance reasons, most UCAPI functions return volatile handles. This results in significantly faster performance since new handles are not allocated for each call - the previous handle is reused “in place”. Applications can get a persistent handle for any volatile handle on demand.

The following rules explain in which conditions handles are persistent or remain valid:

1. Handles returned by `covdb_iterate` and `covdb_qualifed` iterate are persistent and must be freed using `covdb_release_handle` or memory leaks will occur.
2. A handle returned by `covdb_scan` on an iterator is guaranteed to be valid only until the next call of `covdb_scan`, `covdb_iterate`, or `covdb_qualified_iterate`.
3. Handles returned by `covdb_load`, `covdb_merge`, and `covdb_loadmerge` are persistent and should be freed using `covdb_unload` when the application is done with them.
4. Handles returned by `covdb_get_handle` and `covdb_get_qualified_handle` are not persistent and may be invalidated by the next call to either `covdb_get_handle` or `covdb_get_qualified` handle.

5. All test-qualified handles become invalid when the test is unloaded even if they have been made persistent by the application.
6. All handles associated with a design become invalid when the design in unloaded even if they have been made persistent by the application.
7. Handles passed to an error callback function are persistent and remain valid until released by the application using `covdb_release_handle`.

UCAPI Setup and Compilation Requirements

UCAPI is supported on RedHat, Solaris, and SuSE. Depending upon the OS, different compilation options will need to be passed to the compiler. We recommend that you use a setup script similar to the following:

```
#!/bin/csh

#
# Common env var setup script for UCAPI examples
#

if (-f /etc/SuSE-release) then
    #SuSE
    set OTHER_FLAGS = -m32
    set ARCH = suse9
else if (-f /etc/redhat-release) then
    # REDHAT
    set ARCH = linux
    set OTHER_FLAGS = -m32
else
    # Solaris
    set OTHER_FLAGS = "-lsocket -lnsl"
```

```
    set ARCH = sparcOS5
endif

#Make sure VCS_HOME is set to a directory
if (! -d $VCS_HOME) then
   echo Error: VCS_HOME does not point to a valid directory
endif

# Make sure the current ARCH subdirectory exists
if (! -d $VCS_HOME/$ARCH) then
   echo Error: this is an $ARCH machine but there is no VCS
installation for $ARCH at $VCS_HOME
endif

#Setup the LD_LIBRARY_PATH
if(! $?LD_LIBRARY_PATH) then
   setenv LD_LIBRARY_PATH ${VCS_HOME}/${ARCH}/lib
else
   setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${VCS_HOME}/
${ARCH}/lib
endif

# A Summary
echo UCAPI Setup Complete
echo "   VCS_HOME          : " $VCS_HOME
echo "   ARCH              : " $ARCH
echo "   LD_LIBRARY_PATH   : " $LD_LIBRARY_PATH
```

As shown in the script, Redhat and SuSE require the `-m32` option to be passed to the C compiler, while Solaris requires the `-lsocket` and `-lnsl` libraries to be included. The script also verifies that `VCS_HOME` and the OS specific install of VCS is present on the system before attempting to compile. This is required as the UCAPI library is precompiled for different OS's and included within the VCS install.

Once the setup is complete, the c compiler, `cc`, is called with the following arguments:

```
cc -g -I${VCS_HOME}/include -o ucapi_example ucapi_example.c
\
      $VCS_HOME/$ARCH/lib/libucapi.a -ldl -lm $OTHER_FLAGS
|& tee compile.out
```

Note that `$VCS_HOME/include` will include necessary UCAPI include files and `$VCS_HOME/$ARCH/lib` includes the precompiled UCAPI library file. Any operating system specific flags will be supplied by the `setup` script.

Enabling Error Reporting

By default, UCAPI will not print error messages to `stdout`. However, this option can be enabled using the `covdb_configure` call:

```
covdb_configure(covdbDisplayErrors, "true");
// setting to "false" disables / default
```

It is recommended that the above line of code be added to the beginning of your UCAPI program.

Data Model

This section describes the UCAPI objects used to model the contents of a coverage database.

The basic concept of UCAPI is that any coverage metric's objects can be represented using a small set of building blocks. By defining relations between these objects and setting properties (such as name, relative weight, covered/not-covered) on them, the full set of coverage data can be represented, while keeping the interface simple, and the same for all metrics.

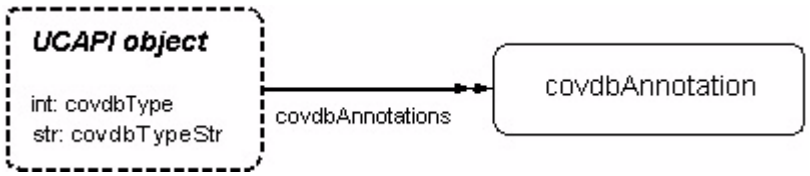
UCAPI Object

This section describes properties and relations that apply to all UCAPI handles.

The `covdbType` property can be read from any UCAPI object handle, and it gives the UCAPI type of the object. The `covdbTypeStr` property gives the UCAPI type as a string, for example, type `covdbTypeStr` property of a handles of type `covdbBlock`.

Any UCAPI object may have string annotations on it, which can be iterated over using the `covdbAnnotations` 1-to-many relation. Annotations may also be retrieved by using the `covdb_get_annotation` and `covdb_get_qualified_annotation` functions described in the “Reading Annotations” section of the *Unified Coverage Database API Reference Manual*.

The `covdbAnnotations` relation is not yet supported. All annotations must currently be accessed by name using `covdb_get_annotation` or `covdb_get_qualified_annotation`. The `covdbTypeStr` property is also not yet supported.



Common Properties

This section describes properties that are supported on many UCAPI handle types.

These properties can be read without a test handle:

- `covdbLineNo` – the line in the source file where the object's text begins.
- `covdbName` – the name of the object
- `covdbFullName` – the full name (including hierarchical name, if applicable)
- `covdbFileName` – the source file in which this object is defined

The following properties may only be read from metric-qualified objects, but don't require a test handle:

- `covdbWeight` – the relative weight of this coverable object.
- `covdbCoverable` – this property represents the number of objects that are coverable. For a simple coverable object such as a toggle sequence, the `covdbCoverable` value will be one. This value may be greater than one for non-coverable objects, such as `covdbContainers`, `covdbSourceDefinitions`, etc. It may also be greater than one for a coverable object handle that represents multiple distinct coverable objects.
- `covdbCovGoal` – the coverage percentage to be reached for this coverable object. By default, this is 100.0.
- `covdbCovCountGoal` – the number of times an object must be hit to count it as covered. By default, this is 1.

- `covdbAutomatic` – non-zero if the object contains only automatically-created sub-objects (for containers, crosses, and sequences)

The following properties require a test handle to specify for which test the data is being queried:

- `covdbCovered` – if the value of the `covdbCovered` property is equal to the value of the `covdbCoverable` property, the handle's contents are fully covered.
- `covdbCovCount` – the number of times the object has been covered.

Note that `covdbCovCount` property can only be read if the coverage tool that monitored the data had counting enabled.

The `covdbCovStatus` property may be read from metric-qualified objects with or without a test handle. If it is read without a test handle, then the `covdbStatusCovered` flag will never be set, since only a test can mark an object covered.

The value of `covdbCovStatus` can be any combination of the following:

- `covdbStatusCovered` – covered
- `covdbStatusUnreachable` – can never be covered, as when a formal tool has proven it is impossible to cover
- `covdbStatusIllegal` – should be considered an error if the object was covered
- `covdbStatusExcludedAtCompileTime` – the object was marked excluded at compile time

- `covdbStatusExcludedAtReportTime` – the object was marked excluded at report time
- `covdbStatusExcluded` – the object was marked excluded either at report time or compile time – can be checked by applications when it doesn't matter when the object was marked excluded

Coverable objects with `covdbStatusExcluded` set (or `covdbStatusExcludedAtReportTime` or `covdbStatusExcludedAtCompileTime`) are ignored when computing coverage scores – the `covdbCoverable` property will not include them in its count, for example. However, they may still be useful information. For example, some coverage tools will collect expression coverage information for debugging or verification analysis purposes that is not counted against a project’s coverage goals. When coverable objects inside a container (or cross, or sequence) are marked `covdbStatusExcluded`, they will be excluded from the `covdbCovered` and `covdbCoverable` values for that container (or cross or sequence).

Note that, while all of these properties may be retrieved from any coverable object handle, all objects may not have useful information. For example, a container object may have no associated file name or line number, but that information may be retrieved from the objects it contains. Whether or not this information is present for a given object depends on the metric and the coverage tool that created that object.

Coverable Objects

Coverable objects are entities in the database that can be covered or not covered. This small set of object types is sufficient to model any type of coverage. All coverable objects are metric-qualified. That means that each coverable object is associated with exactly one metric.

Coverable objects are assigned names that are unique within the region or container in which they are defined. Named objects, such as signal bits, may contain a list of lower-level objects - for example, a signal bit "x[1]" can have toggle objects representing the toggle from 0 to 1 and the toggle from 1 to 0. In this example, the names of those objects in UCAPI are "0 > 1" and "1 > 0".

Variant names are assigned by UCAPI using the name of the module plus a parenthesized list of its parameter/generic values. For example, if a module M has two variants for a given metric, its variant names might be "M(p = 1)" and another variant "M(p = 2)". The name of the unqualified module is "M". See the section titled "Source Definition and Source Instance" for more information on variants.

In the diagrams for coverable objects in this section, only the special properties and relations for each type are shown below.

Block

A block represents an atomic coverable object. The `covdbType` property for a block object is `covdbBlock`. Block objects have no additional relations on them. Depending on the metric, they may

have `covdbFileName`, `covdbLineNo`, and `covdbName` properties.

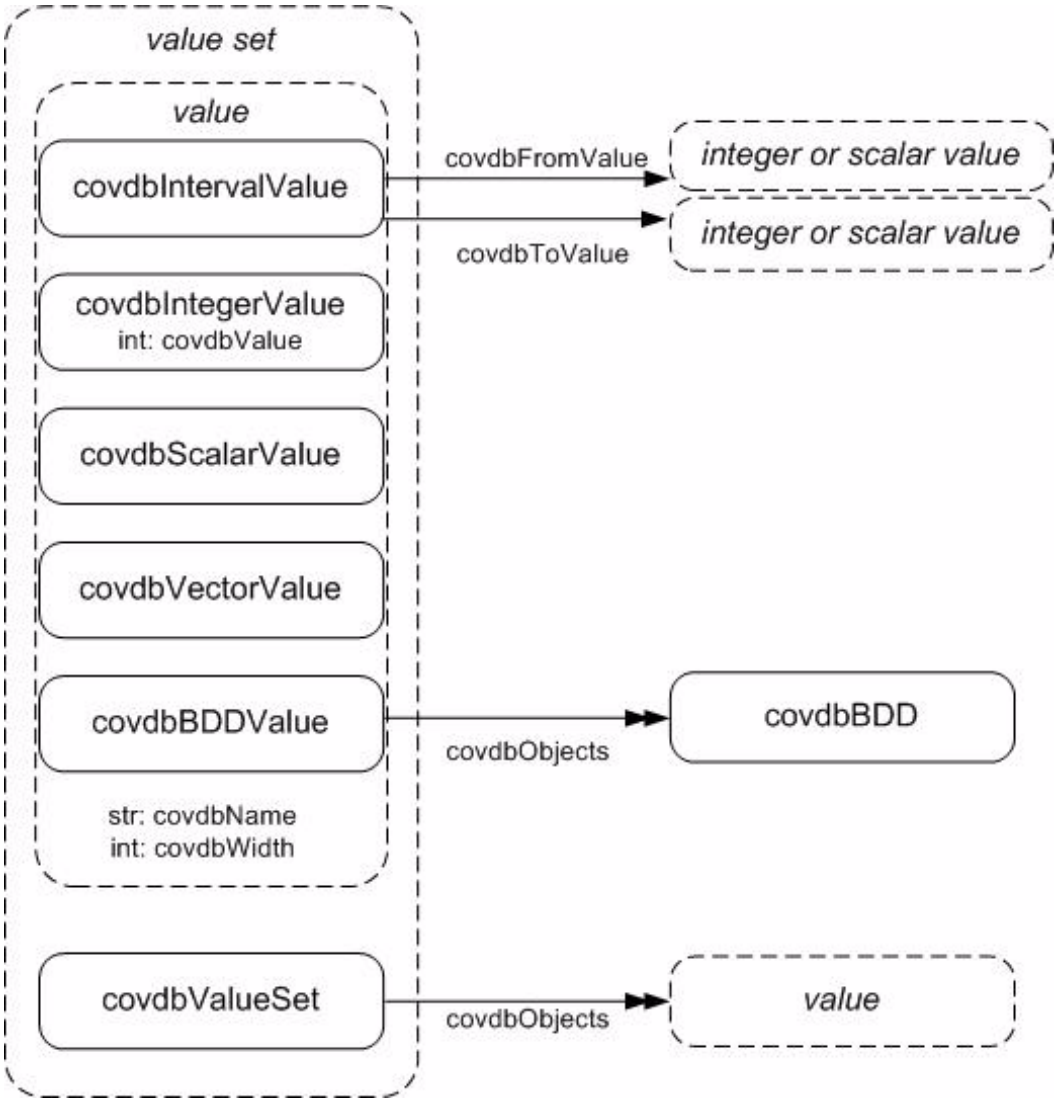
Value Set

A value set represents a set of values taken on by an expression. The value may be represented as an exact value, an interval (range) of values, a Binary-Decision Diagram (BDD), or a set of values that is any combination of these values. These coverable object handle types are collectively called "value sets".

The `covdbType` property of a value set object is one of `covdbIntervalValue`, `covdbIntegerValue`, `covdbScalarValue`, `covdbVectorValue`, `covdbBDDValue`, or `covdbValueSet`. The following diagram shows their structures. This section explains how the properties and value(s) for each type of handle are retrieved.

All value set objects support these properties:

- `covdbName` - a string representation of the value. This may be simply the value as a string, or it could be a name for the value, such as the name of an enumerated type.
- `covdbWidth` - the bit-width of the expression.


value set
value
covdbIntervalValue
covdbIntegerValue
int: covdbValue
covdbScalarValue
covdbVectorValue
covdbBDDValue
str: covdbName
int: covdbWidth
covdbValueSet
covdbFromValue
covdbToValue
integer or scalar value
integer or scalar value
covdbObjects
covdbBDD
covdbObjects
value

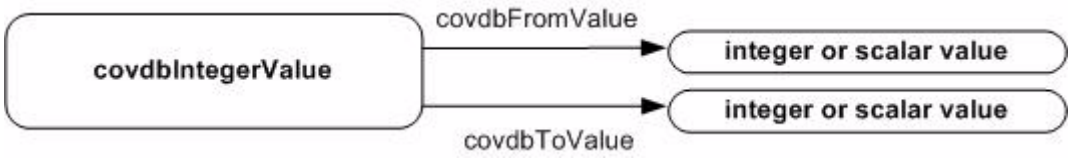
Integer and Scalar values

The value of an object of type `covdbIntegerValue` or `covdbScalarValue` is read using the `covdbValue` property. For integers, the value is the integer; for `covdbScalarValues`, the value is one of the enumerated types `covdbScalar0`, `covdbScalar1`, or `covdbScalarX` (see section 6.5).

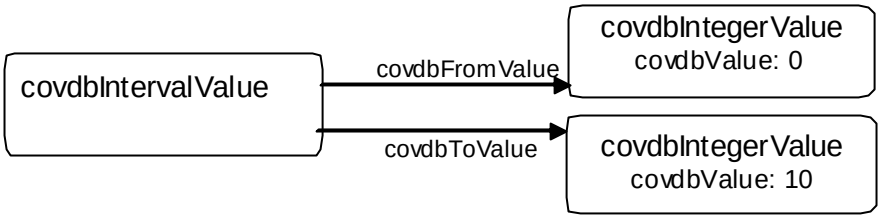
For example:

```
covdbTypesT objty = covdb_get(objHdl, regHdl, NULL,
                                covdbType);
if (covdbIntegerValue == objty) {
     int val = covdb_get(objHdl, regHdl, NULL, covdbValue);
} else if (covdbScalarValue == objty) {
    covdbScalarValueT val =
           covdb_get(objHdl, regHdl, NULL, covdbValue);
}
```

Interval values



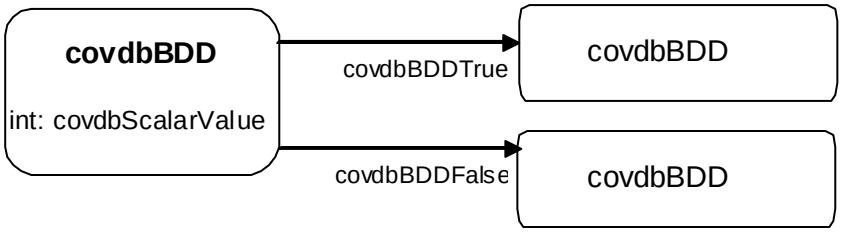
A handle of type `covdbIntervalValue` contains two `covdbIntegerValue` handles - one is the "from" value and the other is the "to" value of the interval. For example, the interval [0, 10] would be represented as shown follows:



BDD values

The `covdbBDD` type is used to represent BDD values - ordered decision diagrams. Whether a BDD object is a terminal or not can be determined from the `covdbScalarValue` property. This property will return:

```
covdbValue0: for 0 valued terminals
covdbValue1: for 1 valued terminals
covdbValueX: for non-terminal objects
```



BDD values are currently not supported.

Vector values

An object of type `covdbVectorValue` has two properties:

- `covdbSigned` - non-zero if the vector value is a signed value.

- `covdbTwoState` - non-zero if the vector value is in two-state representation. If zero, the value is in four-state representation.
- `covdbWidth` - the width of the value in bits.

If the vector value's `covdbTwoState` property is non-zero, the values will be represented as an array of unsigned ints. The value is retrieved with the function `covdb_get_vec_2state_value`.

If `covdbTwoState` is zero, the values will be represented as an array of `covdbVec32ValueT` structs. Each struct has a 32-bit control word and a 32-bit data word. Each bit of the control word and each corresponding bit of the data word represents a single scalar value (0, 1, X, or Z). The value for each combination is shown follows:

Table 1-1

control	data	value
0	0	0
0	1	1
1	0	Z
1	1	X

For example, the following is the declaration of `covdbVec32ValueT` from the `covdb_user.h` header file:

```
typedef struct covdbVec32Value_s {
    unsigned int c;
    unsigned int d;
} covdbVec32ValueT;
```

Also, the following is an example of code that reads and prints out `covdbVectorValueT` handles:

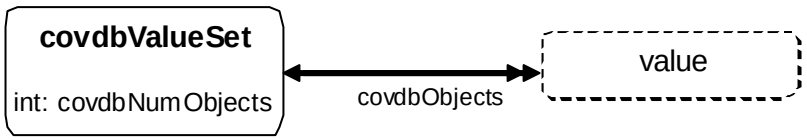
```
if (covdbVectorValue == objty) {
    int signp = covdb_get(objHdl, regHdl, NULL, covdbSigned);
```

```
        int width = covdb_get(objHdl, regHdl, NULL, covdbWidth);

         if (covdb_get(objHdl, regHdl, NULL, covdbTwoState)) {
               unsigned *vals =
                covdb_get_vec_2state_value(objHdl, regHdl);

               for(i = 0; i < width; ) {
                     unsigned wd = vals[i / 32];
                   for(j = 0; j < 32 && i < width; i++, j++) {
                           printf("%d", (wd >> j) & 0x1);
                     }
               }
               printf("\n");
         } else {
               covdb4stateValT *vals =
                  covdb_get_vec_4state_value(objHdl, regHdl);

               for(i = 0; i < width; ) {
                     unsigned wd = vals[i / 32];
                   for(j = 0; j < 32 && i < width; i++, j++) {
                           int c = vals[i].c;
                           int d = vals[i].d;
                           char rep;

                           if (!c && !d) rep = '0';
                           else if (!c && d) rep = '1';
                           else if (!d) rep = 'Z';
                           else rep = 'X';

                           printf("%c", rep);
                     }
               }
               printf("\n");
         }
      }
```

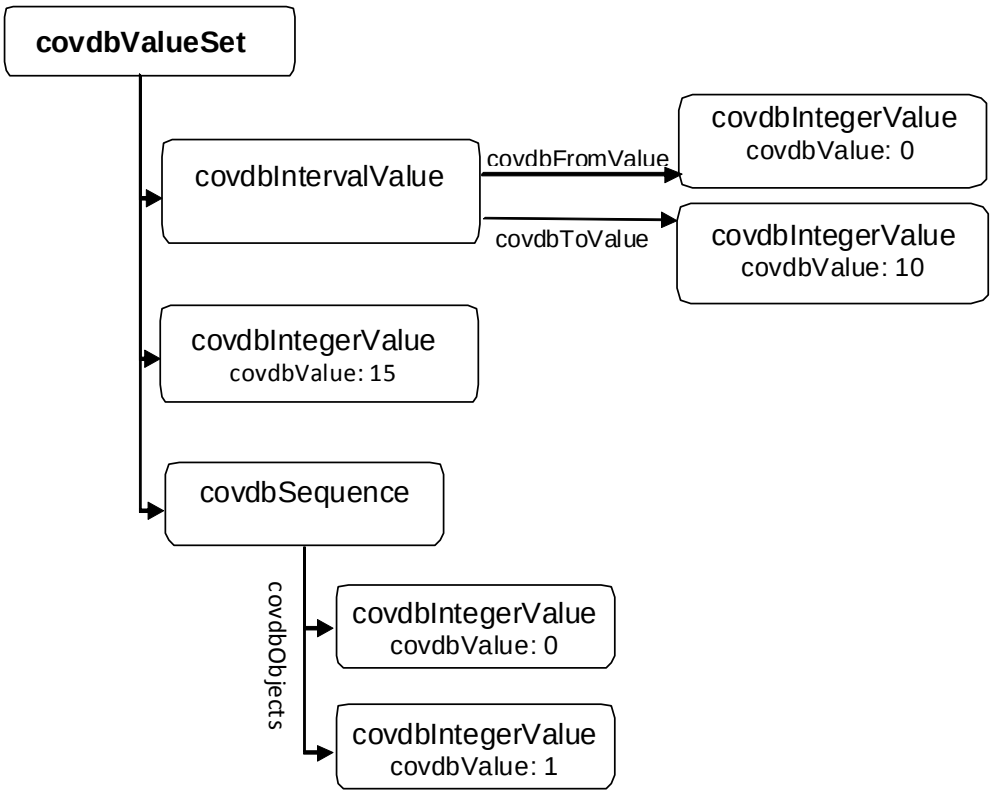
Value Set

A `covdbValueSet` handle is a collection of value sets or sequences. It represents a given expression taking on one of the values in the value sets or going through one of the sequences of values specified by the sequence objects.

