

System Verilog Coverage API

João Geadá

Change Log

Version	Date	Authors	Description
0.1	01/20/2003	João	First draft proposal for incorporating coverage API donation into SV framework
0.2	01/27/2003	João	Updates as per feedback from the face-to-face sv-cc meeting held on 01/23/2003

Table of Contents

System Verilog Coverage API.....	1
1 Requirements overview	2
1.1 SystemVerilog API	2
1.2 VPI extensions API.....	2
1.3 Naming conventions	2
2 Coverage Definitions	2
3 SystemVerilog real-time coverage access	3
3.1 Predefined coverage constants in SystemVerilog.....	3
3.2 Built-in coverage access system functions	3
3.2.1 \$coverage_control.....	3
3.2.2 \$coverage_get_max	6
3.2.3 \$coverage_get	6
3.2.4 \$coverage_merge	6
3.2.5 \$coverage_save	7
4 FSM recognition	7
4.1 FSM pragmas	8
5 VPI coverage extensions.....	8
5.1 Extensions to VPI enumerations	8
5.2 Obtaining coverage information	9
5.3 Controlling coverage.....	9

1 Requirements overview

1.1 SystemVerilog API

1. API should be similar for all coverages
There are a wide number of coverage types available, with possibly different sets offered by different vendors. Maintaining a common interface across all the different types enhances portability and ease of use.
2. Support at minimum the following types of coverage
 - a. statement coverage
 - b. toggle coverage
 - c. fsm coverage
 - i. fsm states
 - ii. fsm transitions
 - d. assertion coverage
3. Set of coverages supported by API should be extensible in a transparent manner ie adding a new coverage type should not break any existing coverage usage.
4. API must provide means to obtain coverage information from only specific sub-hierarchies of the design without requiring full enumeration of all instances in those hierarchies by the user

1.2 VPI extensions API

1.3 Naming conventions

All elements added by this interface will conform to the vpi interface naming conventions:

1. all names will be prefixed by vpi
2. type names will start with “vpi”, followed by Capitalized words with no separators, eg vpiCoverageStmnt
3. all function names will start with “vpi_” followed by all lowercase words separated by ‘_’, eg vpi_control()

2 Coverage Definitions

Statement coverage: whether a statement has been executed or not, where statement is anything defined as a statement in the LRM. Covered means: executed at least once. Some implementations may also permit the execution count to be queried. Note that the granularity of statement coverage may be either per statement or per statement block (however defined) and neither approach will be favored by this standard.

Fsm coverage: the number of states in an FSM that have been reached by this simulation. Fsm automatic extraction is not required by this standard, but a standard mechanism to force specific extraction is available via pragmas.

Toggle coverage: for each bit of every signal (wire and register), whether that bit has had both a 0 value and a 1 value. Full coverage means both seen, otherwise partial coverage may be queried by some implementations. Some implementations may also permit the toggle count of each bit to be queried.

Assertion coverage: for each assertion, whether it has had at least one success. Implementations may permit further details to be queried, such as attempt counts, success counts, failure counts and failure coverage.

Note that the above defines the “primitives” for each coverage type. Over instances or blocks, the coverage number is merely the sum of all contained primitives in that instance or block (if/as appropriate).

3 SystemVerilog real-time coverage access

3.1 Predefined coverage constants in SystemVerilog

The following predefine ‘defines will exist in SystemVerilog to represent basic real-time coverage capabilities accessible directly from SystemVerilog

Coverage control

```
`define SV_COV_START          0
`define SV_COV_STOP          1
`define SV_COV_RESET         2
`define SV_COV_QUERY         3
```

Scope definition (hierarchy traversal/accumulation type)

```
`define SV_COV_MODULE        10
`define SV_COV_HIER          11
```

Coverage type identification

```
`define SV_COV_ASSERTION     20
`define SV_COV_FSM_STATE     21
`define SV_COV_STATEMENT     22
`define SV_COV_TOGGLE        23
```

Status results

```
`define SV_COV_ERROR         -1
`define SV_COV_NOCOV         0
`define SV_COV_OK            1
`define SV_COV_PARTIAL       2
```

