Unified Coverage Database API Reference Manual

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

API Functions

This manual describes the functions available in UCAPI.

Coverage Data Load/Unload

The following functions are used to load and unload designs and tests.

Function	Description
covdb_load	load a design or test
covdb_loadmerge	load and merge addition design or test
covdb_unload	unload or un-loadmerge a design or test
covdb_save	save merged test data

covdb_load

The covdb_load function is used to load a design or test:

```
covdbHandle covdb_load(covdbObjTypesT type,
                       covdbHandle design,
                       char *name);
```

The *type* argument may be **covdbDesign** or **covdbTest**. If *type* is **covdbDesign**, *name* is the pathname to a coverage design directory, and *design* should be NULL. The handle returned is of type **covdbDesign**.

If *type* is **covdbTest**, *name* is the logical test name, and *design* is the design for which to load the test. The handle returned is type **covdbTest**.

For example,

```
covdbHandle design = covdb_load(covdbDesign, NULL,
“simv.cm”);
```

Note:

Only one design may be loaded in a single UCAPI session. If the application wants to load another design, covdb_unload must first be called on the first design.

covdb_loadmerge

The covdb_loadmerge function loads and merges a design (or test) with an already-loaded design (or test).

```
covdbHandle covdb_loadmerge(covdbObjTypesT type,
                            covdbHandle design_or_test,
                            char *name);
```

After a successful call of covdb_loadmerge, the covdbName property of the destination design or test will be the string “merged”. Applications can set a new name for a test using the covdb_set_str function (see “covdb_set_str” on page 19).

The primary use of covdb_loadmerge is to load and merge a list of tests from a design. See the UCAPI user guide for an example.

The other use of covdb_loadmerge is to combine multiple database directories into a single design handle. To do this, call covdb_load (covdbDesign, “.....”) to load the first directory. This must be a directory containing compile time data and not just test data. Then for each additional directory, use covdb_loadmerge(covdbDesign, designHdl, “....”) to add the new directory to the previously-loaded design handle.

---

covdb_unload

The covdb_unload function unloads all of the data corresponding to the given design or test.

```
int covdb_unload(covdbHandle design_or_test);
```

After a design is unloaded, no associated information, such as related tests, can be used. Applications should unload all tests associated with a design before unloading the design to avoid memory leaks. After unloading, all handles related to this design will be invalid and attempts to use such handles will have unpredictable effects.

If a test is unloaded, only the coverage information related to that test will be removed. After this call, handles to this test or coverage information belonging to this test will be invalid and further attempts to use such handles will have unpredictable effects.

Returns non-zero if successful, 0 if it fails.

---

covdb_save

The covdb_save function saves all coverage data in a given test handle to disk.

```
int covdb_save(covdbHandle design_or_test,
               const char* logical_fname);
```

The coverage data will be stored in the directory named ‘dirname’, and the name of the test will be ‘testname’.

---

covdb_save_exclude_file

The covdb_save_exclude_file function is used to save exclusions set on objects using the covdb_set function:

```
int covdb_save_exclude_file(covdbHandle ,
                            const char *filename,
                            const char *mode);
```

A test handle must be given as the first argument. The mode string should be "w" to overwrite any existing file, or "a" to append the exclusions to the end of a file. If the file does not exist, "w" and "a" have the same effect.

The return value is 0 on success and non-zero on failure.

covdb_load_exclude_file

The covdb_load_exclude_file function loads an exclusion file from the disk and applies its exclusions to objects in the currently-loaded design.

```
int covdb_load_exclude_file(covdbHandle design_or_test,
                            const char* filename);
```

A test handle must be given as the first argument. The return value is 0 on success and non-zero on failure.

covdb_save_attempted_file

The covdb_save_attempted_file function is used to print the list of objects that were covered, but that the application tried to exclude. In Strict mode, excluding individual covered objects is not allowed – see the User Guide for more information. This function is typically used for application debugging or to give information to the user of the application:

```
int covdb_save_attempted_file(covdbHandle testHdl,
                              const char *filename);
```

Note that a test handle must be given since objects can't be covered without knowing which test (or merged test handle) is involved.

covdb_load_mapfile

The covdb_load_mapfile function is used to specify how UCAPI should merge data from differing designs as they are loaded.

```
void covdb_load_mapfile(covdbHandle designHdl,
                                const char *mapfilename);
```

The function returns 0 on success and non-zero if an error occurs.The format of the mapfile is any number of entries of the following form:

```
MODULE: modname
INSTANCE:
SRC: instance_list
DST: instance_list
SRC: instance_list
DST: instance_list
…
```

Each instance_list is a comma-separated list of full pathnames in the base design (SRC) or an input design (DST), or the wildcard character *. See the UCAPI User Guide for more on how to use covdb_load_mapfile.