A `covdbValueSet` handle is itself a coverable object. It is either covered or not covered. The objects inside it are there only to describe its structure. For example, the `covdbCovCount` of the `covdbValueSet` handle tells how many times any one of the values or sequences occurred - applications cannot query each object inside the bin to get their individual `covdbCovCount` values.



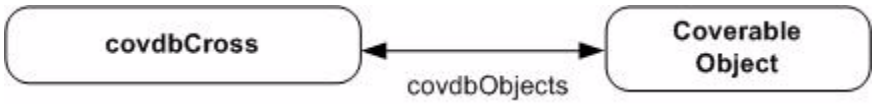
For example, a coverable object that represented an expression taking a value between 0 and 10 (inclusive), the value 15, or the value of the expression changing from 0 to 1 would be represented as follows:



Sequence

An object of type `covdbSequence` contains an ordered collection of other coverable objects and represents the *sequential* coverage of each member object. For example, a sequence of value sets might represent a signal taking on the specified sequence of values.

Note that a `covdbSequence` object itself is either covered or not covered. The coverage statuses of its contents do not contribute to its coverage status, because the sequence itself is an atomic coverable object.

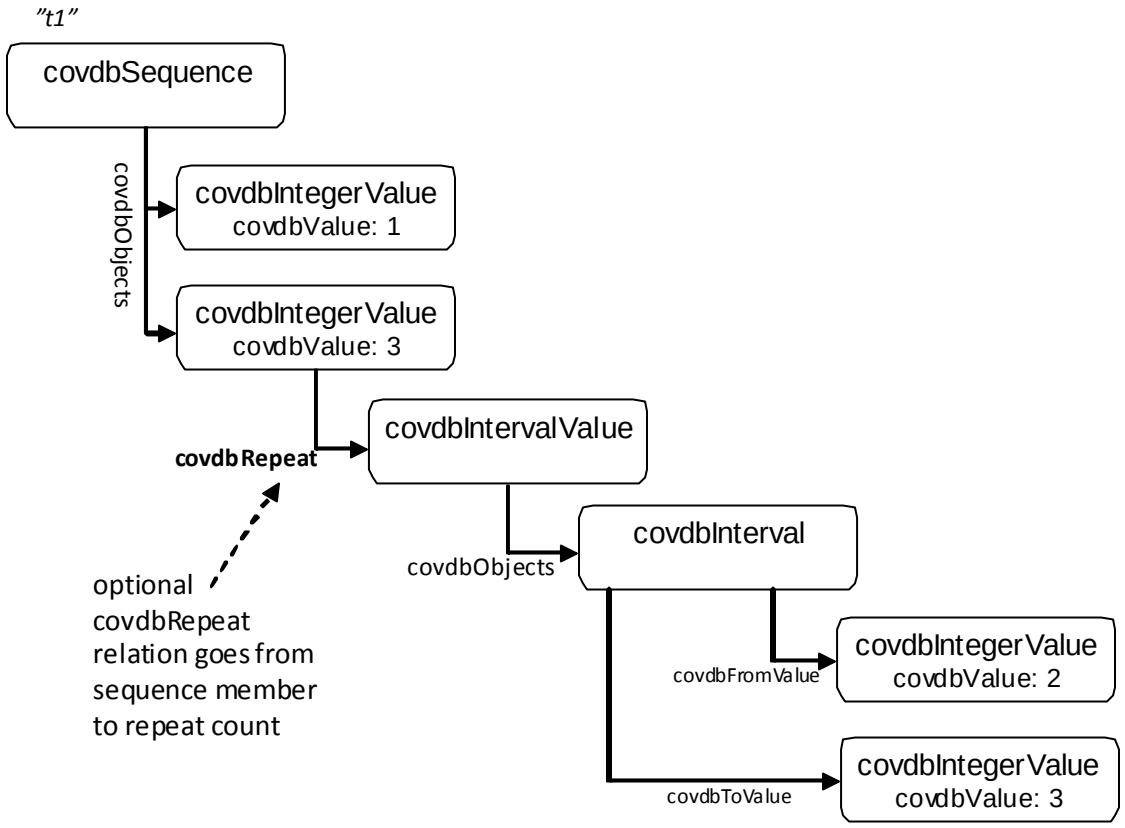


A coverable object in a `covdbSequence`'s objects list has an optional repeat count. This indicates that the object is repeated the specified number of times in the sequence. The repeat count may be a single integer or it may be a set of values, indicating the repetition may be any of the specified numbers.

For example, if a testbench coverage transition bin is defined as:

```
coverpoint a
      bin t1 : 1 -> 3 * (2..3)
```

This indicates that the value of `a` should transition from '1' to '3', then from '3' to '3' again 1 or 2 additional times. In UCAPI, this is represented using the `covdbRepeat` relation, as follows:



When a value set is used as a repeat, it will have a value for `covdbRepeatType` that is one of:

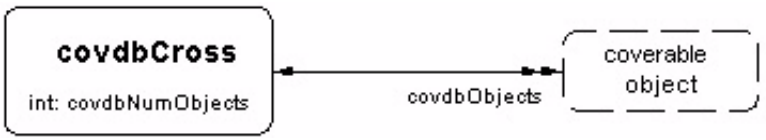
- `covdbGoto`
- `covdbConsecutive`
- `covdbNonsecutive`

When the value set is not a repeat value, the value of `covdbRepeatType` will be -1.

Cross

An object of type `covdbCross` has the same structure as a sequence, but the objects in the `covdbObjects` iterator are in no particular order. A `covdbCross` object represents the *simultaneous* coverage of its member objects.

Like a `covdbSequence` object, a `covdbCross` object is either covered or not covered. The coverage statuses of its `covdbObjects` do not contribute to its coverage status, because the sequence itself is the coverage event.



Design Objects

Design objects are used to give context for coverable objects. Design object handle types include `covdbDesign`, `covdbTest`, `covdbMetric`, `covdbTestName`, `covdbSourceDefinition`, and `covdbSourceInstance`. They do not have coverage state themselves. They can be considered to be the framework on which the coverable objects are distributed. The `covdbName` property can be read from any design object handle.

Design and Test Name

A `covdbDesign` handle indicates a specific design in the coverage database. A design contains a list of test names for which coverage data exists.

The `covdbTestName` handles you get from `covdbAvailableTests` are not the same as `covdbTest` handles – applications cannot use them to get coverage data. But applications can use their `covdbName` properties to load real `covdbTest` handles.

Tests and Metrics

A `covdbTest` handle is a test whose data has been loaded into the UCAPI interface. A test is the result of a single execution of a tool, or the merged results of two or more executions.

Test handles are used to get test-qualified handles and data as described in the introduction to this section. Handles to `covdbTest` objects are accessed using the `covdb_load`, `covdb_merge`, and `covdb_loadmerge` functions.

Metric handles have the covdbType `covdbMetric`. The only property that can be read from a `covdbMetric` handle is its name, using `covdbName`.

The 1-to-many relation `covdbMetrics` may be iterated from a `covdbTest` handle to get the list of metrics that have coverage data in that test.

After tests are loaded, applications can iterate over the list of loaded tests from a design handle using the `covdbLoadedTests` relation:

Source Definition and Source Instance

A `covdbSourceDefinition` handle represents a non-instantiated design region, such as a Verilog module or a testbench coverage group. A `covdbSourceInstance` represents a specific instance of a `covdbSourceDefinition`.

Source regions represent the declarative regions from the design or testbench - modules, entity-architectures, covergroups, packages - and instances of those regions. Region handles can be unqualified, metric-qualified, or test-qualified.

Unqualified regions are the skeleton of the design. You cannot see any coverable objects in them because the list and shape of coverable objects depend on a particular metric.

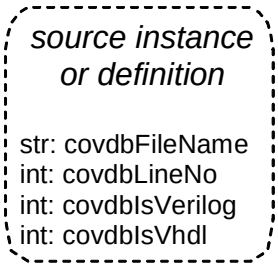
Metric-qualified region handles contain the coverable objects for one specific metric. For example, a condition-qualified handle to a module M will contain the coverable objects for conditions and vectors found and monitored in M.

Test-qualified region handles are used for regions that only exist within a specific test. For example, covergroup handles are always test-qualified, since covergroups are used on a per-test basis - they don't necessarily exist in every test.

The following properties may be read from any source definition or instance:

- `covdbFileName` – the source file name of this definition or instance.
- `covdbLineNo` – the line number where this definition or instance is defined in its source file.
- `covdbIsVerilog` – non-zero if the definition or instance was defined in Verilog.
- `covdbIsVhdl` – non-zero if the definition or instance was defined in VHDL.

Source definitions may represent modules, entity-architectures, or testbench coverage groups. Source instances may be instances of any of these objects.



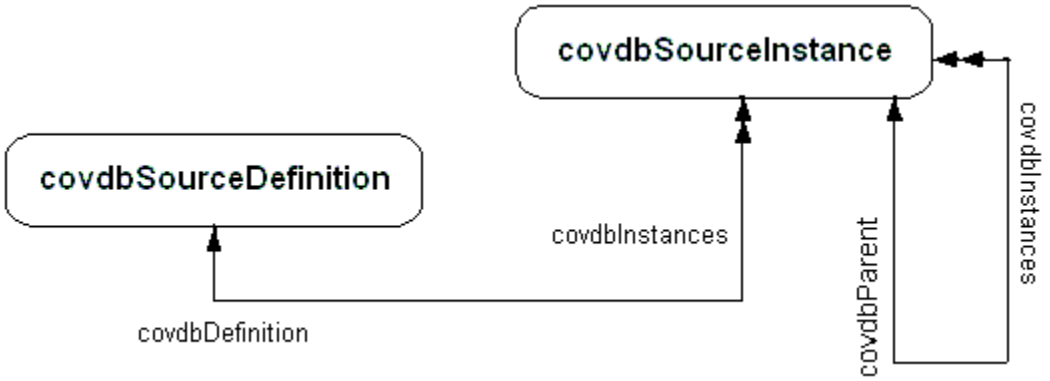
Source definitions have a 1-to-many relation `covdbInstances` to get an iterator of all instances of that definition. Similarly, source instances have a 1-to-1 relation `covdbDefinition` to get the corresponding definition of an instance. Source instances also have

a `covdbInstances` relation from them; each instance in this list is a child instance in the design hierarchy. The `covdbParent` relation is the instance's parent in the design hierarchy.

Properties such as `covdbCovered` and `covdbCoverable` may be read directly from metric-qualified source region handles (`covdbCovered` requires that a test handle be given) to compute the coverage score for that metric. Any property that can be read from an unqualified source definition or instance (such as `covdbFileName`) can also be read from a metric-qualified source definition or instance handle.

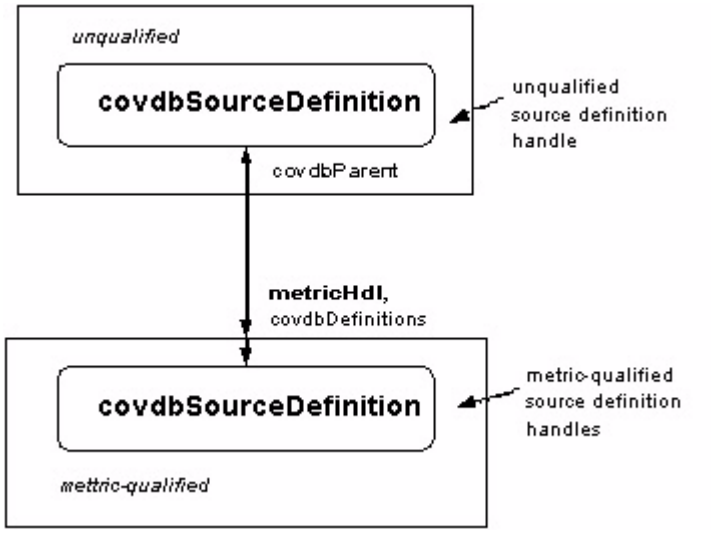
The `covdbDefinitions` and `covdbInstances` 1-to-many relations may be used from a design handle to iterate over all definitions or over the top-level instances in the design, respectively. When these are qualified with a metric, the list of metric-qualified definitions and top-level instances for that specified metric are obtained.

The figures below examine different subsets of the relations between different design region handles. First, is the structure of the unqualified design. Without using any metric handles, an application can walk through the base design – all definitions and all instances of those definitions. Note that the `covdbInstances` relation from a source definition gives its list of self-instances, whereas the `covdbInstances` relation from a source instance gives its instance children in the design hierarchy.

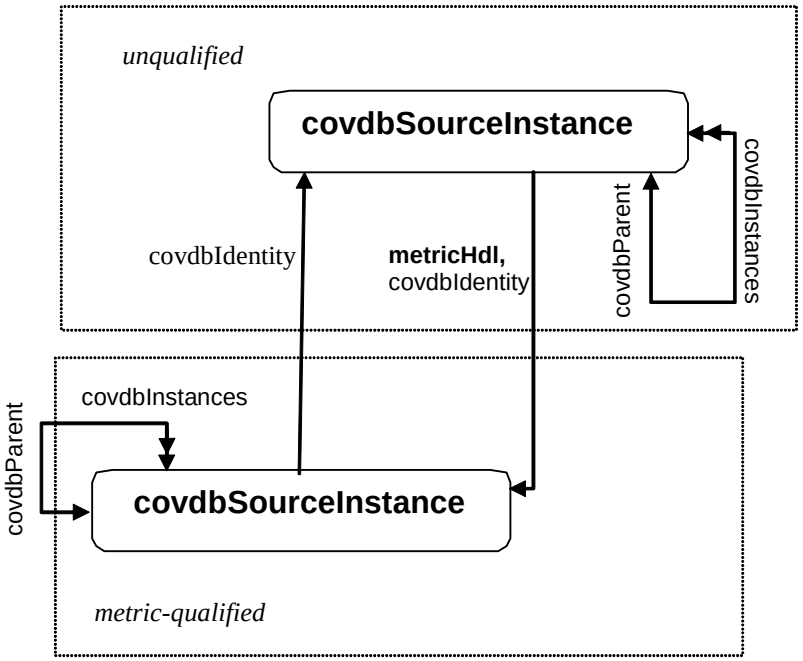


To get coverage information about a given metric in a given region, an application must get a metric-qualified handle for that region. For example, to get the list of FSMs in a module M, applications first get the FSM metric-qualified handle to M.

There may be more than one FSM-qualified handle for M in a design, due to parameter/generic values. In other words, the module M may have been *split* by the FSM metric. Therefore, there is not necessarily only a single FSM-qualified handle for M, and applications use the metric-qualified 1-to-many relation `covdbDefinitions` to walk through those handles, as shown in the following figure:



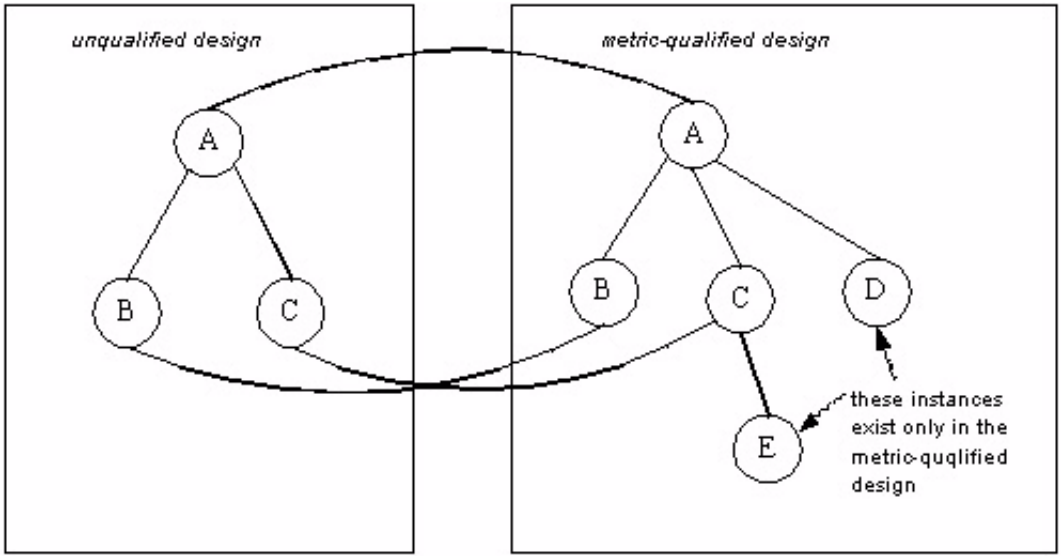
Similarly, unqualified `covdbSourceInstance` handles have metric-qualified versions. However, since each instance is unique in the design – it does not potentially represent multiple versions, like `covdbSourceDefinitions` – there is always only one metric-qualified source instance for each unqualified source instance handle for a given metric. The metric-qualified source instance handle is accessed using the metric-qualified 1-to-1 relation `covdbIdentity`, and the unqualified instance handle is accessed from the metric-qualified instance handle using `covdbIdentity` with no qualifier:



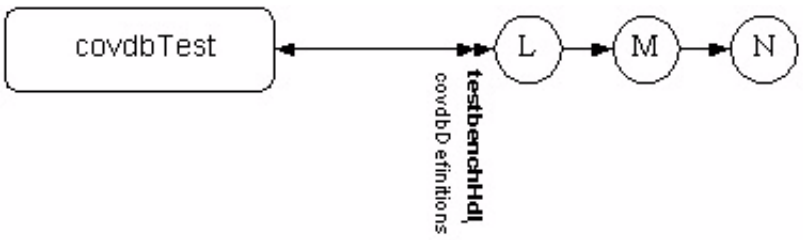
There may be metric-qualified source instances or definitions with no corresponding unqualified handle. For example, some instances exist only for the assertions metric (bound-in OVA units, for example). An application traversing only over the unqualified design will not see these source regions. To distinguish them from metric-qualified handles of unqualified source regions, we call these regions *metric-only* regions.

If a metric-only region is attached to the unqualified design, it will be visible only when iterating over the `covdbInstances` children of the appropriate metric-qualified instance parent.