3.2 Built-in coverage access system functions

3.2.1 \$coverage_control

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instance)
```

Enables, disables, resets or **queries** the availability of coverage information for the specified portion of the hierarchy. The return value is an integer, with the value indicating the success of the action and will be one of

- **SV_COV_OK**
Request successful. If starting or stopping or resetting this means the desired effect occurred, if **querying**, means coverage available. A successful reset clears all coverage (ie a ...get() == 0 after a successful ...reset())
- **SV_COV_ERROR**
Call failed with no action, typically due to errors in the arguments, such as non-existing module or instance specifications
- **SV_COV_NOCOV**
Coverage not available for the requested portion of the hierarchy
- **SV_COV_PARTIAL**
Coverage only partially available in the requested portion of the hierarchy (ie some instances have the requested coverage information, some don't)

Starting, stopping or resetting coverage multiple times in succession for the same instance(s) has no further effect if coverage is already started/stopped/reset for those instance(s).

The hierarchy(ies) being controlled/queried are specified as follows:

SV_MODULE_COV, "unique module def name" :

coverage of all instances of the given module (**unique** module name given as a string), but excluding any child instances in the instances of the given module. **Note that module definition name may have special notation to describe nested module definitions.**

SV_COV_HIER, "module name" :

coverage of all instances of the given module including all the hierarchy below

SV_MODULE_COV, instance_name :

coverage of the one named instance. Instance is specified as a normal Verilog hierarchical path



SV_COV_HIER, instance_name :

coverage of the named instance plus all the hierarchy below.

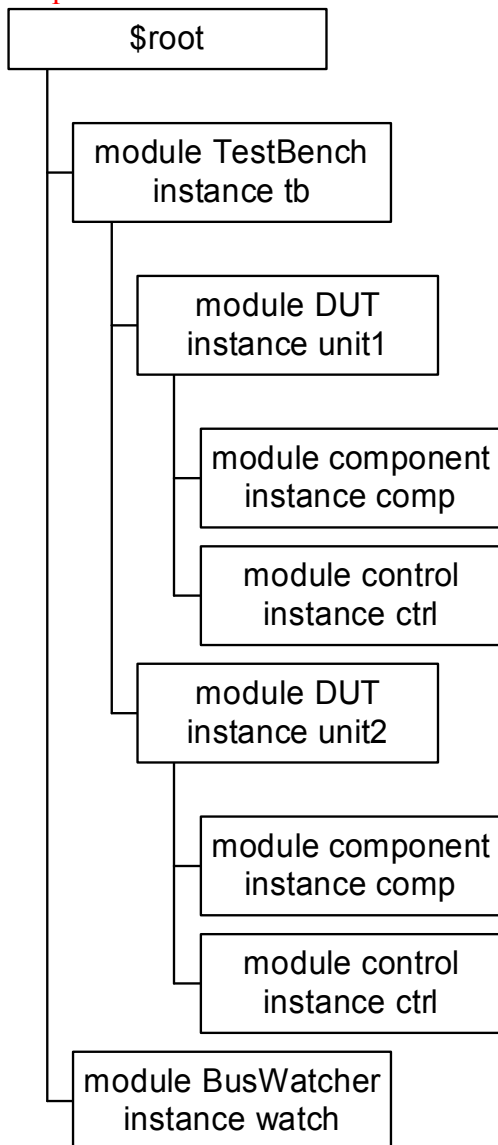
All the permutations are presented in a table below:

	"definition name"	instance.name
SV_COV_MODULE	sum of coverage for all instances of the named module, but excluding any hierarchy below those instances.	coverage for just the named instance, excluding any hierarchy in instances below that instance
SV_COV_HIER	sum of coverage for all instances of the named module, including all coverage for all hierarchy	coverage for the named instance and any hierarchy below that one instance.

| below those instances |

Note: definition names are represented as strings, whereas instance names are referred by hierarchical paths. Note also that a hierarchical path need not include any '.' if the path refers to an instance in the current context (ie normal Verilog hierarchical path rules apply)

Example:



Assuming that coverage is enabled on all the instances of the example design above, then:

- `$coverage_control('SV_COV_CHECK, 'SV_COV_HIER, $root)`
will check all instances to verify that they have coverage, and in this case, will return `'SV_COV_OK`
- `$coverage_control('SV_COV_RESET, 'SV_COV_MODULE, "DUT")`
will reset coverage collection on both instances of DUT, specifically,

- `$root.tb.unit1` and `$root.tb.unit2`, but will leave coverage unaffected in all other instances
- `$coverage_control('SV_COV_RESET, 'SV_COV_MODULE, $root.tb.unit1)` will reset coverage of only the instance `$root.tb.unit1`, leaving all other instances unaffected
 - `$coverage_control('SV_COV_STOP, 'SV_COV_HIER, $root.tb.unit1)` will reset coverage of the instance `$root.tb.unit1` and also reset coverage for all instances below, specifically `$root.tb.unit1.comp` and `$root.tb.unit1.ctrl`.
 - `$coverage_control('SV_COV_START, 'SV_COV_HIER, "DUT")` will start coverage on all instances of the module DUT and of all hierarchy(ies) below those instances. In this design, this means that coverage will be started for the instances `$root.tb.unit1`, `$root.tb.unit1.comp`, `$root.tb.unit1.ctrl`, `$root.tb.unit2`, `$root.tb.unit2.comp` and `$root.tb.unit2.ctrl`

3.2.2 \$coverage_get_max

`$coverage_get_max(coverage_type, scope_def, modules_or_instance)`

Obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy. This value will remain constant across the duration of the simulation. **Note:** This value is proportional to the design size and structure, so it should also be constant through multiple independent simulations and compilations of the same design, assuming that compilation options do not modify either coverage support or design structure. The return value is an integer, with the following meanings:

- -1 (SV_COV_ERROR): an error occurred (incorrect arguments)
- 0 (SV_COV_NOCOV): no coverage available for that coverage type on that hierarchy(ies)
- +ve: maximum coverage number, which is the sum of all coverable items of that type over the given hierarchy(ies).

Scope specified as per `$coverage_control`

3.2.3 \$coverage_get

`$coverage_get(coverage_type, scope_def, modules_or_instance)`

Obtains the current coverage value for the given coverage type over the given portion of the hierarchy. This number can be converted to a coverage percentage by use of the

equation
$$\text{coverage\%} = \frac{\text{coverage_get}()}{\text{coverage_get_max}()} * 100$$
. The return value follows the same

pattern as `...get_max()`, but with the +ve number representing the current coverage level ie the number of the coverable items that have been covered in this hierarchy(ies)

Scope specified as per `$coverage_control`

3.2.4 \$coverage_merge

`$coverage_merge(coverage_type, "name")`

Loads and merges coverage data for the specified coverage into the simulator. **"name" is an arbitrary string used by the tool, in an implementation specific way, to locate the**

appropriate coverage database¹. If “name” does not exist or does not correspond to a coverage database from the same design an error will occur. If an error occurs during loading, the coverage numbers generated by this simulation might not be meaningful.

The return values from this function are:


COV_OK	coverage data found and merged
COV_NOCOV	coverage data found, but did not contain the coverage type requested
COV_ERROR	coverage data not found, or did not correspond to this design or other error

3.2.5 \$coverage_save

\$coverage_save(coverage_type, “name”)

Saves the current state of coverage to the tool’s coverage database and associated with the name “filename”. This filename must not contain any directory specification or extensions. Data saved to the database must be able to be retrieved later by the ...merge function supplied the same name. Saving coverage must not have any effect on the state of coverage in this simulation.

The return values from this function are:

COV_OK	coverage data successfully saved
COV_NOCOV 	no such coverage available in this design (nothing saved)
COV_ERROR	some error occurred during the save. If an error occurs, the coverage database entry for “name” must be removed automatically by the tool to preserve the coverage database integrity. Note that it is <i>not</i> an error to overwrite a previously existing “name”.

Notes:

1. Coverage database format is implementation dependent.
2. Mapping of names to actual directories/files is implementation dependent. There is no requirement that a coverage name map to any specific set of files or directories.

4 FSM recognition

It is assumed that coverage tools will have automatic recognition of many of the common FSM coding idioms in Verilog/SystemVerilog. The standard will not attempt to describe or require any specific automatic FSM recognition mechanisms.

However, the standard will prescribe a means by which non-automatic FSM extraction will occur. The presence of any of these standard FSM description additions *must* override the tool’s default extraction mechanism.

Identification of an FSM consists of identifying the following items

1. the state register (or expression)
2. the next state register (this is optional)
3. the legal states

¹ ie tools are allowed to store coverage files any place they want with any extension they want *as long* as the user can retrieve the information by asking for a specific saved name from that coverage database

4. the legal transitions between states

4.1 FSM pragmas²

FSM pragmas directly identify the state register, next state register (if any) and the valid states. From this information it is straightforward to identify the valid transitions from analysis of the Verilog source code.

State variable or expression identification