Coverage Data Model Traversal

The following functions are defined for traversing coverage data.

Function	Description
covdb_get_handle	get the handle for a 1-to-1 relation
covdb_get_qualified_handle	get a qualified handle for a 1-to-1 relation
covdb_iterate	get an iterator for a 1-to-many relation
covdb_qualified_iterate	get a qualified iterator for a 1-to-many relation
covdb_scan	get the next handle from an iterator
covdb_release_handle	release reference to a UCAPI handle
covdb_qualified_object_iterate	get qualified information about coverable objects

Functions returning handles will generally return NULL if an error occurs or if the relationship is empty. If an error occurs, any registered error callback function will also be invoked.

covdb_get_handle

The covdb_get_handle function may be called with any covdbHandle and any covdb1To1RelationT relation. If the relation `rel` is not defined for the `handle` type it will return NULL. NULL will also be returned if the relation is empty for *handle*, even if it applies to its type.

```
covdbHandle covdb_get_handle(covdbHandle handle,
                             covdb1To1RelationT rel);
```

covdb_get_qualified_handle

Similar to covdb_get_handle, but requires a qualifier handle, which must be of type covdbMetric or covdbTest. If the relation `rel` is not defined for the `handle` type it will return NULL. NULL will also be returned if the relation is empty for `handle`, even if it applies to the *handle* type.

```
covdbHandle covdb_get_qualified_handle(covdbHandle handle,
                                       covdbHandle qual,
                                     covdb1To1RelationT rel);
```

covdb_iterate

Returns a handle to an iterator over the objects for the specified 1-to-many relation for `handle`. If the relation `rel` is empty, or does not apply to `handle`, a NULL iterator handle will be returned.

Handles returned by covdb_iterate may only be scanned through one time - they cannot be reset to the beginning. To start over and iterate from the beginning again, acquire a new handle using covdb_iterate:

```
covdbHandle covdb_iterate(covdbHandle handle,
                          covdb1ToManyRelationT rel);
```

covdb_qualified_iterate

Returns a handle to an iterator over the objects for the qualified 1-to-many relation for `handle`. If the relation `rel` is empty, or does not apply to `handle`, a NULL iterator handle will be returned.

Handles returned by covdb_qualified_iterate may only be scanned through one time - they cannot be reset to the beginning. To start over and iterate from the beginning again, acquire a new handle using covdb_qualified_iterate.

```
covdbHandle covdb_qualified_iterate(covdbHandle handle,
                                    covdbHandle qual,
                                  covdb1ToManyRelationT rel);
```

covdb_scan

The covdb_scan function returns the next object from an iterator handle and advances the iterator.

Only one handle returned by covdb_scan for a given iterator is valid at any time – once covdb_scan is called again, the handles returned by previous calls are invalid.

If called on an object that is not an iterator, it returns NULL.

```
covdbHandle covdb_scan(covdbHandle iter_handle);
```

The handles returned by covdb_scan are volatile and may be overwritten by the next call to a UCAPI function. You can make a handle persistent by calling covdb_make_persistent_handle.

covdb_qualified_object_iterate

The covdb_qualified_object_iterate function is used when iterating the qualified contents of an object inside a region.

```
covdbHandle covdb_qualified_object_iterate(
                          covdbHandle objHdl,
                           covdbHandle regionHdl,
                          covdbHandle qualHdl,
                          covdb1ToManyRelationsT rel);
```

For example, the covdbTests relation from an object handle is such a 1-to-many relation. The object is the coverable object whose test coverage you want to iterate. The region handle is the region that contains the object (e.g., a covdbSourceInstance handle). The qualified is the merged test handle containing tests t1, t2, …, tN.

The iterator for a given object will contain the subset of { t1, t2, …, tN } where any ti in the set is a test that covers the object. See the Examples section in the user guide for a detailed example.