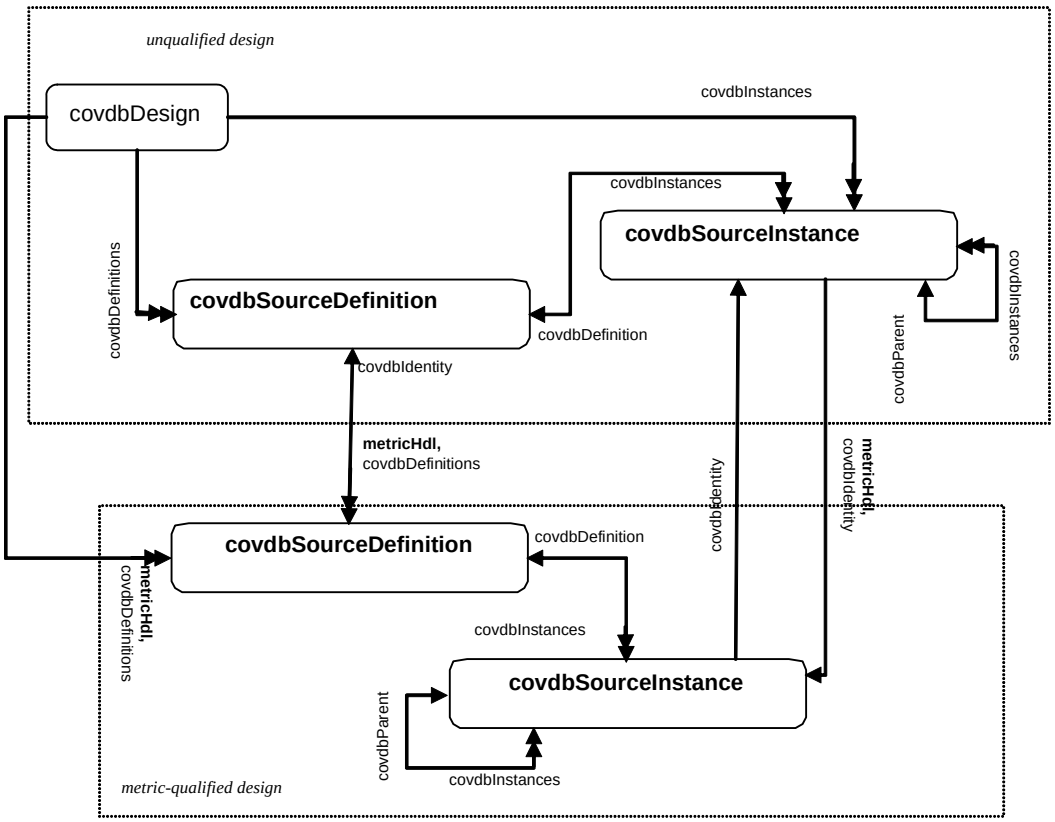
For example, the following figure shows the unqualified and metric qualified design hierarchies for an example design. The source instances D and E are only visible in the metric-qualified design and may not be directly accessed from any unqualified source handle.



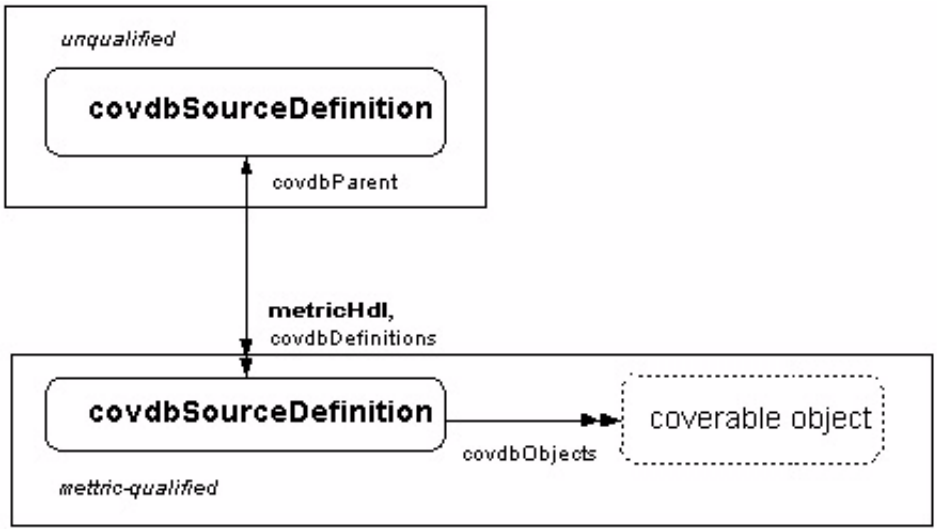
If a metric-only region is separate from the design (such as a testbench coverage covergroup declared outside of any module), applications must use the metric-qualified 1-to-many relations `covdbDefinitions` or `covdbInstances` from a test or design handle – this will give access to the complete list of definitions or the list of metric-qualified top-level instances. For example, in the following figure, L, M, and N are Vera covergroups, accessed from a test handle.



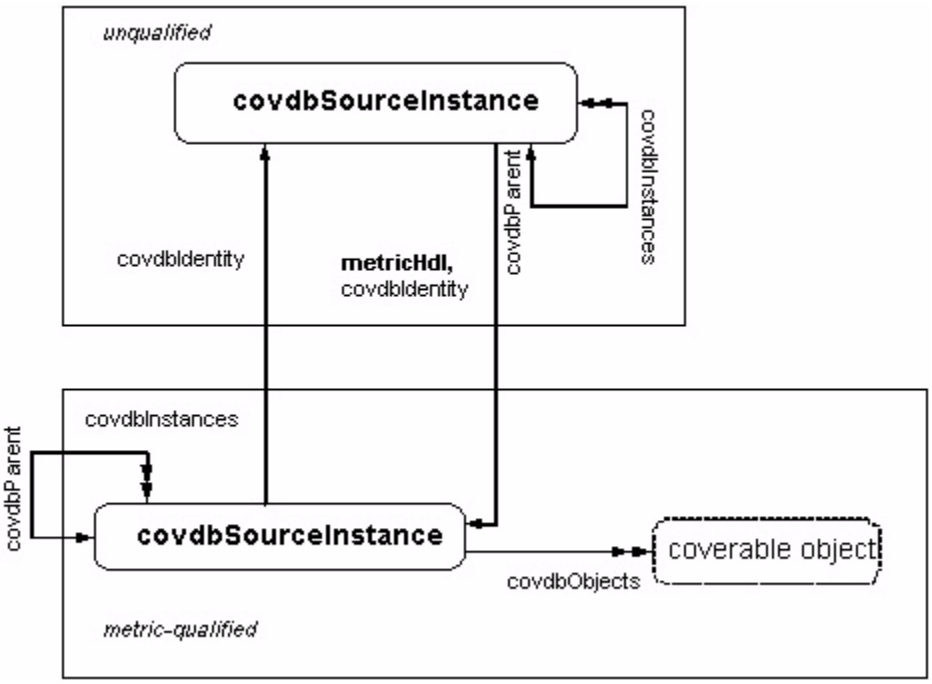
The following figure shows the complete set of relations used to traverse the design.



There is no metric-qualified `covdbObjects` list off of source definitions or instances. All coverable objects are accessed from the metric-qualified source definitions and instances. For example, to get to the coverable objects list for a source definition:



Similarly, to get to the metric-qualified objects from a `covdbSourceInstance` handle:



The `covdbDeepCoverable` and `covdbDeepCovered` properties apply only to metric-qualified `SourceInstance` handles, and return the coverable (covered) counts for the source instance and its entire design subtree.

Test-qualified Source Regions

Some metrics create test-qualified source regions that only exist within a given test. For example, testbench coverage groups do not exist in the unqualified design or even in the metric-qualified design - they only exist in the test-qualified data.

Test-qualified regions are accessed in two ways:

- Directly from a test handle using the metric handle as a qualifier

- From within metric-qualified source definitions and instances in the main design

Like unqualified source definitions, test-qualified source definitions may have multiple versions, due to variations between instances of the groups. These versions are accessed with the test-qualified `covdbDefinitions` relation. As for HDL source definitions, coverable objects are only accessed from the versions of definitions, not from the master definition.

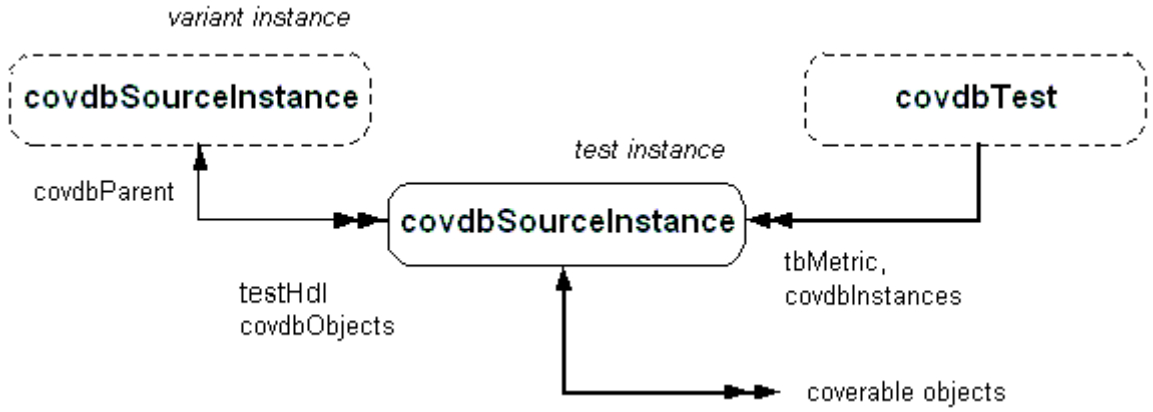
The following figure shows how test-qualified definitions are accessed. Access is either from a testbench-qualified module definition to the list of its test-qualified definitions, or from a test handle giving the list of all test-qualified definitions in the design.

Test-qualified instances may occur inside any HDL module instance, or entirely outside the design hierarchy. They may occur inside an HDL module instance even if they are not defined within the corresponding HDL module[1].

Test-qualified instances that are inside HDL module instances are accessed by the test-qualified `covdbObjects` relation from the metric-qualified HDL module instance handle.

Test-qualified instances that are not within the HDL design hierarchy are accessed from a `covdbTest` handle, using the metric-qualified `covdbInstances` relation. The following figure shows how test-qualified instances are accessed in UCAPI.

1. Such as an SVTB covergroup defined in one module but instanced by another module - UCAPI supports a covergroup defined in $root and instanced in the design, for example.



Covergroup definitions and instances have the same `covdbInstances`/`covdbParent` relationship as HDL definitions and instances.

Other Objects

Container

Objects of type `covdbContainer` are abstractions that represent a collection of coverable objects that logically belong together. Containers may have two different lists in them:

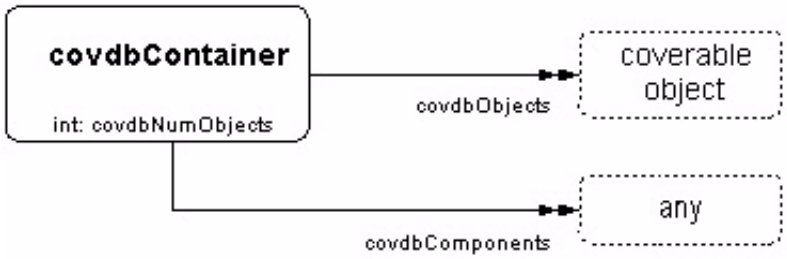
- `covdbObjects` – a list of coverable objects or more `covdbContainers`. The covered/coverable numbers for the container is the sum of the covered/coverable numbers for all of the contained objects.
- `covdbComponents` – a list of objects making up the structure of the container. For example, a container representing a cross declaration will have the list of coverpoints being crossed in its `covdbComponents` list.

Containers whose objects are `covdbCrosses` or `covdbSequences` may also have a `covdbComponents` list. If a container has a `covdbComponents` list, it means that:

- Each cross or sequence in the container's `covdbObjects` list has the same number of objects in it.
- The container will have the same number of items in its `covdbComponents` list as are in each item in its `covdbObjects` list.
- Each item in each cross or sequence will correspond to the same numbered item in the `covdbComponents` list.

The objects in the `covdbComponents` list do not contribute to the coverage values for the container – they are provided only for reference.

Containers are themselves not coverable objects – they contain coverable objects in their `covdbObjects` lists. This is distinct from Sequences and Crosses, each of which represent a single coverable object.



In general, objects in containers are in source declaration order. This is tool-dependent and metric-dependent.

2

Predefined Coverage Metrics

This chapter describes how UCAPI models each of the predefined coverage metrics. Note that while the various metrics described below use terms such as “basic block”, “condition” and “signal,” these are not UCAPI object types, they are just describing what the UCAPI types are being used to model. *All object types are defined in* “Data Model” on page 1-7– there are no special object types for any metric.

Statement Coverage

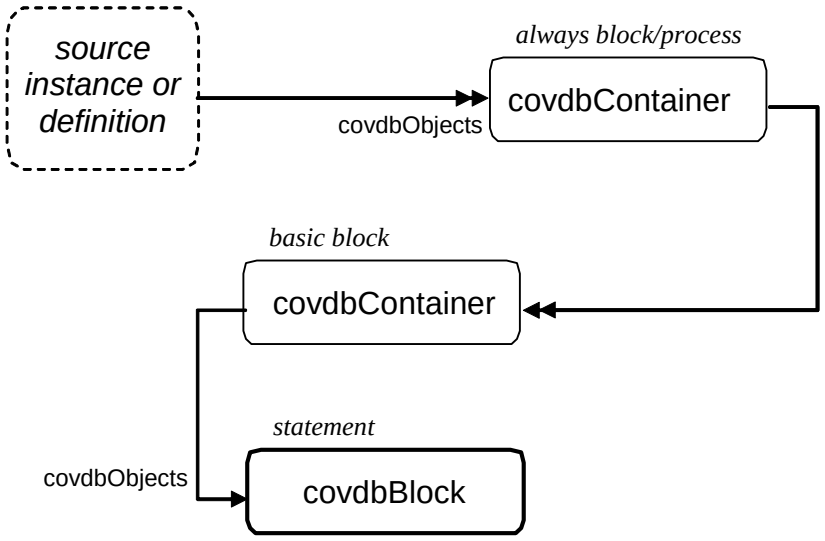
Statement coverage tracks which statements in the design code were executed during simulation.

Statement coverage is organized in two levels, with always blocks/processes at the top level. The type of an always block or process is covdbContainer.

Within each always block/process container, statements are organized into *basic blocks*, which are straight-line sections of code. Basic blocks are also `covdbContainer` objects.

Within the basic block containers, each statement is modeled as a `covdbBlock` object.

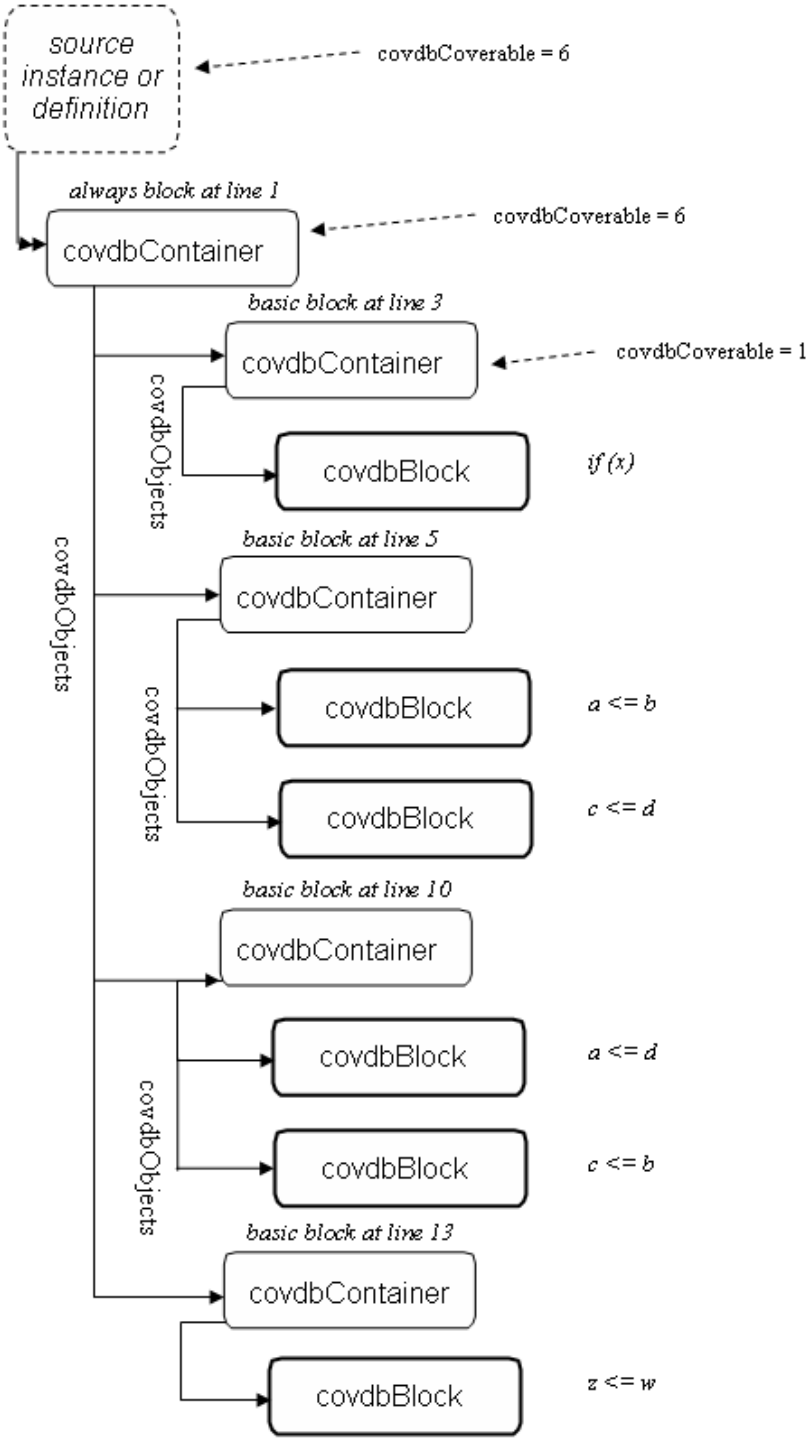
The organization of data for statement coverage is shown in the following figure.



For example, consider the following code:

```
1    always@(posedge clk)
2    begin
3      if (x)
4      begin
5        a <= b;
6        c <= d;
7      end
8      else
9      begin
10       a <= d;
11       c <= b;
12     end
13     z <= w;
14   end
```

This is represented in UCAPI as follows:

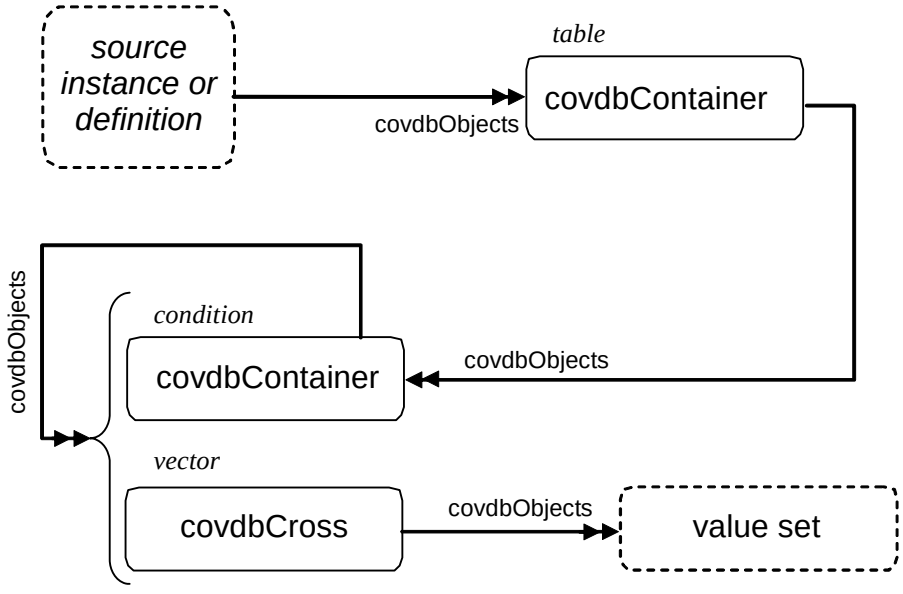


There is no representation of source *lines* as objects in UCAPI. There may be multiple statements on a source line, or a statement may extend across several lines. Source line information is available using the `covdbLineNo` property on the `covdbContainer` and `covdbBlock` handles.