```
/* SV FSM state_vector <stateVarName> [<FSMName>] [enum <enumName>] */
/* SV FSM state_vector <stateVarPartSel> <FSMName> [enum <enumName>] */
/* SV FSM state_vector <stateVarConcat> <FSMName> [enum <enumName>] */
```

Associate a reg, wire or parameter definition with a specific FSM

```
/* SV FSM enum <enumName> */
```

5 VPI coverage extensions

5.1 Extensions to VPI enumerations

Coverage control

- #define vpiCoverageStart
- #define vpiCoverageStop
- #define vpiCoverageReset
- #define vpiCoverageCheck
- #define vpiCoverageMerge
- #define vpiCoverageSave

VPI properties

- Coverage type properties
 - #define vpiAssertCoverage
 - #define vpiFsmStateCoverage
 - #define vpiStatementCoverage
 - #define vpiToggleCoverage
- Coverage status properties
 - #define vpiCovered
 - #define vpiCoverMax
 - #define vpiCoveredCount
- Assertion specific coverage status properties
 - #define vpiAssertAttemptCovered
 - #define vpiAssertSuccessCovered
 - #define vpiAssertFailureCovered
- FSM specific methods
 - #define vpiFsmStates
 - #define vpiFsmStateExpression

FSM handle types (vpi types)

- #define vpiFsm

² full details given in an attached extract from VCS Coverage Metrics (VCM) documentation

- `#define vpiFsmHandle`

5.2 Obtaining coverage information

All use `vpi_get()` with the appropriate properties and object handles
coverage type, instance -> number of covered items in the given instance

`vpiCovered`, handle -> number of items of handle type covered. Only applicable to:
statement handles, signal (wire/reg) handles, assertion handles, fsm handles

`vpiCoveredCount`, handle -> number of times each item of handle type covered. Only easily interpretable when handle points to a unique coverable item (otherwise sum of counts of all contained items)

`vpiCoveredMax`, handle -> total possible coverable items in the given handle. Handle types limited as per above. Note that this is only really useful when handle is a handle to an object possibly containing more than 1 coverable item.

Use `vpi_iterate(vpiFsm, instance-handle)` to get iterator to all fsms in an instance

Use `vpi_handle(vpiFsmStateExpression, fsm-handle)` to get handle to the signal/expression encoding the Fsm state.

Use `vpi_iterate(vpiFsmStates, fsm-handle)` to get iterator to all states of an fsm

Use `vpi_get_value(fsm_state_handle, state-handle)` to get value of a state

5.3 Controlling coverage

`vpi_control()`

3 arguments: coverage control (start, stop, reset, **query**), coverage type, and handle to appropriate instance or assertion. Note that statement, toggle and fsm coverage are not individually controllable (ie controllable only at the instance level, **not on a per statement/signal/fsm**). **Semantics and behavior** as per the equivalent system function (`$coverage_control`, [section 3.2.1](#))

`vpi_control()`

3 arguments: coverage control (merge, save), coverage type, "name". Merges coverage into the current simulation. **Semantics and behavior** as per the equivalent system functions (`$coverage_merge`, [section 3.2.4](#) and `$coverage_save`, [section 3.2.5](#))