Memory and Pointer Management

A covdbHandle is a pointer that points to an object. If the object is made persistent, it gets copied into a safe region in the memory. When the handle is later passed to covdb_release_handle, the memory is cleaned up. The covdbHandle will still be pointing to the same (corrupted) memory location, but the object itself will be gone.

Example 1:

```
covdbHandle H = covdb_get(obj, covdbObjects);
H = covdb_make_persistent_handle(H);
covdbHandle C = H;
covdb_release_handle(H)
```

Here, C is a persistent handle.

However, releasing handle H invalidates the safe memory created for the object, thus C and H both point to memory that is now corrupt.

Example 2:

```
covdbHandle K = covdb_get(obj, covdbObjects);
covdbHandle J = covdb_make_persistent_handle(K);
covdb_release_handle(J);
```

Here, K is not persistent and covdb_release_handle(J) does not affect the status of K. However, since K was not made persistent, it may become invalid at the next UCAPI function call.

UCAPI handles returned by covdb_scan, covdb_get_handle, covdb_get_qualified_handle are not persistent. That is, they are only guaranteed to be valid until the next call to a UCAPI function.

Handles returned by covdb_iterate and covdb_qualified_iterate are persistent and must be explicitly released by the application after it exits the loop, or the memory associated with them will leak.

The following functions are provided for managing handles:

Function	**Description**
covdb_make_persistent_handle	make a UCAPI handle persistent
covdb_release_handle	release a persistent handle or iterator

covdb_make_persistent_handle

The covdb_make_persistent_handle function returns a persistent handle for an object.

```
covdbHandle covdb_make_persistent_handle(covdbHandle
                                                    handle);
```

The handle it returns is guaranteed to remain valid until the application releases it with covdb_release_handle.

covdb_release_handle

The covdb_release_handle function releases the given handle.

```
void covdb_release_handle(covdbHandle handle);
```

When an application is done using a persistent handle, it should call covdb_release_handle. If applications do not call covdb_release_handle before discarding a persistent handle, memory may be lost (leaked).

Reading Properties

These functions are defined to read properties from UCAPI object handles.

Function	Description
covdb_version()	returns the version string
covdb_get	read an integer-valued property
covdb_get_str	read a string-valued property

covdb_get_real	read a real-valued property

covdb_version

The char *covdb_version() returns the version string. For example:

```
char *version = covdb_version();
  if (!is_supported_version(version))  {
   printf("Error: this version of UCAPI (%s) is not in the
supported list for my application.\n", version);
  }
```

covdb_get

The covdb_get function returns the value of the specified integer property *prop* for the given object *handle*. If the property is not an integer-valued property, or the value is not defined for the given object, it returns -1. Note that the value returned may be an integer, covdbObjTypeT, or covdbScalarValueT, depending on which property is read.

An object handle must always be specified. If the object is itself a region, no region handle must be specified. If the object is not a region, then a handle to its enclosing region must be given as well.

A test handle must be given if the property is test-qualified (such as covdbCovered or covdbCovCount).

The integer properties are defined in.

```
int covdb_get(covdbHandle object,
              covdbHandle region,
              covdbHandle test,
              covdbPropertiesT prop);
```

For example, to get the UCAPI type of an instance handle:

```
type = covdb_get(instHdl, NULL, NULL, covdbType);
```

To get the number of coverable objects for a given metric for a metric-qualified instance handle:

```
instTot = covdb_get(metricInstHdl, NULL, NULL,
                    covdbCoverable);
```

To get the number of covered objects for a given metric for a metric-qualified instance handle, you have to add a test handle:

```
instCov = covdb_get (metricInstHdl, NULL, testHdl,
                    covdbCovered);
```

To get the number of coverable objects inside a metric-qualified object (such as a covdbContainer object), we specify the instance, the object, and the covdbCoverable property, but no test handle is required:

```
objTot = covdb_get(objHdl, metricInstHdl, NULL,
                        covdbCoverable);
```

To get the number of covered objects inside such an object, we have to use a test handle as well. This is the same form we'd use to query whether a given coverable object is covered or not with respect to a given test:

```
objCov = covdb_get(objHdl, metricInstHdl, testHdl,
                        covdbCovered);
```

covdb_get_str

The covdb_get_str function returns the value of the specified string property `prop` for the given object `handle`. If the property is not a string-valued property, or the value is not defined for the given object, it returns NULL.

```
char *covdb_get_str(covdbHandle handle,
                    covdbPropertiesT prop);
```

The string properties are defined in section “Object Properties” on page 26.