Condition Coverage

Condition coverage tracks which values occurred in logical or bitwise expressions in the design. A *condition* is that logical or bitwise expression, and a *vector* is distinct set of values taken on by the terms in the expression. Conditions are modeled as `covdbContainer` objects, and vectors are modeled as `covdbCross` objects containing value sets.

Conditions are organized into tables, such as logical conditions, non-logical conditions, and event conditions. The `covdbName` of the table container gives the type of conditions in that table.



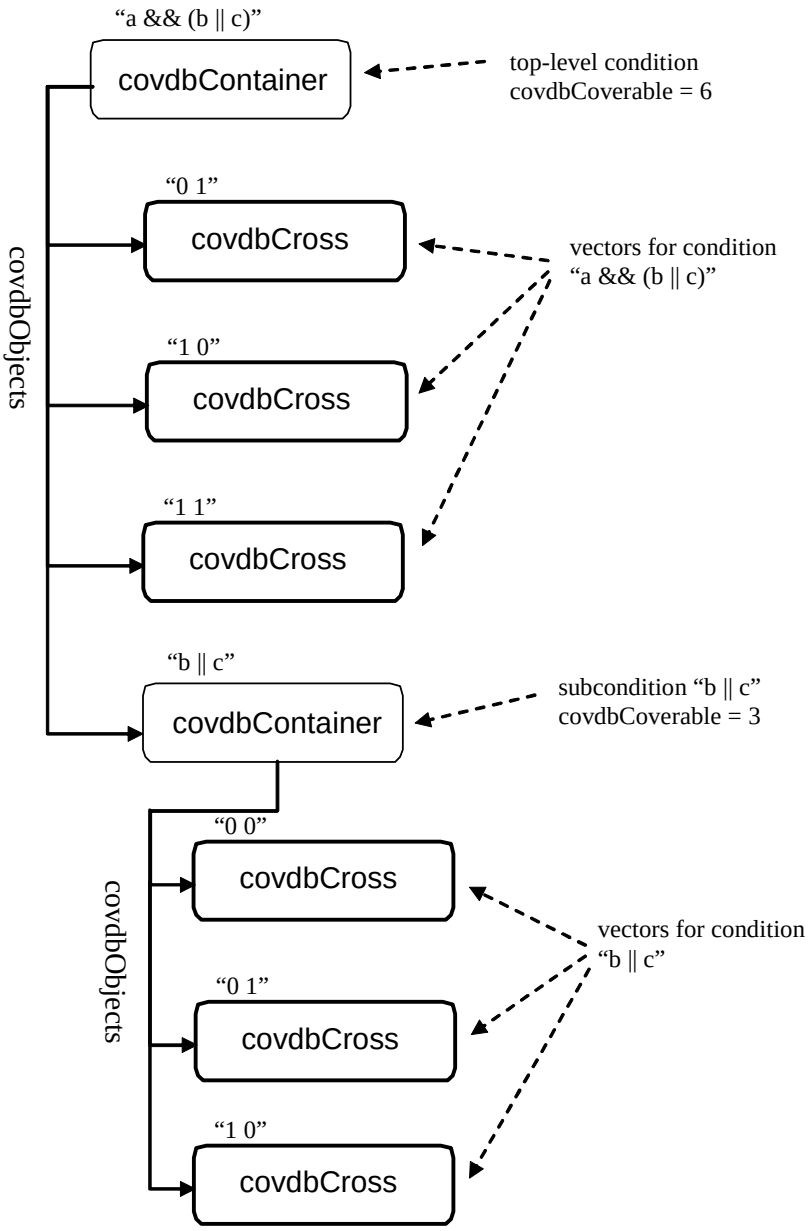
The exact vectors in a given database depend on the compile-time options used with VCS. For example, using the `-cm_cond` full flag results in full truth tables (vectors) for every condition, and using `-cm_cond sop` results in sum-of-products-style vectors.

Note that there is no representation of which conditions are nested *textually* within other conditions (such as the condition for an "`if`" statement that is nested inside a "`case`" statement). However, conditions can have sub-conditions within the same expression, and in this case, the sub-condition will be in the list of objects for the parent condition.

For example, if the following code is compiled in default condition coverage in VCS:

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

The parent condition consists of two terms in a "`&&`" expression, "`a`" and "`(b || c)`". The sub-condition is the expression "`b || c`", which has two terms in a "`||`" expression, "`b`" and "`c`". In this case, UCAPI would represent the condition coverage objects as shown in the following diagram (these conditions would be in the "logical" conditions table, which is not shown in the diagram).



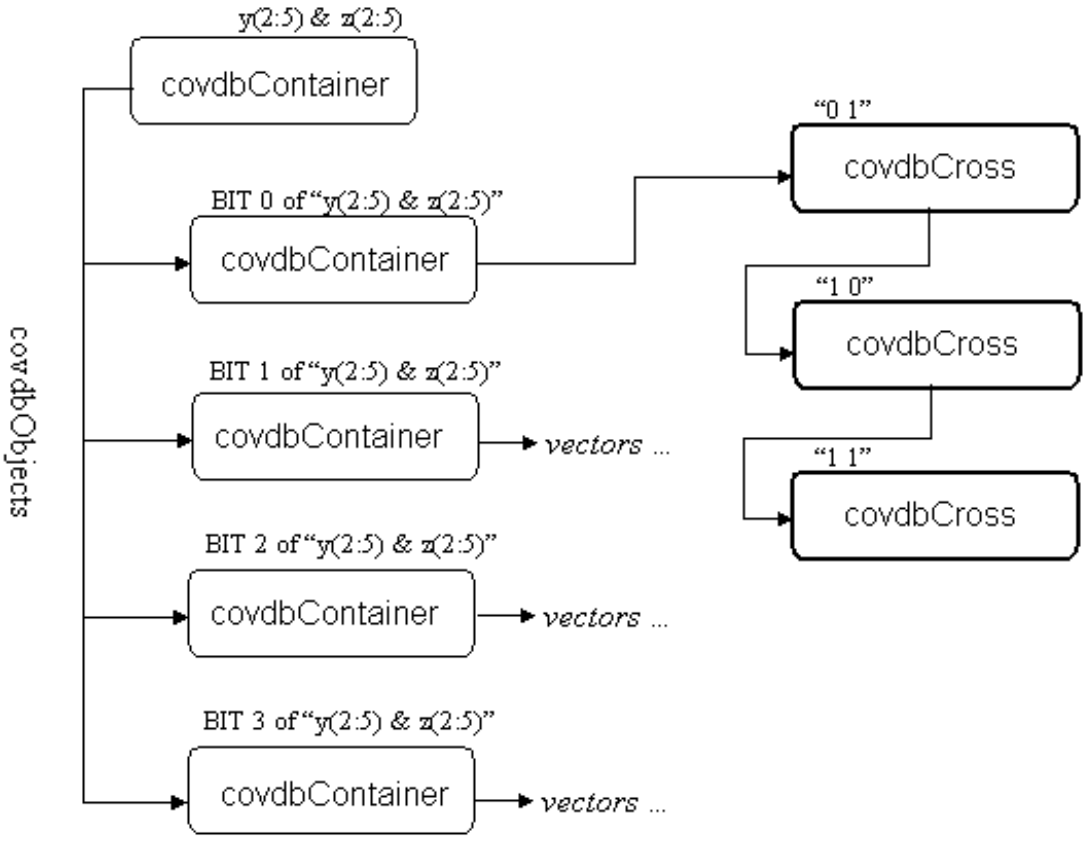
Some conditions are expanded into multiple conditions. A bitwise expression such as:

```
x = y & z;
```

If `x`, `y` and `z` are 4-bit vectors, this represents four different conditions:

```
y(0) & z(0)
y(1) & z(1)
y(2) & z(2)
y(3) & z(3)
```

This is represented in UCAPI as additional conditions inside the main condition, as shown in the following figure. Note that these “non-logical” conditions themselves do not have any vectors of their own – only the single-bit conditions have vectors.



Note that “`BIT 0 of y(2:5) & z(2:5)`” is the expression for “`y(2) & z(2)`”, but UCAPI does not provide the original indices.

Another type of condition coverage, called “sum-of-products” or SOP condition coverage, is modeled similarly. For example, consider the expression:

```
((a == 2) || b && c) || (d && e)
```

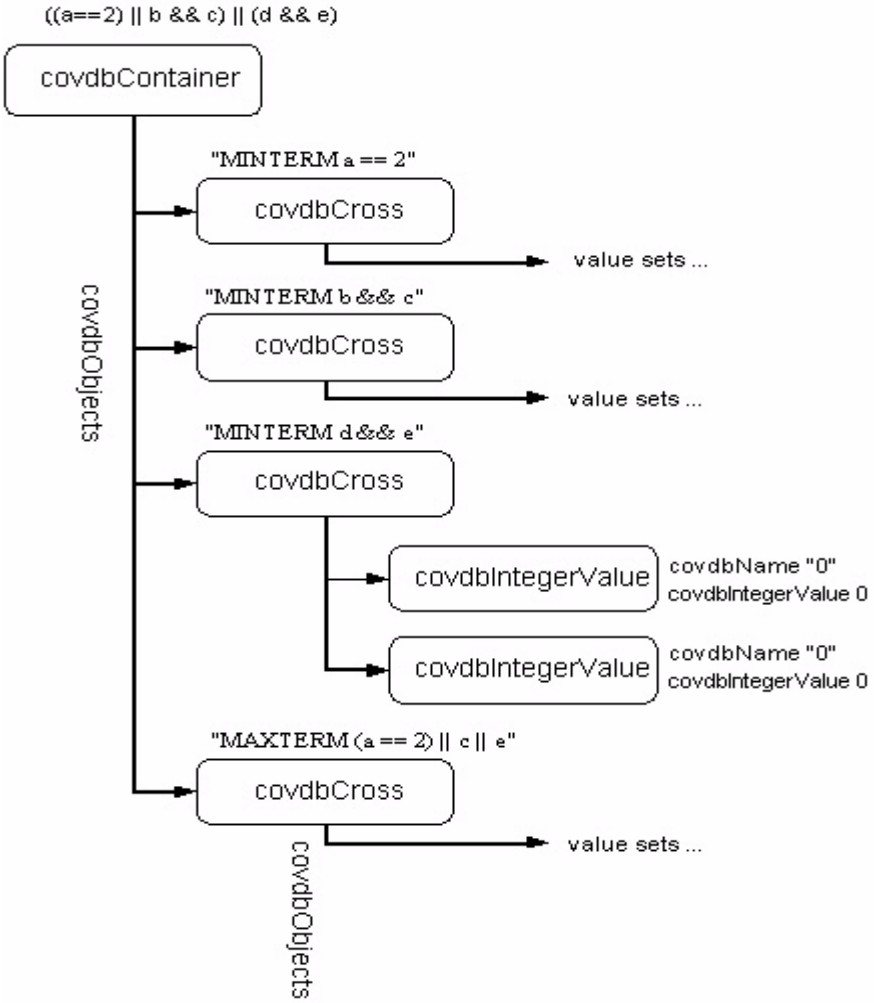
The SOP format of this expression is the disjunction of the following minterms:

```
(a == 2) ||
(b && c) ||
(d && e)
```

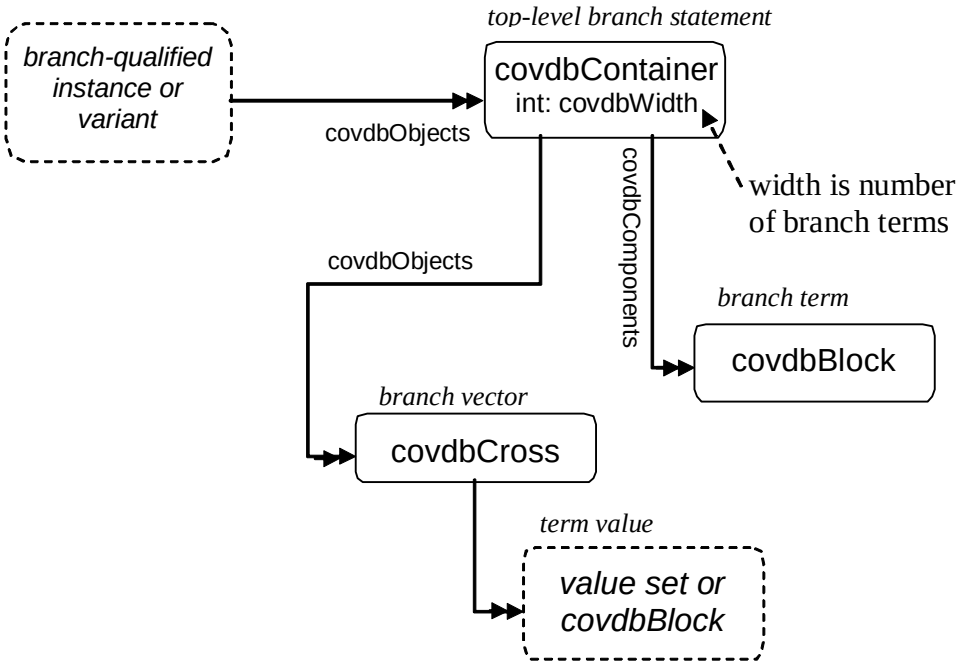
Similarly, the product-of-sums (POS) format of this expression is the conjunction of the following maxterms:

```
((a == 2) || c || e) &&
((a == 2) || b || e) &&
((a == 2) || c || d) &&
((a == 2) || b || d)
```

UCAPI organizes the SOP (minterms) and POS (maxterms) expressions into containers under the expression, as shown in the following figure.



As described in “Value Set” on page 1-13, values are not always integers. In condition coverage, “don’t care” values may be present in vectors, and will have the value type `covdbScalarValue` and a `covdbValue` of `covdbValueX`.



Branch Coverage

Branch coverage monitors the execution of conditional statements in your design. Conditional statements include `if/else` statements, `case` statements, and the ternary operator “`?:`”

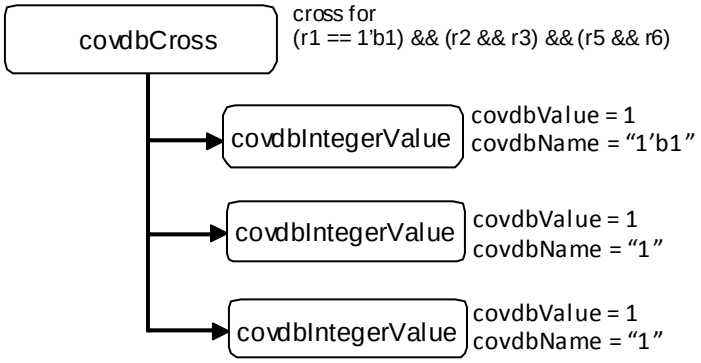
Branch coverage differs from condition coverage in that branch coverage does not track all the ways in which a Boolean condition can be true or false – it only tracks whether it is true or false. Branch coverage also tracks all conditions, rather than the subset of

complex conditions that condition coverage monitors. Branch coverage also tracks `case` statements, which condition coverage ignores.

The coverable objects for branch coverage are all crosses of value sets. For example, consider a branch that is covered when the following expression is true:

```
(r1 == 1'b1) && (r2 && r3) && (r5 && r6)
```

That branch would be modeled as a cross of these three terms, each of which is a `covdbIntegerValue` value set:

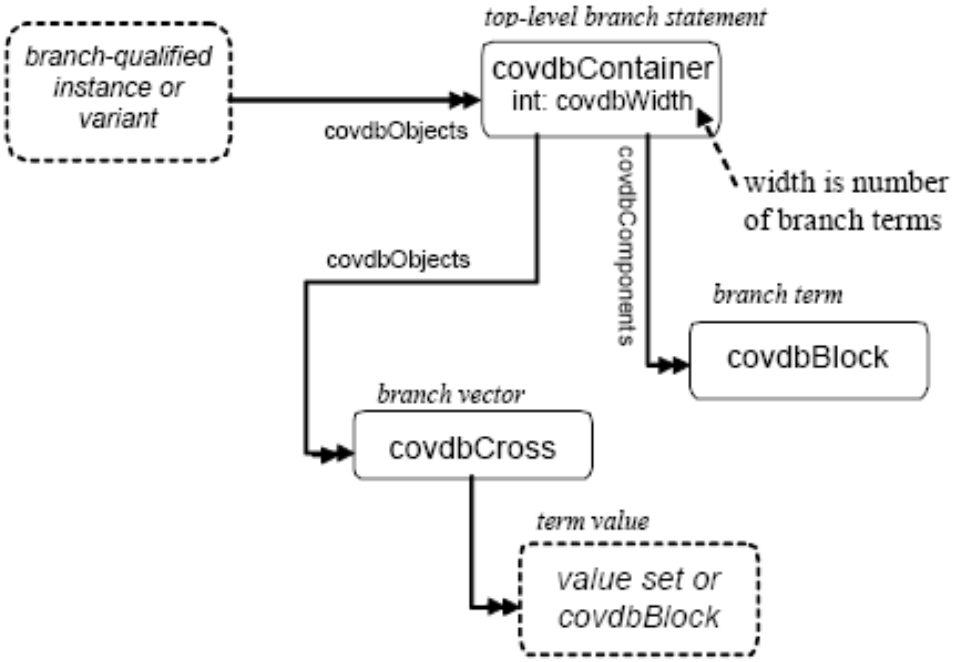


Branch coverage is organized into disjoint sections of each always/initial block or process by the top-level branch statements. For example, the following code has two top-level branch statements:

```
1 initial
2 begin
3   case (r1)          First top-level statement
4           1'b1 : if (r2 && r3)
5                    r4 = 1'b1;
6           1'b0 : if (r7 && r8)
7                    r9 = r10 ? 1'b0 : 1'b1;
```

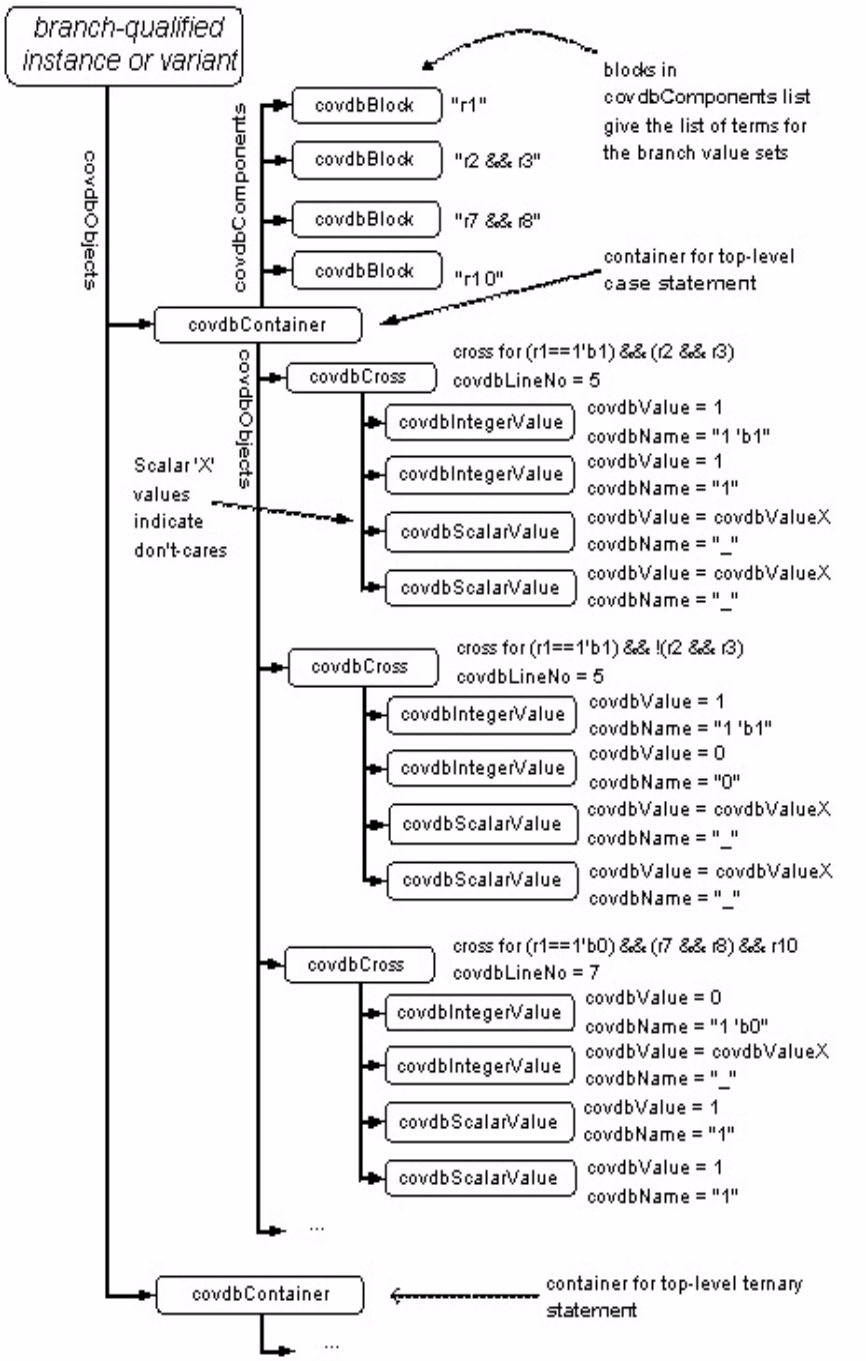
```
 8            1'bx : $display("x");
 9            default : $display("no op");
10    endcase
11
12    r9 = (r10 && r11) ? 1'b0 : 1'b1; Second top-level
                                       statement
13 end
```