String values returned by covdb_get_str are volatile and are not guaranteed to persist beyond the next call to covdb_get_str. Applications must make a copy if they want a persistent string.

covdb_get_real

The covdb_get_real function returns the value of the specified real property `prop` for the given object `handle`. If the property is not a real-valued property, or the value is not defined for the given object, it returns -1.0.

```
double covdb_get_real(covdbHandle handle,
                      covdbPropertiesT prop);
```

Reading Annotations

Annotations may be unqualified or qualified. Annotations may be read either by the name (key) of the annotation, or through iteration of covdbAnnotation objects. To iterate over all annotations for an object, the iteration functions described in this section are used. The following functions are used to read annotations by name.

Function	Description
covdb_get_annotation	read an annotation by name
covdb_get_qualified_annotation	read a qualified annotation by name
covdb_get_integer_annotation	read an integer type annotation

covdb_get_annotation

The covdb_get_annotation function returns the value of the specified annotation `key` for the given object `handle`. If the annotation is not defined for `handle`, it returns NULL.

```
char *covdb_get_annotation(covdbHandle handle, char *key);
```

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");
```

covdb_get_qualified_annotation

The covdb_get_qualified_annotation function returns the value of the specified annotation `key` for the given object `handle` qualified by covdbTest or covdbMetric handle `qual`. If the annotation is not defined for this handle and qualifier, it returns NULL.

```
char *covdb_get_qualified_annotation(covdbHandle handle,
                                     covdbHandle qual,
                                     char *key);
```

covdb_get_integer_annotation

The covdb_get_integer_annotation function returns the value of the specified integer-type annotation key for the given object handle. If the annotation is not defined for this handle, an error will be flagged and -1 will be returned. The only way to distinguish between the value not being set and it being a valid value of -1 is to check the error status, either by calling covdb_get_error or by using a error callback function.

```
int covdb_get_integer_annotation(covdbHandle obj, const
                                                 char* key);
```

Setting Properties

Function	Description
covdb_set	set an integer property of an object
covdb_set_str	set a string property of an object

covdb_set

UCAPI allows applications to change some properties for line, toggle, FSM, assertion, branch and condition coverage on coverable objects, as described in this section. UCAPI currently does not allow applications to change property values for any covergroup objects, or for any other handle types.

The covdb_set function is used to set the value of an integer-type property on a UCAPI handle.

```
void covdbStatusT covdb_set(covdbHandle handle,
                            covdbHandle region,
                            covdbHandle test,
                            covdbPropertiesT property,
                            int value);
```

This function is used to set integer properties of UCAPI handles. The properties which can be set are:

- `covdbCovered`
- `covdbCovCount`

Applications may also set one of the following flags in the covdbCovStatus property:

- `covdbStatusExcludedReportTime`
- `covdbStatusUnreachable`
- `covdbStatusCovered`

These properties can only be set on coverable objects (value sets, blocks, sequences and crosses), and not on coverable objects used only for annotation purposes (such as on the value sets inside sequences). covdbCovered and covdbCovCount are set directly:

```
covdb_set(objHdl, regHdl, testHdl, covdbCovered, 1);
covdb_set(objHdl, regHdl, testHdl, covdbCovCount, 19);
```

To set a value in the covdbCovStatus property, applications should read the existing value and modify it:

```
curval = covdb_get(objHdl, regHdl, testHdl, covdbCovStatus);
covdb_set(objHdl, regHdl, testHdl, covdbCovStatus,
```

```
                                    (curval | covdbStatusCovered));
```

Clearing a flag in the covdbCovStatus property is similar:

```
curval = covdb_get(objHdl, regHdl, testHdl, covdbCovStatus);
      covdb_set(objHdl, regHdl, testHdl, covdbCovStatus,
                           (curval & (~covdbStatusCovered)));
```

Setting covdbStatusCovered on a supported handle type will automatically set the covdbCovered value of that handle equal to its covdbCoverable value.

Setting the covdbCovered property equal to the covdbCoverable property for a handle will automatically set the covdbStatusCovered flag on the handle.

Setting covdbCovered to any value less than the covdbCoverable for a handle will clear the covdbStatusCovered flag on that object.

The test handle is always a required argument to covdb_set, because properties that can be set are always test-specific. When a property is changed on an object for a given test, if that test is saved, the new value of the property will be saved as well.

The covdb_set function is currently not supported.

---

covdb_set_str

UCAPI allows applications to change the name of a covdbTestHandle using the covdb_set_str function. Changing the value of other string properties is not supported for any other handle types of string properties.

The covdb_set_str function is used to set the value of a string-type *property* on a UCAPI *handle*.

```
char *covdb_set_str(covdbHandle handle,
                    covdbPropertiesT property,
                    char *value);
```

Error Handling and Recovery

Function	Description
covdb_set_error_callback	register a function to be called back
covdb_get_error	check error status
covdb_configure	set error reporting status

covdb_set_error_callback

The covdb_set_error_callback function may be used by an application to register a function to be called if an error is detected by UCAPI.

```
int covdb_set_error_callback(void (*cbfn)(covdbHandle
                                          errHdl, void* data),
                             void *data);
```

When an error is detected, UCAPI calls the `cbfn` function with a UCAPI error handle and the data pointer that was given when the callback function was registered. The error handle can be queried for the error code using the covdbValue property and a string error message using the covdbName property, for example:

```
covdb_set_error_callback(mycallback, ”phase0”);
…
perform some UCAPI operations
```

```
…
covdb_set_error_callback(mycallback, ”phase1”);
…

void mycallback(covdbHandle errHdl, void *data)
{
     char *phase = (char*)data;
     int errcode = covdb_get(errHdl, NULL, NULL, covdbValue);
     char *errmsg = covdb_get_str(errHdl, covdbName);

     printf(“error #%d occurred in phase %s: %s\n”, errcode,
             errmsg, phase);
}
```

The handle passed to the application’s callback function is of type covdbError. It should be released using covdb_release_handle when the application is finished with it.

The covdb_set_error_callback function returns -1 if the callback function could not be registered, and 0 if the registration was successful.

---

covdb_get_error

The covdb_get_error function returns a covdbStatusT code (as defined in covdb_user.h). A value of covdbNoError indicates no error has occurred. Other values give the type of error (such as covdbOutOfMemoryError). If an error occurred, the value of *msg will be set to a descriptive string.

```
covdbStatusT *covdb_get_error(char **msg);
```

covdb_configure

The covdb_configure function allows applications to configure the error reporting status of UCAPI. By default, UCAPI will not print error messages to the display – applications must use the covdb_get_error function to check when an error has occurred.

If covdb_configure is called to set covdbDisplayErrors to true, then error messages will be printed. For example:

```
covdb_configure(covdbDisplayErrors, “true”);
```

The legal configuration items for covdb_configure include:

- covdbDisplayErrors. Default value false. If true, errors will be printed to the standard output when they occur.

The default value of each of these configuration options is true. If set to "false", data for the given metric will not be loaded even if it is present in the coverage directories:

- covdbLoadLine.

- covdbLoadCond

- covdbLoadTgl

- covdbLoadFsm

- covdbLoadBranch

- covdbLoadAssert

- covdbLoadGroup

The configuration option covdbLimitedDesign has the default value "false". If set to "true", then the design hierarchy information will not be loaded, and only assertion and covergroup data will be loaded. In "limited design" mode, to access assertion or covergroup data applications must iterate from the test handle. For example, to get the list of assertions:

```
assts = covdb_qualified_iterate(testHdl, assertMetricHdl,
                                        covdbObjects);
```

Applications can also control simple mapping (merging of data from different designs) using covdb_configure. The configuration item covdbMappedModule is set to the name of the module to be mapped. This has the same effect as the -map option to URG. Finer control of mapping is available using the mapfile - see covdb_load_mapfile.

---

covdb_qualified_configure

The covdb_qualified_configure is used to set some other configuration options. For implementation reasons, some configuration options require that the design handle be passed.

The legal configuration items for covdb_qualified_configure include:

- covdbKeepTestInfo. Default value "false". If set to true, the list of tests that covered each object will be preserved on that object when multiple tests are loadmerged. This data can be read using covdb_qualified_object_iterate.

- covdbMaxTestsPerCoverable. Default value "0". If set to non-zero, the maximum number of tests that will be preserved for each object when multiple tests are loadmerged.

Types, Properties, and Relations

This section contains the complete list of all properties and relations in the UCAPI model.

Object Types

UCAPI has a small number of different object types. The type of an object may be retrieved from any handle, for example:

```
covdbObjTypesT objty = covdb_get(objHandle, covdbType);
```

The type of a UCAPI handle will always be one of:

```
typedef enum {
              covdbSourceDefinition,
              covdbSourceInstance,
              covdbDesign,
              covdbTest,
              covdbTestName,
              covdbMetric,
              covdbContainer,
              covdbSequence,
              covdbCross,
              covdbBlock,
              covdbAnnotation,
              covdbIterator,
              covdbIntervalValue,
              covdbBDDValue,
              covdbIntegerValue,
              covdbScalarValue,
              covdbVectorValue,
              covdbInterval,
              covdbVector,
              covdbBDD
             }
```

```
            covdbObjTypesT;
```

1-To-1 Relations

You can use these with the covdb_get_handle or covdb_get_qualified_handle functions.

```
typedef enum {
              covdbIdentity,
              covdbParent,
              covdbDefinition,
              covdbBDDTrue,
              covdbBDDFalse,
              covdbVecValue,
              covdbFromValue,
              covdbToValue
             }
            covdb1To1RelationsT;
```

1-To-Many Relations

You can use these with the covdb_iterate or covdb_qualified_iterate functions.

```
typedef enum {
              covdbObjects,
              covdbMetrics,
              covdbInstances,
              covdbDefinitions,
              covdbLoadedTests,
              covdbAvailableTests,
              covdbTests,
              covdbAnnotations,
              covdbComponents
             }
            covdb1ToManyRelationsT;
```

Object Properties

These properties can be used with the covdb_get, covdb_get_str, and covdb_get_real functions, as noted:

```
typedef enum {
  /* integer properties for covdb_get */
              covdbLineNo,
              covdbWeight,
              covdbCoverable,
              covdbDeepCoverable,
              covdbValue,
              covdbType,
              covdbCovCount,
              covdbCovCountGoal,
              covdbCovered,
              covdbDeepCovered,
              covdbNumObjects,
              covdbIsVerilog,
              covdbIsVhdl,
              covdbAutomatic,
              covdbCovStatus,
              covdbWidth,
              covdbSigned,
              covdbTwoState,

  /* string properties for covdb_get_str */
              covdbName,
              covdbValueName,
              covdbFullName,
              covdbFileName,
              covdbSamplingEvent,
              covdbGuardCondition,
              covdbTypeStr,
              covdbParameters,
              covdbMessages,
              covdbTool,

  /* properties for covdb_get_real */
              covdbCovGoal
```

```
}
covdbPropertiesT;
```

Limitations

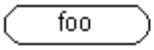
covdbNumObjects, covdbSamplingEvent, covdbParameters, and covdbGuardCondition are not yet supported.

Values

Objects of type covdbScalarValue return one of these enumerated values for the property covdbValue (objects of type covdbIntValue return an integer).

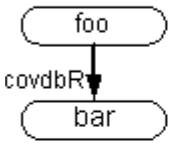
```
typedef enum {
        covdbValue0,
        covdbValue1,
        covdbValueX
    }
            covdbScalarValueT;
```

The figures below show how diagrams are used to indicate properties and relations in the user guide:



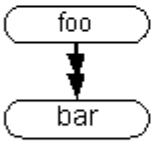
Object of type foo

An abstract class foobar of which bar is a subtype



One-to-one relation *covdbR* from an object *foo* to an object of type *bar*

```
bar = covdb_get_handle(foo, covdbR)
```



One-to-many relation *covdbR* from an object *foo* to an object of type *bar*

```
iter = covdb_iterate(foo, covdbR);
while((bar = covdb_scan(iter))) { ... }
```

Qualified one-to-many relation *covdbR* from an object *foo* to an object of type *bar,* qualified by *handle*

```
iter = covdb_qualified_iterate(foo, handle, covdbR);
while((bar = covdb_scan(iter))) { ... }
```

Object of type *bar* has integer property *prop*