Since the branches in each top-level statement do not interact, UCAPI collects their branches into separate containers underneath the source region handle. The model for objects inside a branch coverage region is as follows:



For the sample code shown above, the UCAPI model is:

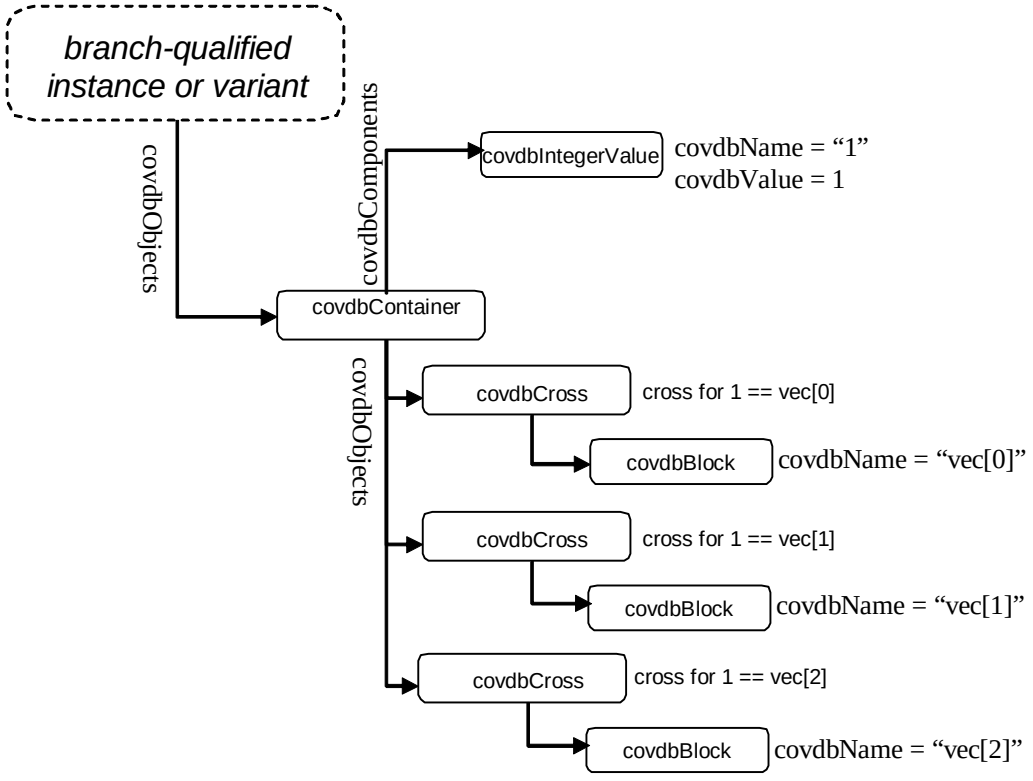
.


branch-qualified instance or variant
covdbObjects
covdbComponents
covdbBlock "r1"
covdbBlock "r2 && r3"
covdbBlock "r7 && r8"
covdbBlock "r10"
blocks in covdbComponents list give the list of terms for the branch value sets
container for top-level case statement
covdbContainer
covdbObjects
covdbCross
cross for (r1==1'b1) && (r2 && r3)
covdbLineNo = 5
covdbIntegerValue
covdbValue = 1
covdbName = "1 'b1"
covdbIntegerValue
covdbValue = 1
covdbName = "1"
Scalar 'X' values indicate don't-cares
covdbScalarValue
covdbValue = covdbValueX
covdbName = "_"
covdbScalarValue
covdbValue = covdbValueX
covdbName = "_"
covdbCross
cross for (r1==1'b1) && !(r2 && r3)
covdbLineNo = 5
covdbIntegerValue
covdbValue = 1
covdbName = "1 'b1"
covdbIntegerValue
covdbValue = 0
covdbName = "0"
covdbScalarValue
covdbValue = covdbValueX
covdbName = "_"
covdbScalarValue
covdbValue = covdbValueX
covdbName = "_"
covdbCross
cross for (r1==1'b0) && (r7 && r8) && r10
covdbLineNo = 7
covdbIntegerValue
covdbValue = 0
covdbName = "1 'b0"
covdbIntegerValue
covdbValue = covdbValueX
covdbName = "_"
covdbScalarValue
covdbValue = 1
covdbName = "1"
covdbScalarValue
covdbValue = 1
covdbName = "1"
...
covdbContainer
container for top-level ternary statement
...

Note that in Verilog, case alternatives do not have to be constant values. For example, the following is legal:

```
case(1)
    vec[0]: …
    vec[1]: …
    vec[2]: …
endcase
```

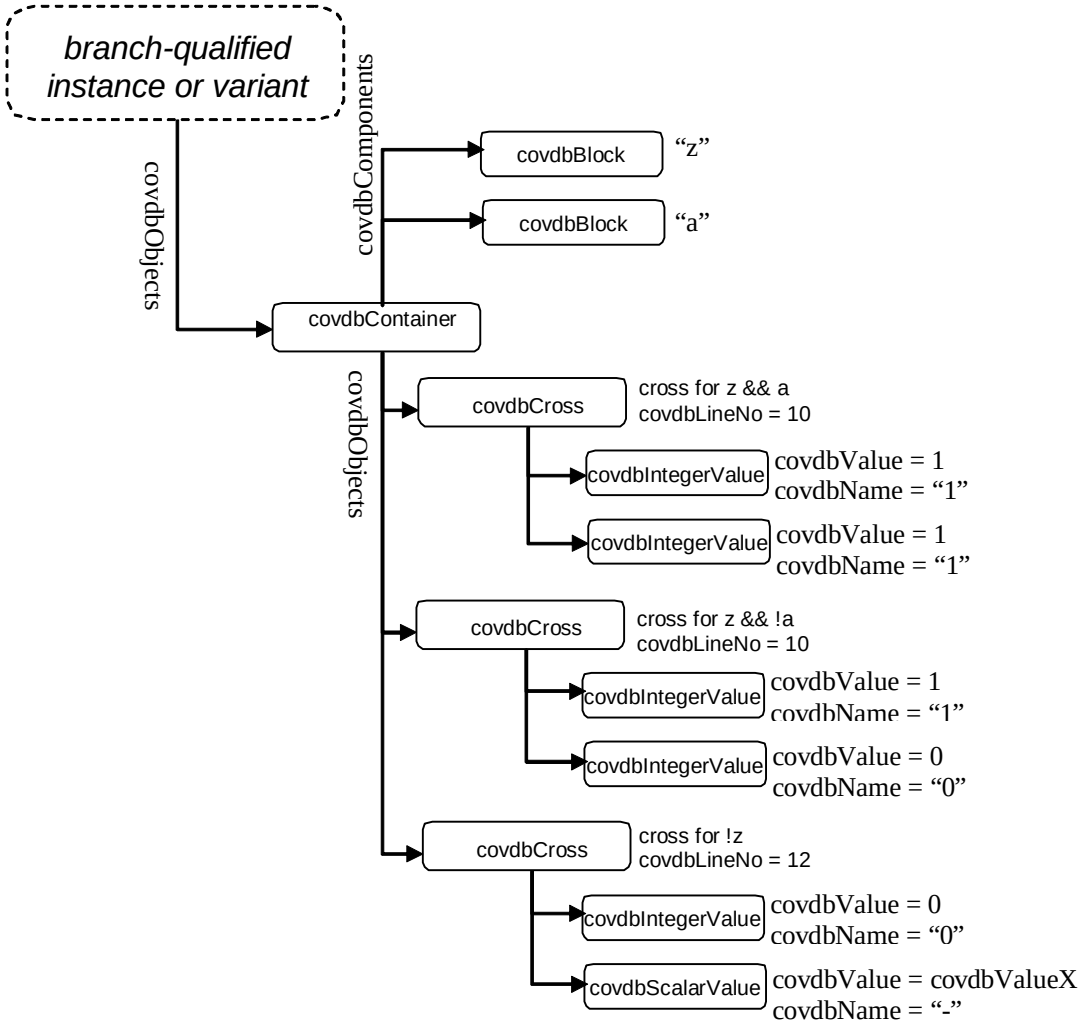
In this example, the case alternatives are not constant values, and therefore cannot be modeled as value sets. In this case, we use `covdbBlock` handles for the case alternatives. Note that the argument to the case conditional ("`1`") is when possible modeled as a value set:



Ternary operators are also monitored by branch coverage. In UCAPI, a ternary conditional is treated the same as an "`if/else`" condition. For example:

```
9 if (z)
10     x <= a ? b : c;
11 else
12    x <= 1'b0;
```

Turns into this in UCAPI:



Finite State Machine Coverage

Finite State Machine (FSM) coverage models three distinct types of coverable objects.

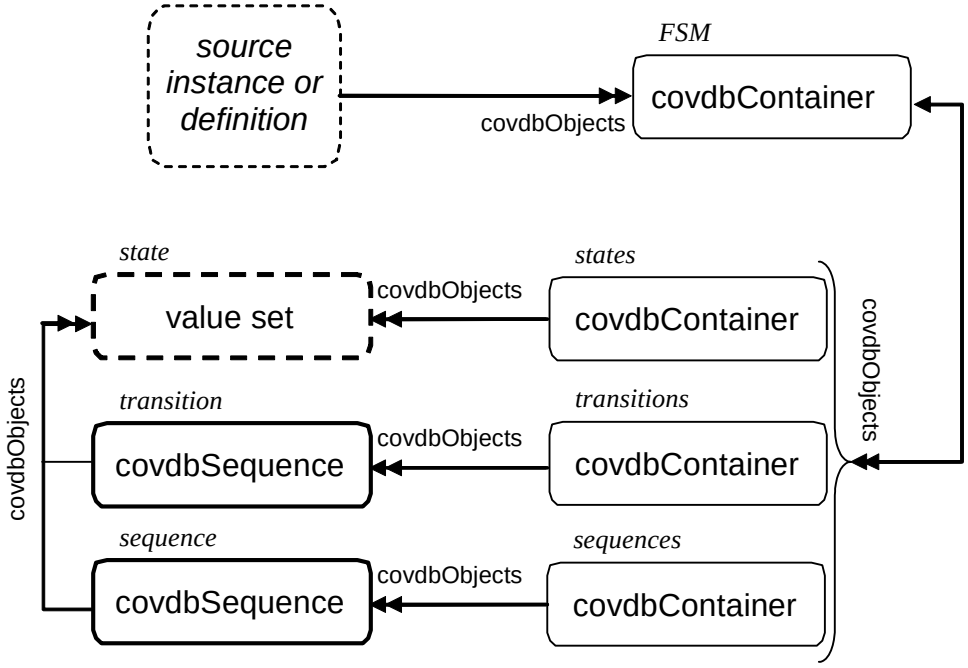
Whether the FSM:

- entered each possible state (state coverage)
- state followed each possible state transition (transition coverage)
- went through all possible sequences of states from the initial state(s) (sequence coverage)

In UCAPI, state coverage is modeled with value sets, and transition and sequence coverage are modeled with `covdbSequence` objects containing value sets. The `covdbCoverable` property of the FSM container is the total number of states, transitions, and sequences for that FSM.

The `covdbFileName` and `covdbLineNo` properties may be read from the FSM handles (of type `covdbContainer`), state value sets, and transition `covdbSequence` handles. No other FSM objects have source information.

State and sequence objects are all considered to be used for information only and do not count for `covdbCoverable` and `covdbCovered` numbers for FSM coverage. Each coverable object for states and sequences will have `covdbExcluded` set in its `covdbCovStatus` property.



For example, consider an FSM encoded as:

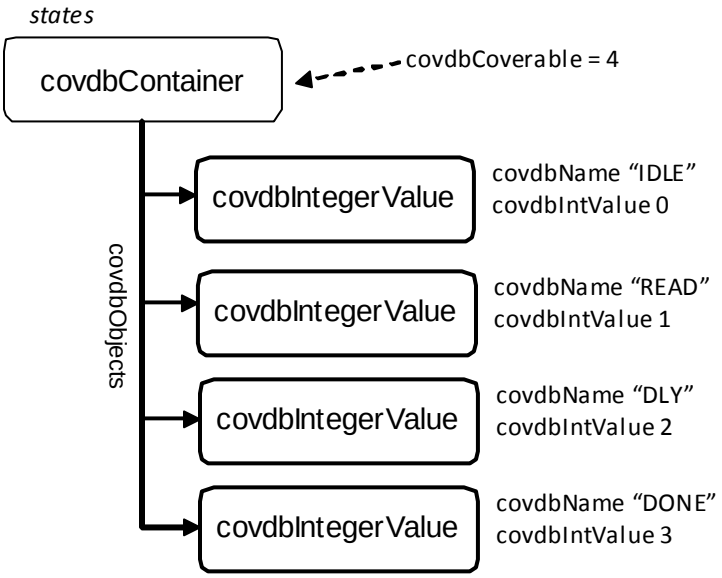
```
always @(posedge clk) begin
     case (state)
          `IDLE: if (go) next = `READ else next = `IDLE;
          `READ: if (go) next = `DLY else next = `IDLE;
          `DLY: if (!go) next = `IDLE;
                else if (ws) next = `READ;
                else next = `DONE;
          `DONE: next = `IDLE;
     endcase
end
```

The FSM has four states and seven possible transitions:

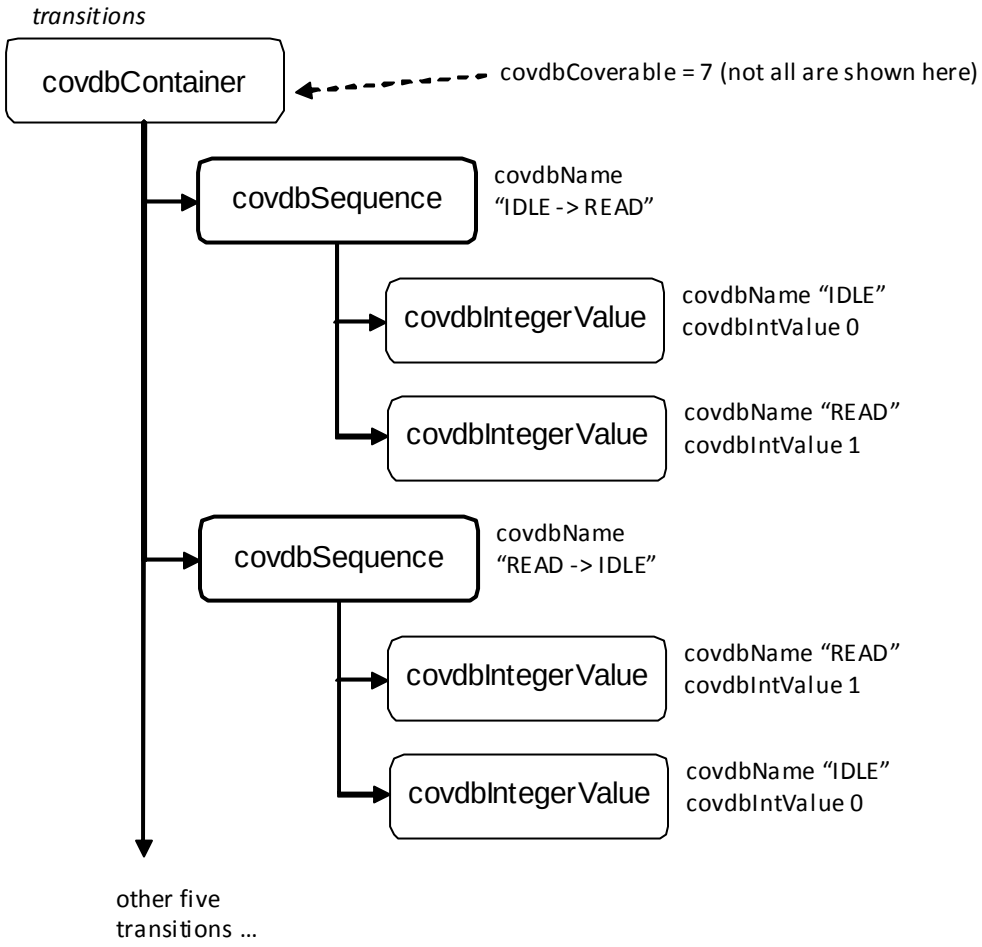
```
States:
IDLE
READ
DLY
DONE
```

```
Transitions:
IDLE->READ
READ->IDLE
READ->DLY
DLY->IDLE
DLY->READ
DLY->DONE
DONE->IDLE
```

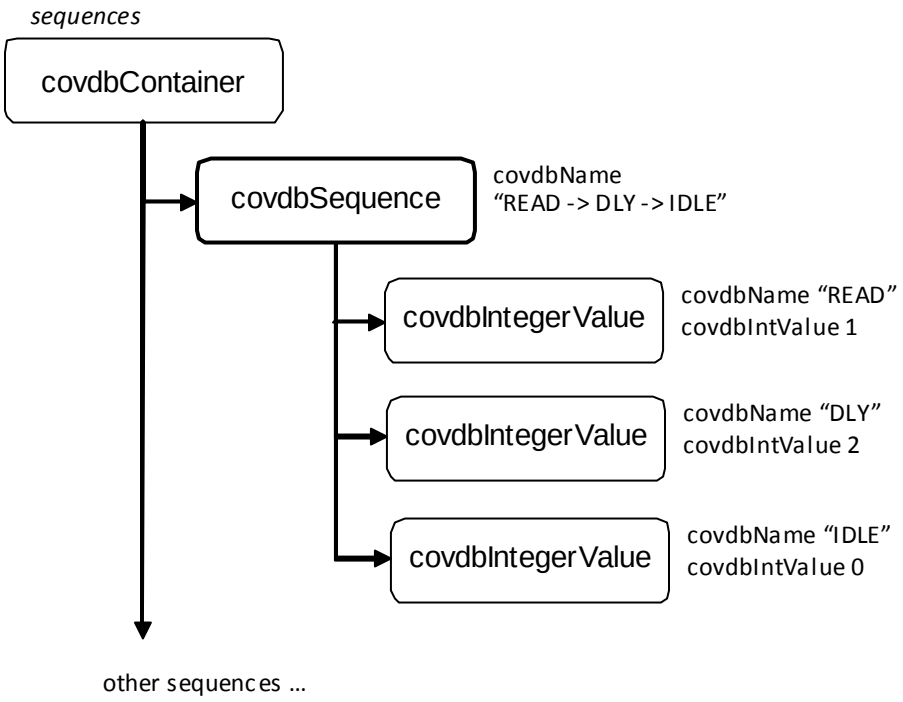
The states in UCAPI will look like:



The transitions are represented as `covdbSequences`, as shown in the following illustration:



The FSM’s sequences are `covdbSequence` objects, but with potentially more than two objects. For example, the sequence `READ -> DLY > IDLE` looks like this:



Toggle Coverage

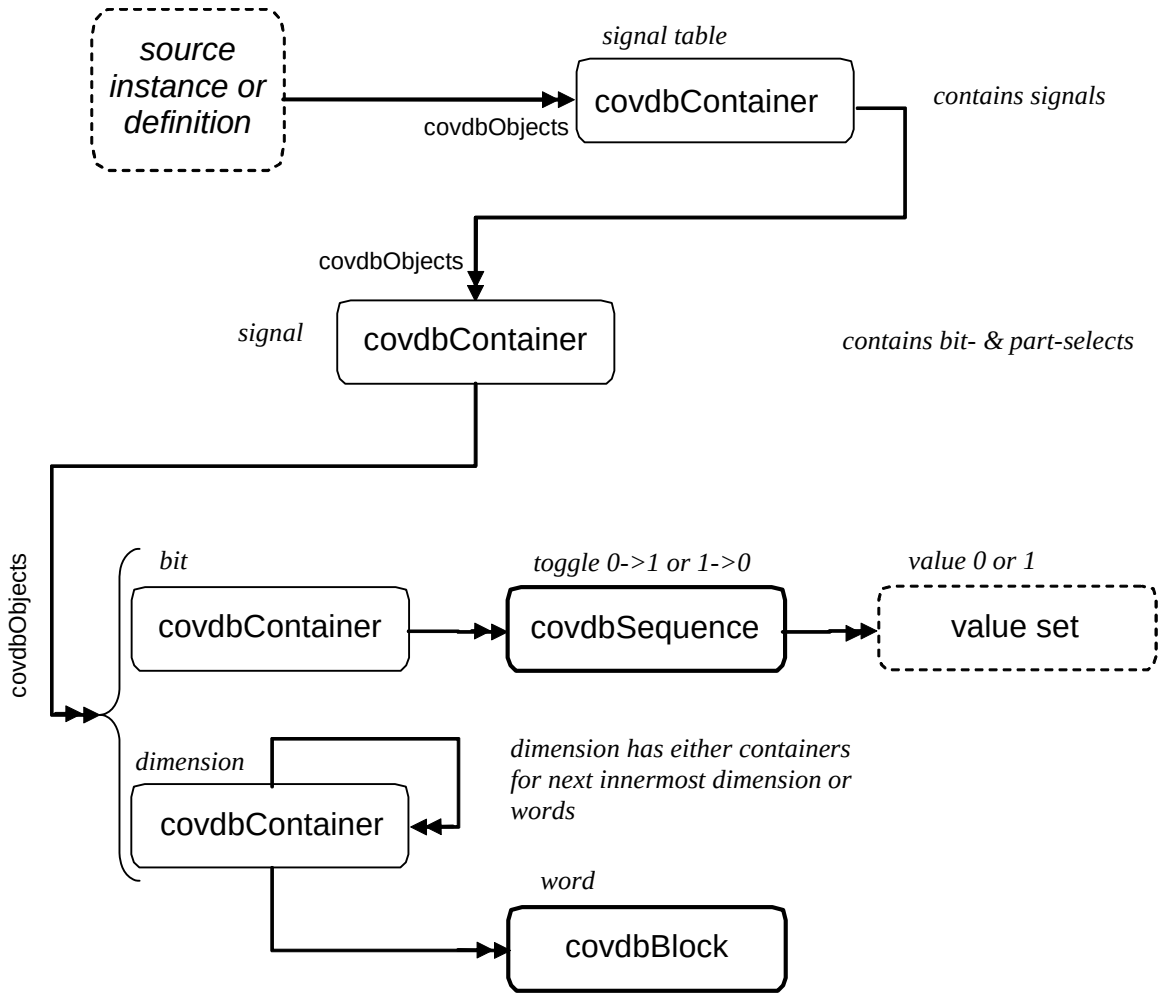
Toggle coverage in UCAPI is organized into tables, each of which contain signals, which are modeled as `covdbContainers`.

The tables are groupings of signals by category, such as "signal" and "port". The `covdbName` property of the table container is the category of all signals in that table (e.g, "ports").

Each signal's container has a list of sub-containers in it, representing bits of that signal. Each sub-container has two `covdbSequences` in it, one representing the transition from 0 to 1, and one representing the transition from 1 to 0.

Signals and MDA dimension container objects for toggle coverage have the annotations "`lsb`" and "`msb`" set on them. These return a string given the least-significant bit and most-significant bit indices, respectively, as strings.

The `covdbCoverable` of a given signal for toggle coverage is twice the number of distinct bits for that signal – that is, the `covdbCoverable` is the number of `covdbSequences` under the container for the signal. The `covdbWidth` property is the total number of bits.



For example, consider a module “M” with these signals:

```
reg x[3:0];
input y;
wire y;
```

Now assume the toggle coverage data is as shown in the following table:

Table 2-1

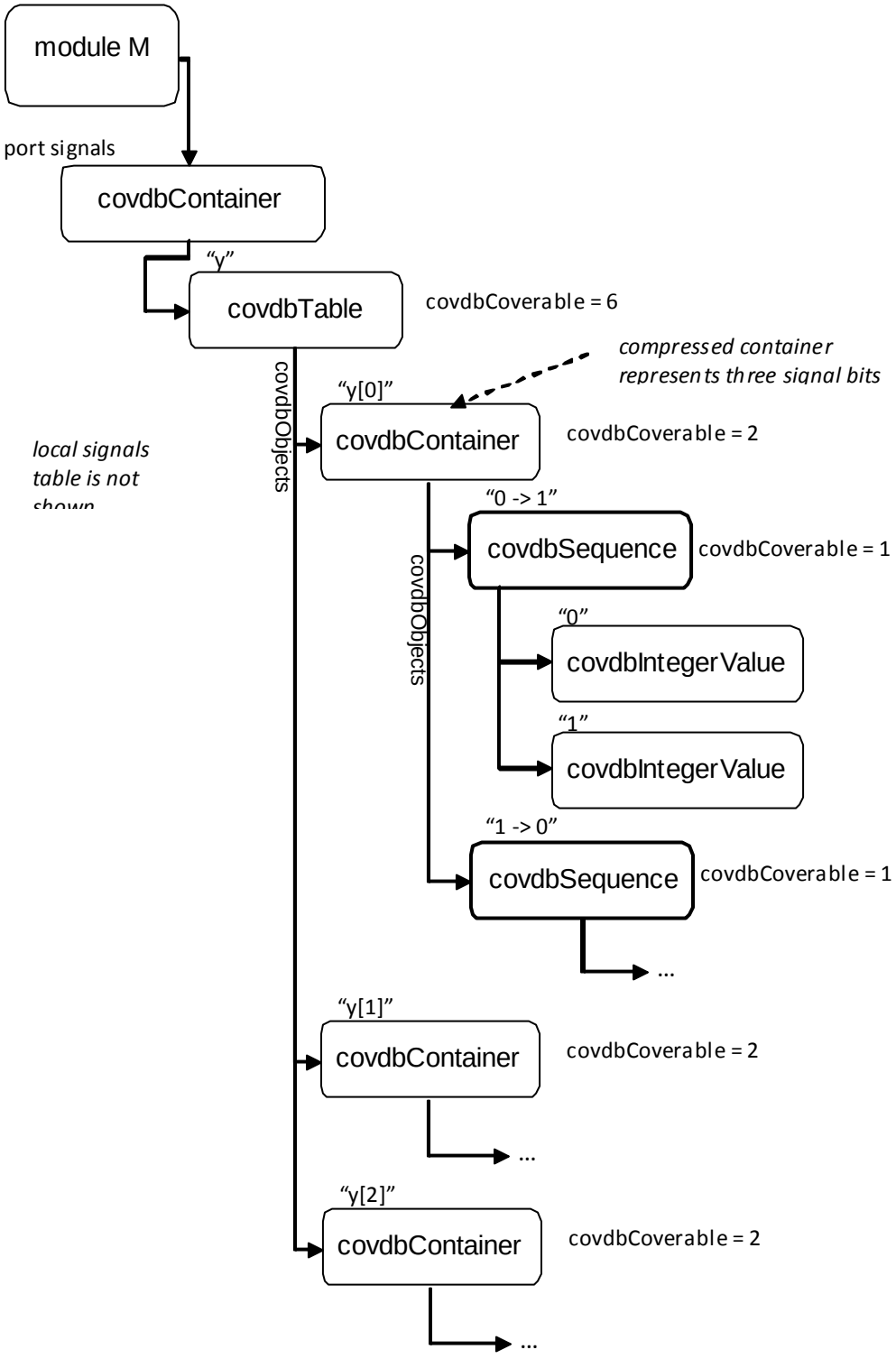
Signals	0->1	1->0
x[3]	Y	N
x[2:1]	Y	Y
x[0]	N	Y
y	Y	N

The toggle coverage structure would look like the following diagram. (In this figure, only some of the `covdbSequence` and value set children are shown.)

Now consider a case with a multiple-dimension array signal:

```
reg [2:0] mda[3:0][1:0];
```

Then the data would look like the following (only the handles containing `mda[1][0][2]` are shown):


module M
port signals
covdbContainer
"y"
covdbTable
covdbCoverable = 6
compressed container represents three signal bits
covdbObjects
"y[0]"
covdbContainer
covdbCoverable = 2
local signals table is not shown
"0 -> 1"
covdbSequence
covdbCoverable = 1
covdbObjects
"0"
covdbIntegerValue
"1"
covdbIntegerValue
"1 -> 0"
covdbSequence
covdbCoverable = 1
...
"y[1]"
covdbContainer
covdbCoverable = 2
...
"y[2]"
covdbContainer
covdbCoverable = 2
...

Assertion Coverage

Assertion coverage includes coverage information about assertions, cover properties and sequences. The `covdbName` of the assertion coverage metric handle is **Assert**.

Assertion coverage objects may appear directly in source definitions and instances (inlined assertions) or in special source regions called **units**, which are test-qualified source regions (see “Test-qualified Source Regions” on page 1-34). Inlined assertions are metric-qualified but not test-qualified, and they may be read without loading any test. Assertions in units (and the list of units itself) are not available except with respect to a given test handle.

In the current version, inlined assertions are accessible through the interface as described, but will only appear in the design as tests are loaded.

There are three types of assertion coverage objects: assertions, cover properties, and cover sequences. The handles for these different assertion coverage objects are organized into `covdbContainers` in each source definition and instance.

Assertions are modeled in UCAPI as `covdbContainers`. The `covdbBlock` objects in these containers represent the different type of information collected. For example, there is a `covdbBlock` representing success, a `covdbBlock` for failures, and another for attempts. The `covdbName` of each `covdbBlock` indicates what it represents.

All but one of the `covdbBlock` objects will have the `covdbStatusExcluded` flag set, indicating they should be ignored for coverage. For example, for assertions there will be a block for

attempts. The number of attempts is not used to compute the coverage score, so this block is marked excluded. The assertion is covered if its non-excluded `covdbBlock` object is covered.

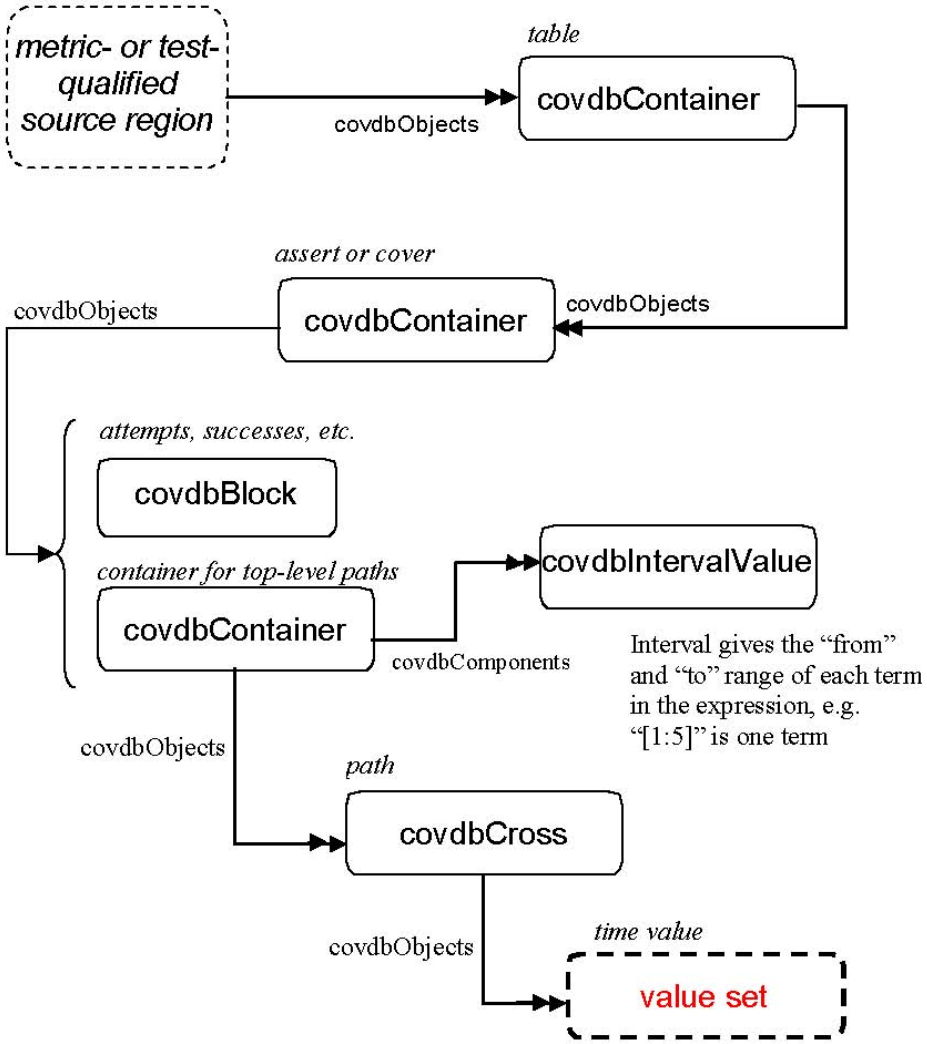
The `covdbCovCount` may be read from each of these `covdbBlock` objects to get the number of times each type of thing occurred. Non-excluded `covdbBlocks` are covered if their `covdbCovCount` is greater than or equal to the assertion's `covdbCovCountGoal`.

Some assertions will have assertion path coverage collected for them. When this information is collected at runtime, the affected assertions will have a more detailed structure. In addition to the blocks for attempts, successes, etc., assertions that have assertion path coverage information will also have crosses in them. Each cross is a path through the assertion (there are only multiple crosses if there is a top-level "`or`" in the assertion expression. See the following example for more on how this works).

SystemVerilog assertions and cover directives will always be in the assert-qualified module definitions and instances in the `covdbObjects` list. OpenVera Assertions are contained in test-qualified source regions, as discussed in "Test-qualified Source Regions" on page 1-34.

The `covdbFileName` and `covdbLineNo` properties may be read from handles of type `covdbContainer` for assertion coverage. There is no source information for `covdbBlock` handles in assertion coverage.

The following diagram shows the model for assertion coverage:

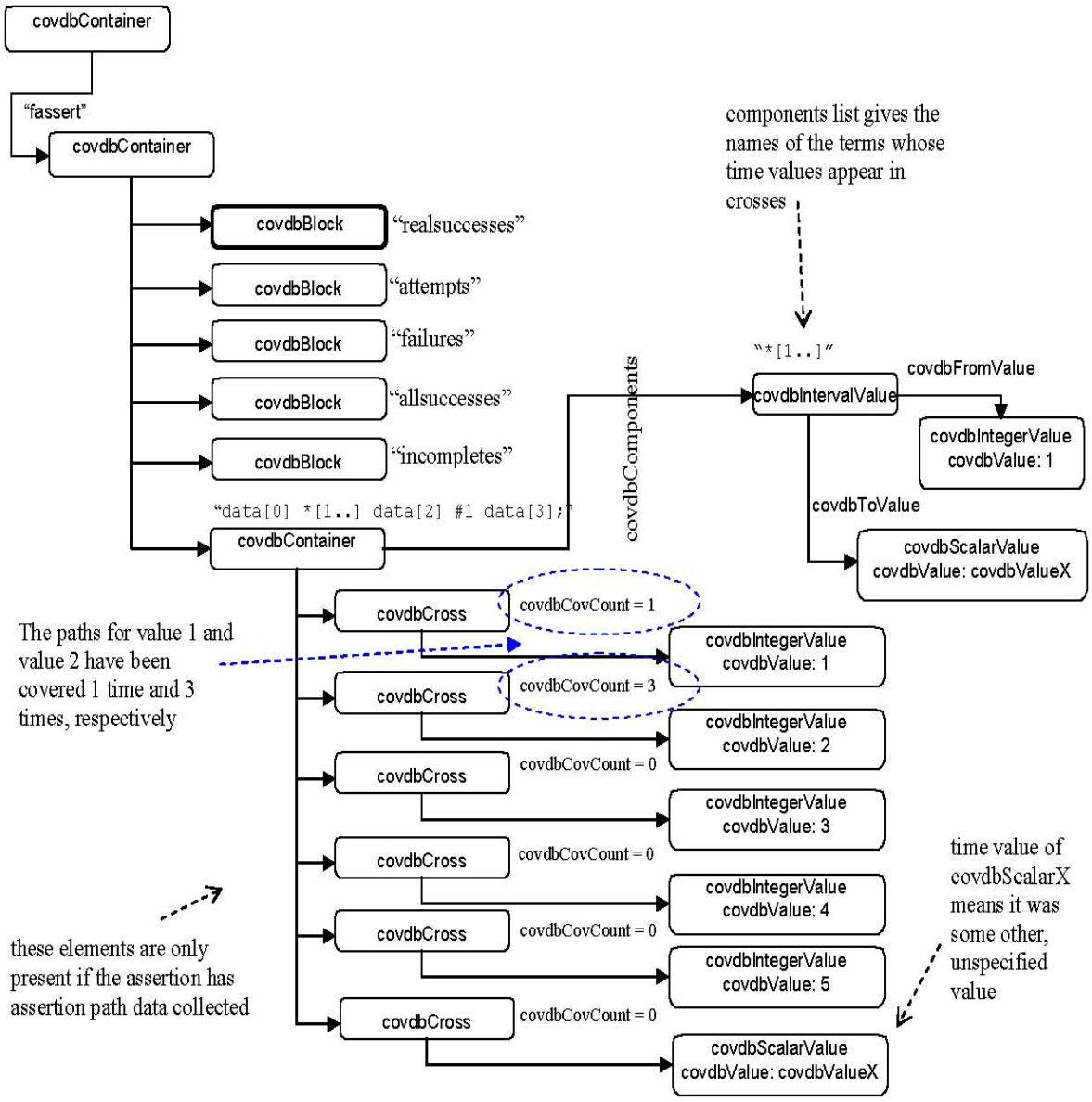


For example, consider the following assertion:

```
clock posedge clk {
  event fevent : data[0] *[1..] data[2] #1 data[3];
}
assert fassert : check(fevent);
```

The following diagram shows the UCAPI structure for this assertion with basic coverage. Note that there are five `covdbBlocks` under the assertion, but that only the `realsuccesses` block counts for coverage, since the other blocks are all marked excluded. These other blocks (e.g., `attempts`, `incompletes`) are in the database, but are never considered for coverage.

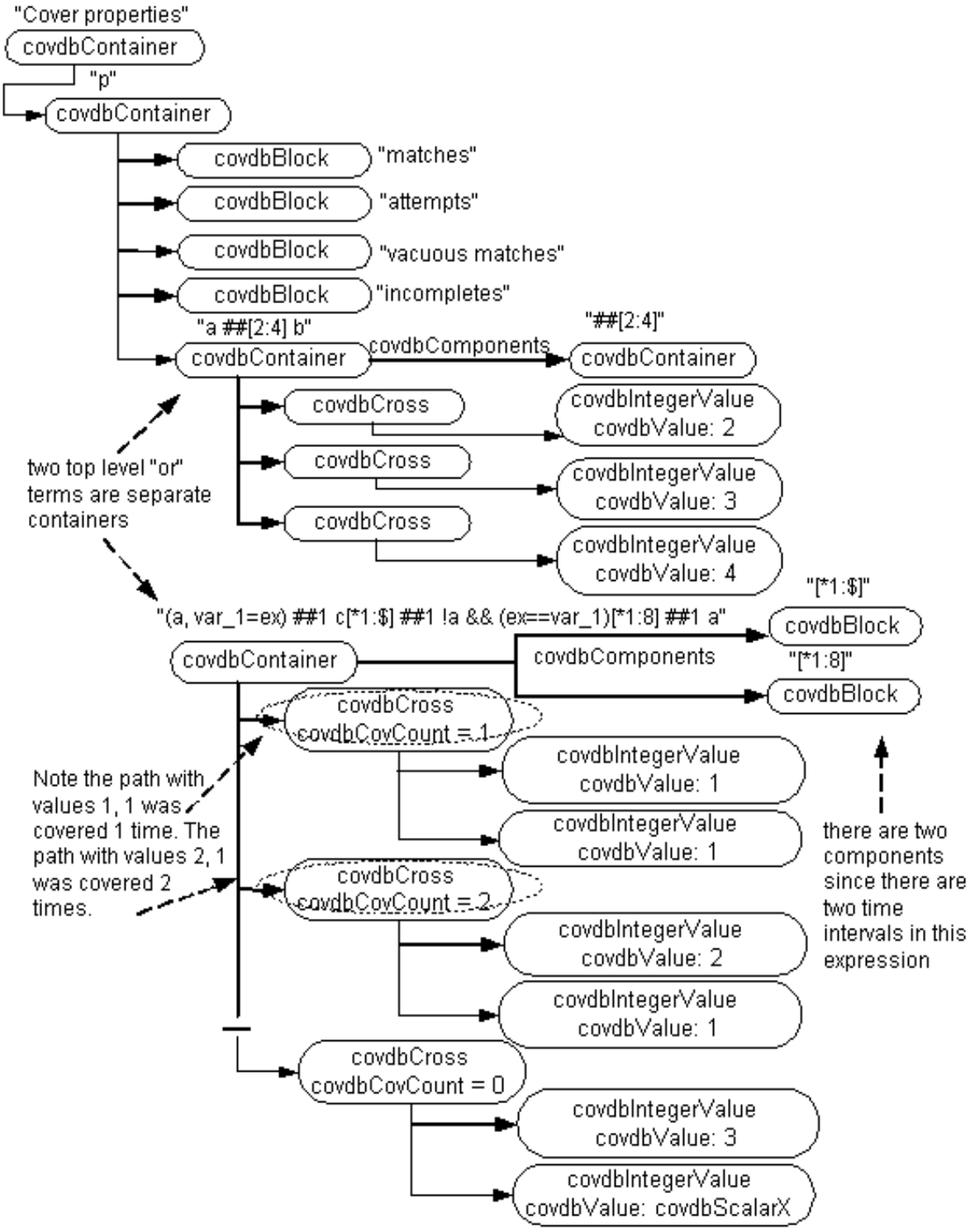
In this example that only the paths where the first range (`#[1..]`) had value `1` or `2` was the assertion covered. It was covered one time for the value `1` and three times for the value `2`. You can see this as the `covdbCovCount` of the individual `covdbCross` bin handles.



Now consider another example, a cover property with multiple time intervals in it:

```
property p;
 int var_l;
  @clk
   a ##[2:4] b or
  (a, var_l=ex) ##1 c[*1:$] ##1 !a && (ex==var_l)[*1:8]
                                                ##1 a;
endproperty
```

Since this property has a top-level `or`, it will have two containers for paths under the assertion container. Also, since the second term has two time intervals in it, the crosses contains two values each.


"Cover properties"
covdbContainer
"p"
covdbContainer
covdbBlock
"matches"
covdbBlock
"attempts"
covdbBlock
"vacuous matches"
covdbBlock
"incompletes"
"a ##[2:4] b"
covdbContainer
covdbComponents
"##[2:4]"
covdbContainer
covdbCross
covdbIntegerValue
covdbValue: 2
covdbCross
covdbIntegerValue
covdbValue: 3
covdbCross
covdbIntegerValue
covdbValue: 4
two top level "or" terms are separate containers
"(a, var_1=ex) ##1 c[*1:$] ##1 !a && (ex==var_1)[*1:8] ##1 a"
"[*1:$]"
covdbBlock
covdbContainer
covdbComponents
"[*1:8]"
covdbBlock
covdbCross
covdbCovCount = 1
covdbIntegerValue
covdbValue: 1
covdbIntegerValue
covdbValue: 1
Note the path with values 1, 1 was covered 1 time. The path with values 2, 1 was covered 2 times.
covdbCross
covdbCovCount = 2
covdbIntegerValue
covdbValue: 2
covdbIntegerValue
covdbValue: 1
there are two components since there are two time intervals in this expression
covdbCross
covdbCovCount = 0
covdbIntegerValue
covdbValue: 3
covdbIntegerValue
covdbValue: covdbScalarX

You can read the category and severity failures for a given assertion handle using the `covdb_get_annotation` function:

```
char *category = covdb_get_annotation(assertionHdl,
“FCOV_ASSERT_CATEGORY”); char *severity =
covdb_get_annotation(assertionHdl,
“FCOV_ASSERT_SEVERITY”);
```

Testbench Coverage

Testbench coverage monitors user-specified expressions during simulation. Testbench coverage data may be monitored through coverage constructs in Vera, Native Testbench (NTB), or System Verilog. The `covdbName` of the testbench coverage metric handle is Testbench.

Testbench coverage monitors three different types of coverable objects: states (value sets), transitions (modeled as `covdbSequence` objects), and crosses (`covdbCross` objects). For testbench coverage, the containers of these coverable objects are called bin tables.

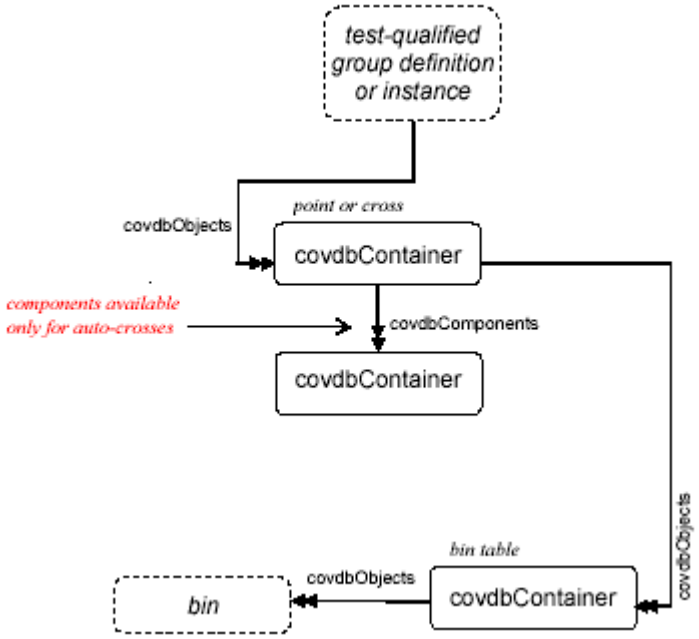
As for assertion coverage, many of the coverable objects in testbench coverage may be marked `covdbStatusExcluded`. Such objects are not used when computing coverage information such as the `covdbCovered` or `covdbCoverable` properties.

The `covdbFileName` and `covdbLineNo` properties may be read from group, group instance, point and cross declaration handles for testbench coverage. No other handles types for testbench coverage have source information.

When testbench covergroups are defined within HDL modules (for example, inside a SystemVerilog module), they are accessed as the `covdbObjects` list from the testbench-qualified HDL module handle.

The following figure shows the structure of testbench coverage within a test-qualified group or instance definition. The top-level list is containers representing point or cross declarations within the group. Each point or cross declaration has one or more bin tables, and each bin table has its list of bins.

For cross declarations, there may be a `covdbComponents` list, which will contain the list of point declarations that are being crossed.

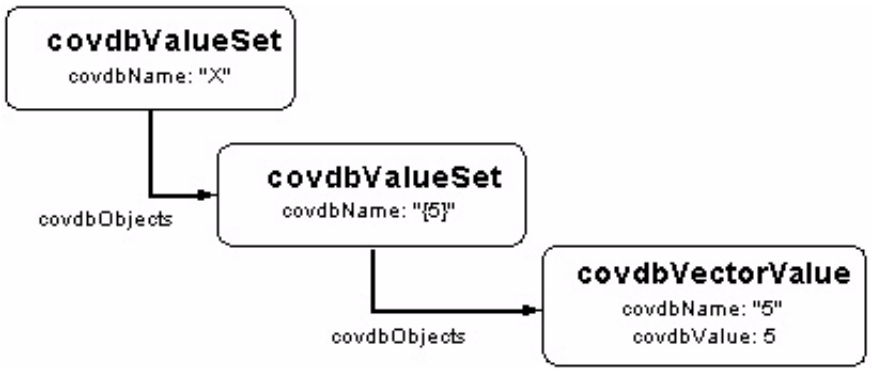


The bin handle can be different depending on the type of bin table. The types of bin tables are user-defined bin table, auto bin table, user-defined cross table, and auto cross table. The type of the table is given by its `covdbName` property.

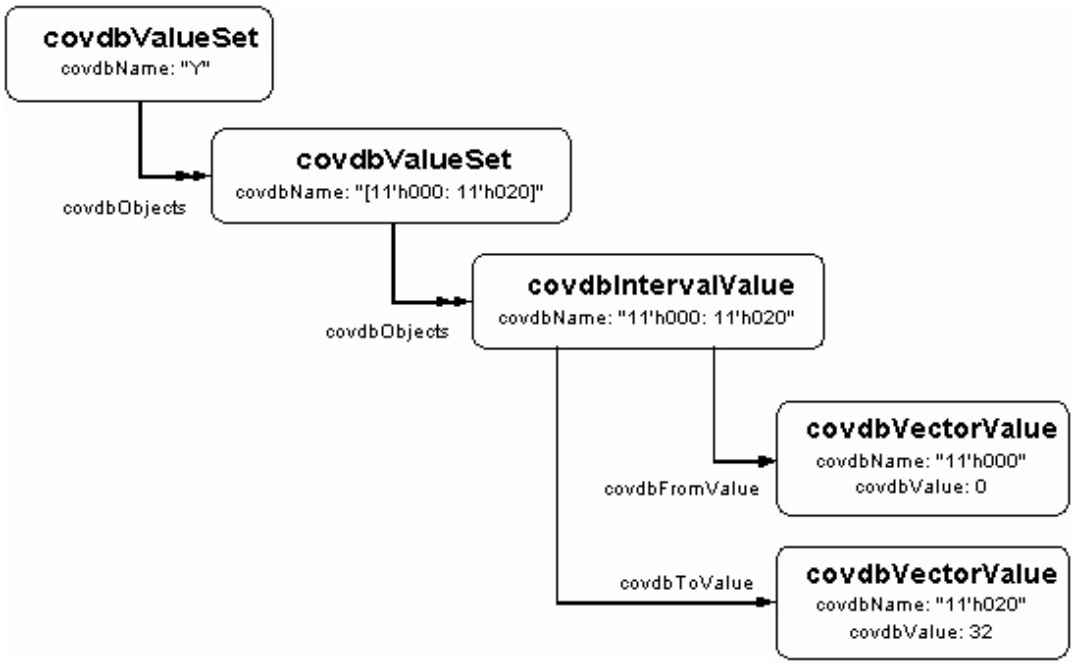
Currently there is no representation in UCAPI of the containing program for covergroups declared in programs.

Contents of the Bin Tables

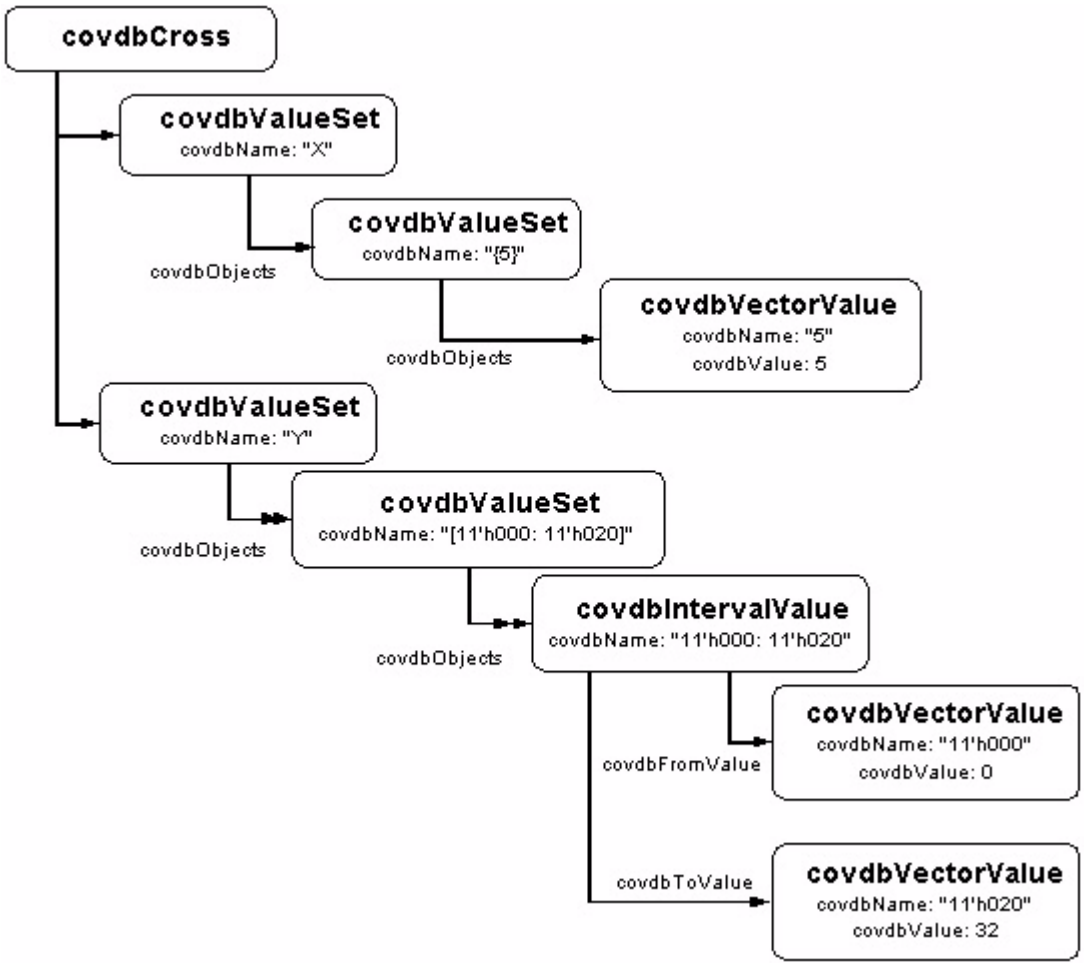
For user-defined and auto bin tables, the bins will all be value sets. For example, a bin `x` with the value `5` would be:



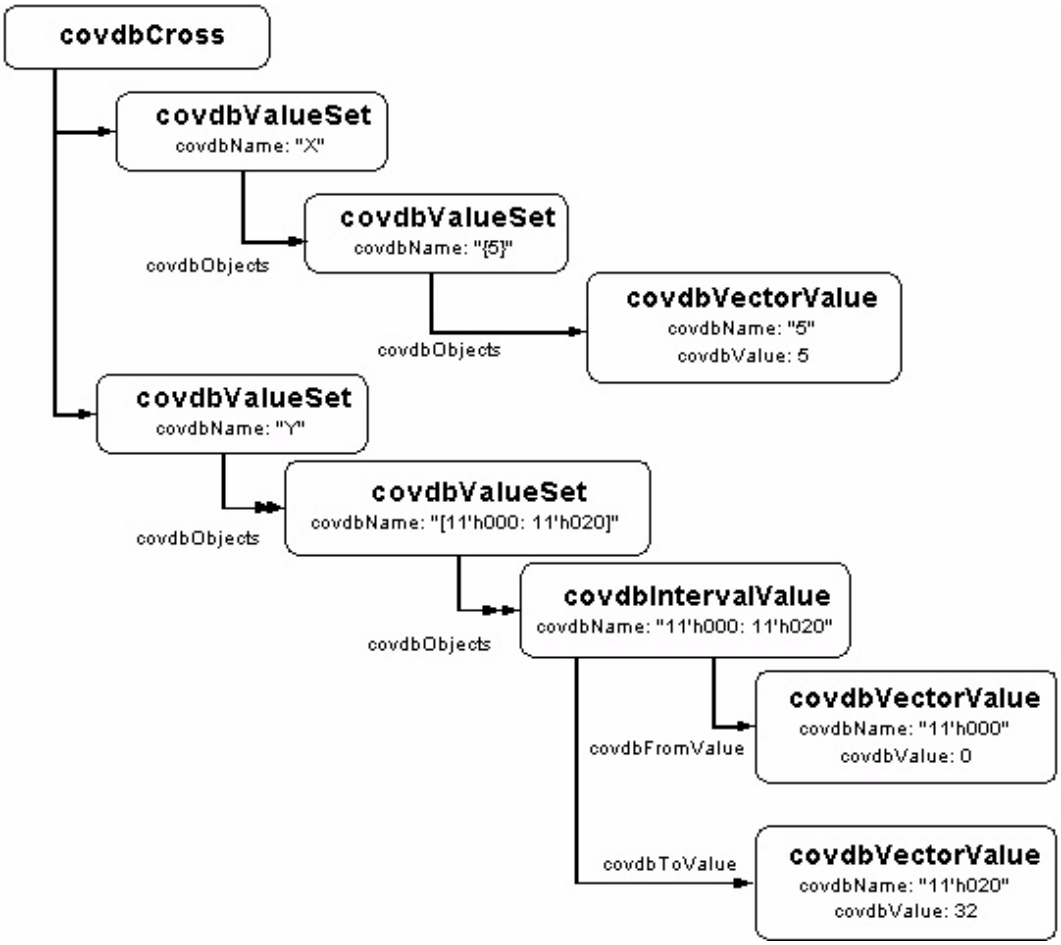
And a bin `Bin Y = [0:32]` will be represented by the following:



Crosses contain handles to the bins being crossed, so a cross of `X = 5` and `Y` in `[11'h000 : 11'h020]` would be:



User-defined cross bins are defined by the (what would otherwise be) auto-cross bins they contain. The user-defined cross bins (in the user-defined cross bins table), therefore have an extra level of structure:



The `covdbName` string property of a point container gives the name of the expression being sampled. For example, a coverpoint on variable "`myvar`" would have `covdbName = "myvar"`. A cross container has the relation `covdbComponents`, which is an iterator over the coverpoint containers that were crossed to create the cross

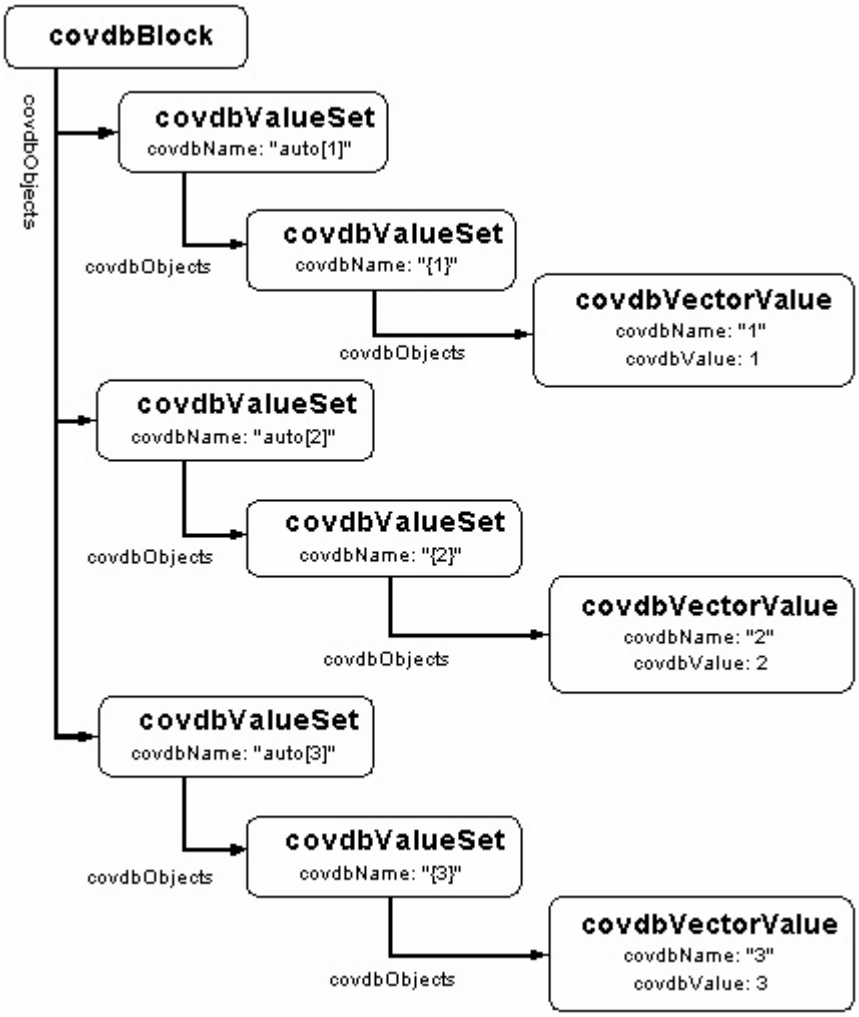
itself. Using these two methods (names for points and `covdbComponents` for crosses), applications can determine the meaning of the values given in the bins.

For example, if a coverpoint's name is `"myvar"`, then each of its value set bins will refer to a value of "`myvar`" that was observed. If a cross's components are points `x` and `y`, then the value set bins in each cross bin are bins from coverpoints `x` and `y`, respectively.

Compressed Bins

Each bin handle will have the string annotation “`COVERPOINT_BIN`” set to `1` or `0`. If it is set to `1`, it means the bin is a regular bin. If it is set to `0`, it means the bin is a compressed bin. Compressed bins have `covdbCoverable` greater than `1` (the value is the number of bins that were compressed), and contain a `covdbObjects` list of the bins that were compressed.

For example, the following bin, `auto[1:3]`, is a compressed bin made up of three bins, one each for the values `1, 2` and `3`. The compressed bin is a `covdbBlock`, and its `covdbObjects` list is the list of bins that make it up:



Limitations

Currently, only automatically-generated cross bins are represented as `covdbCross` objects. User-defined cross bins are represented as `covdbBlock` objects. All transitions are currently represented as `covdbBlock` objects.

The `covdbComponents` list is currently only available for automatically generated crosses, not for user-defined crosses.

3

Loading a Design

VCS provides coverage data within two directories, `simv.cm` and `simv.vdb`. The design data and coverage metrics collected are stored within `simv.cm`. Test related data such as assertion coverage and functional coverage are stored within `simv.vdb`. In a future version of VCS, these two directories will be collapsed into one `simv.vdb`.

Note, when using the VCS comp, designs are loaded with the `covdb_load()` function. The following function, `loadDesign`, accepts a string argument representing the path to the design and then loads the design.

```
covdbHandle loadDesign(char *design)
{
   covdbHandle designHandle = covdb_load(covdbDesign, NULL,
                                         design);
if(!designHandle)
       printf("Error: could not load design %s\n", design);
    else
```

```
      printf("Info: loaded design %s\n", design);
   return designHandle;
}
// example of calling the function:
covdbHandle designHandle = loadDesign("simv");   //
```

Note:

A handle returned by `covdb_load` is persistent until unloaded by `covdb_unload`.

Available Tests

Loading a design does not load the test(s) associated with a design. Given a valid design handle, the available tests can be accessed.

Calling `covdb_iterate()` with `covdbAvailableTests` returns an iterator to a list of the testnames associated with the design. These tests are not automatically loaded; they must be loaded with one of the API functions. The following C function shows the available tests associated with a design handle.

```
void showAvailableTests(covdbHandle designHandle)
{
   covdbHandle availableTestName availableTestNameIterator;
   availableTestNameIterator = covdb_iterate(designHandle,
                                    covdbAvailableTests);
    printf("Design: %s has the following tests:\n",
              covdb_get_str(designHandle, covdbName));
    while(availableTestName =
                  covdb_scan(availableTestNameIterator))
     printf("\ttest: %s\n", covdb_get_str(availableTestName,
                                          covdbName));
    covdb_release_handle(availableTestNameIterator);
}
```

Loading Tests

Tests are not automatically loaded with a design. Once a design handle has been obtained, the handle is used to access the list of test names and then load one or more of the tests. A single test is loaded with `covdb_load` while two or more tests can be loaded with successive calls to `covdb_loadmerge`.

Loading a single test requires that a valid design handle and test name string be available:

```
/* load a single test from a design */
covdbHandle testHandle;
testHandle = covdb_load(covdbTest, designHandle, testName);
if(!testHandle) {
  printf("UCAPI Error: could not load test %s\n", testName);
   exit(1);
}
```

Loading and merging two or more tests first requires the loading of a single test with `covdb_load` followed by multiple calls to `covdb_loadmerge` to load and merge the additional tests' data. The following example shows a function used to load all the available tests related to a design. It is assumed that all of the test data is contained in one location with the design data.

```
covdbHandle loadTests(covdbHandle designHandle)
{
 covdbHandle availableTestName; availableTestNameIterator;
   covdbHandle mergedTest;

   // get an iterator to the list of availableTests
  availableTestNameIterator = covdb_iterate(designHandle,
                                    covdbAvailableTests);

   // Using the iterator get the first available test
   availableTestName =
```

```
covdb_scan(availableTestNameIterator);

   // load the first test
   mergedTest = covdb_load(covdbTest, designHandle, ,
                           availableTestName);
   if(mergedTest)
      printf("Successfully loaded test %0s\n",
              covdb_get_str(availableTestName, covdbName);
   else
   {
      printf("Unable to load test\n");
      return NULL;
   }

   // using loadmerge load and merge the remaining tests

   while((availableTestName =
        covdb_scan(availableTestNameIterator)))
   {
      mergedTest = covdb_loadmerge(covdbTest, mergedTest,
                                   availableTestName);
      if(!mergedTest)
         printf("Could not load test %s\n",
          covdb_get_str(availableTestName, covdbName));
      else
        printf("Successfully loaded and merged test %s\n",
               covdb_get_str(availableTestName, covdbName);
   }
   covdb_release_handle(availableTestNameIterator);
   return mergedTest;
}
```

The UCAPI method `covdb_release_handle()` function is called to release the iterator `availableTestNameIterator`. This is done because a handle returned from a `covdb_iterate()` call is persistent. Releasing the handle frees the associated memory and prevents a memory leak.

Determining Which Test Covered an Object

This section describes how to use UCAPI to find which test covered a given coverable object. When you merge two or more tests in UCAPI, the information about which test covered which object is retained. You can recover it by using the `covdbTests` relation from a coverable object handle.

For example:

```
design = covdb_load(covdbDesign, NULL, "mysimv");
test = covdb_load(covdbTest, design, "mysimv/test1");
test = covdb_loadmerge(covdbTest, test, "mysimv/test2");
…
tstsIter = covdb_qualified_object_iterate(binHdl, groupHdl,
                                          testHdl,covdbTests);
while((testCt = covdb_scan(tstsIter))) {
   printf("Bin %s was covered by test %s %d times\n",
     covdb_get_str(binhdl, covdbName),
     covdb_get_str(testCt, covdbName),
     covdb_get(testCt, grpHdl, testHdl, covdbValue));
     }
   covdb_release_handle(tstsIter);
```

The handle returned by `covdb_scan` for `testsIter` is a `covdbIntegerValue` handle. Its `covdbName` is the name of the test that was loaded/loadmerged. Its `covdbValue` is the hit count for `binHdl` for that test. For example, if a bin "`b1`" had the following hits in these different tests:

```
testhits
mysimv/test15
mysimv/test20
mysimv/test32
mysimv/test40
```

Then the above code would print the following for `b1`:

```
Bin b1 was covered by test mysimv/test1 5 times
Bin b1 was covered by test mysimv/test3 2 times
```

The tests that did not cover the bin are not given in the iterator. All the metrics and modes do not track the hit counts. Some will only record that a given object was covered or not covered in a given test. In this case, each test in the list will have a count value of 1 or it will not be in the list.

Metrics or modes that do not support this relation will return a NULL iterator handle when `covdb_qualified_iterate` is called. However, no error will be flagged. Once a merged test handle is saved using c`ovdb_save`, the information about which of the tests that were loaded or loadmerged covered which object, is discarded by default. That is, if you then reload the saved test, no data about the individual tests that were merged to create it will have been retained. You can change this behavior by using `covdb_configure` to set `covdbKeepTestInfo` to true before you load the initial design as follows:

```
covdb_configure(design, covdbKeepTestInfo,
                “true”);
```

Exclusion

UCAPI allows you to mark some objects as excluded, which means they are ignored when computing the `covdbCoverable` and `covdbCovered` properties of handles - they are excluded from the computation of the coverage score. This section describes how objects and regions can be excluded.

Hierarchy Exclusion Files

UCAPI supports hierarchy exclusion files, which allow you to specify source regions, design hierarchies, and some individual objects for exclusion or explicit inclusion for coverage. For example, a hierarchy exclusion file with the following content would exclude the sub-hierarchy of the design rooted at the path `top.d1`:

```
-tree top.d1
```

See the *VCS Coverage Metrics User Guide* for the complete description of the hierarchy exclusion file syntax.

Hierarchy exclusion files are loaded using the `covdb_loadmerge` function with an already-loaded design handle:

```
covdb_loadmerge(covdbHierFile, designHdl, filename);
```

Exclusion by Object Handle

Applications may set the `covdbExcludedAtReportTime` flag using the `covdb_set` function:

```
status = covdb_get(obj, region, test, covdbCovStatus);
status = status |= covdbStatusExcludedAtReportTime;
covdb_set(obj, region, test, covdbCovStatus, status);
```

When an object is marked excluded, it no longer contributes its `covdbCoverable` (or covdbCovered) count to the count of any containing object or region. For example, if you exclude a statement in a basic block, it lowers the `covdbCoverable` count of the basic block by 1.

Exclusion bits set using the `covdb_set` function are not saved to the main UCAPI database or to test files – they are saved to a separate exclusion file. Exclusion is saved using the `covdb_save_exclude_file` function:

```
covdb_save_exclude_file(testHdl, "myexcludes.el", "w");
```

Either a design handle or a test handle must be given as the first argument. If the first argument is a test handle, then all exclusions of non-test-qualified objects are saved to the file, along with any exclusions of objects in test-qualified regions that appear in that test. If the first argument is a design handle, then only exclusions of non-test-qualified objects are saved.

Exclude files can be saved with mode "`w`" (write a new file, overwriting if the file already exists) or mode "`a`" (append the exclusion data onto an existing file).

You can load a previously-saved exclude file using `covdb_load_exclude_file`:

```
covdb_load_exclude_file(testHdl, "projectexcl.el");
```

As for the `covdb_save_exclude_file` function, a design handle or a test handle may be given as the first argument. If a test handle is given, exclusions will apply to all non-test-qualified objects as well as to any test-qualified objects in the specified test. If a design handle is given, exclusions will be loaded only for non-test-qualified objects.

The `covdb_unload_exclusion` function removes all `covdbExcludedAtReportTime` bits from all coverable objects in the design.

```
covdb_unload_exclusion(designHdl);
```

Default vs. Strict Exclusion

UCAPI has two exclusion modes - default and strict. In default mode, any object may be excluded. The exclusion mode can be selected using the `covdb_configure` function. To change the mode from default to strict:

```
covdb_configure(designHdl, covdbExcludeMode, "strict");
```

In strict mode, excluding a covered object is illegal. If a container is excluded, then only the uncovered objects in that container will be excluded. If a region, such as a module, is excluded, then only the uncovered objects throughout that region will be excluded.

In strict mode, UCAPI keeps track of the covered objects that were in regions or containers that were excluded - it calls these "attempted exclude" objects. The list of such objects can be saved to a file using the `covdb_save_attempted_file` function. For example:

```
covdb_save_attempted_file(mytestHdl, "attempts.txt", "w");
```

The third argument specifies that the file should be overwritten. Applications can also use ‘`a`’ to append the attempts to the specified file.

4

Accessing Coverage Data During Simulation

You can access covergroup coverage data during simulation using C code. You need to add `$coverage_dump` and `$coverage_reset` system tasks to the Verilog code to perform the following tasks:

- Monitor the coverage status periodically and modify external stimuli.
- Clear the coverage data from the simulation state so that any further data collected starts from scratch.

Monitoring the Coverage Data

You can monitor the coverage data during simulation as follows:

- Invoke a system task inside Verilog to dump the current coverage status to disk.
- Open the dumped database using UCAPI functions and examine the coverage status.

The system task `$coverage_dump` atomically dumps all coverage data to the specified test name. You can give a simple test name, `mytest`, or a composite name `mydir/mytest`, and specify where the data will be dumped. For example, a Verilog program could periodically dump the coverage status as follows:

```
always@(dump_me) begin
  $coverage_dump("mydir/myfile");
  $check_my_coverage(…);
end
```

Using this syntax, you can toggle the signal `dump_me` to cause the current coverage status to be dumped to the `mydir` directory. The snapshot of data that UCAPI calls will be given the name, `myfile`. In this example, `mydir/mytest` is overwritten whenever `dump_me` is toggled, but you can write a code to give a different name each time based on a counter.

You can also call `$cm_dump_function` directly using DPI if it is wrapped in a Verilog task.

```
export "DPI" task dump_cov_data(string s);
task dump_cov_data(string s);
$coverage_dump(s);
endtask
```

After the dump is complete, the PLI function (in this example, `check_my_coverage`) will be invoked, and you can use UCAPI to open the dumped database, check the coverage status, and take appropriate action. You can use UCAPI to load the data as follows:

```
covdbHandle design = covdb_load(covdbDesign, NULL,
"mydir"); covdbHandle test = covdb_load(design, design,
"mydir/myfile");
```

From here, the application can iterate through all covergroups, or walk over the design hierarchy. The `$coverage_dump` operation is automatic, and there is no opening or closing of a database. When `$coverage_dump` is called, the specified database is opened, written, and closed, in one automatic operation.

Note:

As soon as a database is dumped using `$coverage_dump`, it is accessible. Do not attempt to access the database until the `$coverage_dump` operation is complete.

Resetting the Coverage Data

You can reset the accumulated coverage data using the system task, `$coverage_reset`. The system task, `$coverage_reset`, clears all coverage data for all covergroups in the current simulation. For example:

```
always
begin
#10000
   $coverage_dump("mydir/myfile");
   if ($check_my_coverage(…))
   $coverage_reset();
end
```

If the PLI function, `$check_my_coverage`, returns `true`, all functional coverage data is reset, as if nothing had been covered up to this point. To dump an empty coverage database, where no bins are marked covered, you can write as follows:

```
$coverage_reset(); $coverage_dump("mydir/myfile");
```

The $coverage_reset system task does not remove any covergroups from the current simulation run. It just resets all the bins' hit counts to 0. You can only overwrite existing databases. However, if you want additional data to be merged with an existing database, you can write it with a new name as follows:

```
$coverage_dump.

$coverage_dump("mydir/mytest1"); … $coverage_reset(); …
$coverage_dump("mydir/mytest2");
```

This code writes two different databases to disk – mydir/mytest1 and mydir/mytest2. You can use UCAPI to load the first one with covdb_load, and merge the second with covdb_loadmerge, merging the data from the two separate databases. This way, you can save and merge distinct databases.

Ignoring Coverage Collected during Parts of Simulation

You can ignore any collected coverage data. Consider the following example:

```
At time N: $coverage_dump("mydir/beforeN");
At time M: $coverage_reset();
```

In this example, if there is a period between time N and M for which you wish to ignore any collected coverage data, turn coverage off during this time by writing the coverage status before time N to one database, then clearing coverage data at time M before allowing simulation to proceed.

The data collected after time M will be written out when simulation exits (or at the next call to $coverage_dump) and will include only the data collected since time M. For example, if test is called "mydir/test.db", when merged with the data stored in mydir/beforeN, the effect is as if coverage had been disabled between time N and M.

```
design = covdb_load(covdbDesign, NULL, "mydir");
test = covdb_load(covdbTest, design, "mydir/beforeN");
test = covdb_loadmerge(covdbTest, test, "mydir/test");
```

Now the handle "test" contains the coverage data from simulation from time 1 through N, and from time M through the end of simulation. However, any covergroups instantiated between time N and M will still be in the final written database (or any database written after time M). You cannot disable or delete these groups through this interface.

How the Coverage Data Is Accessed

The coverage data is accessed during simulation as follows:

1. DPI function is defined and coverage data is dumped on demand.

```
export "DPI" task dump_cov_data(string s);
export "DPI" task clear_cov_data;
task dump_cov_data(string s);
$coverage_dump(s);
endtask task clear_cov_data;
$coverage_reset();
endtask
```

2. After simulation begins, coverage is dumped by PLI code calling the DPI-exported.

```
task dump_cov_data:
… dump_cov_data("mydir/test1"); …
```

3. The PLI code, which has been linked with UCAPI, then opens the database and looks for the covergroup of interest (assume `tbMetric` has already been found).

```
covdbHandle design = covdb_load(covdbDesign, NULL,
"mydir");
covdbHandle test = covdb_load(design, design, "mydir/
test1");
covdbHandle group, groups =
covdb_qualified_iterate(test, tbMetric,
covdbDefinitions);
while((group = covdb_scan(groups))) { if (this is the
group I want …) { … check coverage status …
} } covdb_release_handle(groups);
```

4. Control then returns to the simulation, which continues collecting coverage. At a later point, the PLI code again causes coverage results to be dumped and checks the covergroup of interest once more. The new test is loaded and the covergroup handle is acquired from scratch.

```
… dump_cov_data("mydir/test2");
design = covdb_load(covdbDesign, NULL, "mydir");
test = covdb_load(design, design, "mydir/test2");
groups = covdb_qualified_iterate(test, tbMetric,
covdbDefinitions);
while((group = covdb_scan(groups))) { if (this is the
group I want …) { … check coverage status …
} } covdb_release_handle(groups);
```

5. The coverage data is reset and simulation continues.

```
clear_cov_data(); return;
```

6. This iteration is repeated as many times as required.

At any given point, the PLI code can dump coverage, analyze it, reset it, or allow it to continue without resetting. When simulation exits, coverage data is dumped as normal2 – the `$coverage_dump` and `$coverage_reset` system tasks have no effect on the final database name.

After simulation exits, a standalone UCAPI program can be used to load any of the databases dumped during simulation (or the name of the final db), using the names specified when they were dumped, as shown in the example code.

The name specified explicitly at compile time or the name given during simulation using the SystemVerilog `$set_coverage_db_name` system task. Because covergroups can be dynamically instantiated during simulation, the database written at time `N` may have fewer groups in it than the database written at a later time `N+C`. This is because VCS only collects coverage data for covergroups that are instantiated during simulation.

Thus, there is no guarantee that all databases dumped from a given simulation run will have the same groups in them. However, these databases can still be merged to create a single result, just as coverage results from multiple distinct simulation runs that may instantiate different groups can be merged. Note that there is an existing mechanism that allows individual covergroups to be disabled, `$cg_coverage_control`, but there is no existing mechanism to turn off all covergroups' coverage collection.

There is no additional way to explicitly disable functional coverage. If `$coverage_dump` is called from multiple blocks at the same time with the same argument, both calls will be processed, although the order does not matter since they both will write exactly the same data

to disk. If $coverage_dump is called from multiple blocks at the same time with different arguments, two identical databases will be written.

If $coverage_dump is called multiple times during simulation with no $coverage_reset, the databases dumped will have "overlapping" data. For example, if you perform the following:

```
$coverage_dump("mydir/test1");
…
more simulation
…
$coverage_dump("mydir/test2");
```

Then the mydir/test2 directory contains all the coverage data (hit counts, etc.) from mydir/test1 and the data collected after mydir/test1 is dumped. Thus, if you use covdb_loadmerge to merge mydir/test1 and mydir/test2, some of the coverage data in effect is duplicated.

For example, say that when mydir/test1 is dumped, bin B1 had 15 hits. When mydir/test2 is dumped, B1 will have 25 hits. If you merge mydir/test1 and mydir/test2, B1 will have 40 hits. It is the responsibility of your application to manage the relationship of these databases, since there is no way for UCAPI to know that mydir/test is really a subset of mydir/test2.

5

Limitations

This section describes which features of UCAPI are currently supported in the 2008.09 release.

- All of the information needed to generate coverage reports is present, but some value set object types (`covdbIntervalValue` and `covdbBDDValue`) are not yet supported for any metric.
- For both assertions and testbench coverage, metric-qualified data cannot be viewed without loading a test, because the metric-specific data (e.g., list of assertions or list of covergroups) is contained in test files.
- The following 1-to-1 relations are not supported in this version:
  - `covdbBDDTrue`
  - `covdbBDDFalse`

    - `covdbFromValue`
    - `covdbToValue`
- The following 1-to-many relations are not supported:
    - `covdbAnnotations`
- The following properties are not supported:
    - `covdbSigned`
    - `covdbTwoState`
    - `covdbSamplingEvent`
    - `covdbGuardCondition`
    - `covdbTypeStr`
    - `covdbParameters`
    - `covdbMessages`
    - `covdbTool`