

Section 2

SystemVerilog Assertion API

This chapter defines the Assertion Application Programming Interface (API) in SystemVerilog 3.1/draft 2.

2.1 Requirements

SystemVerilog 3.1/draft 2 provides assertion capabilities to enable:

- a user's C code to react to assertion events,
- third-party assertion "waveform" dumping tools to be written,
- third-party assertion coverage tools to be written, and
- third-party assertion debug tools to be written.

2.1.1 Naming conventions

All elements added by this interface shall conform to the Verilog Procedural Interface (VPI) interface naming conventions.

- All names are prefixed by `vpi`.
- All *type names* shall start with `vpi`, followed by initially capitalized words with no separators, e.g., `vpiAssertCheck`.
- All callback names shall start with `cb`, followed by initially capitalized words with no separators, e.g., `cbAssertionStart`.
- All *function names* shall start with `vpi_`, followed by all lowercase words separated by underscores (`_`), e.g., `vpi_get_assert_info()`.

2.1.2 Nomenclature

The following terms are used in this standard.

Directive — a type applied to a temporal expression describing how the results of the temporal expression are to be captured and/or interpreted.

Assertion clock — the Verilog event expression that indicates to an assertion when time has advanced (and when HDL signals can be sampled, etc.).

Temporal expression — ****Add this from the SV-AC LRM****

Assertions — A declarative expression (one or more clock cycles) describing the behavior of a system over time.

2.2 Extensions to VPI enumerations

These extension shall be merged into the contents of the `vpi_user.h` file, described in *IEEE Std 1364-2001*, Annex G. The numbers in the range 700 - 799 are reserved for the assertion portion of the VPI.

2.2.1 Object types

This section lists the object type VPI calls. The VPI reserved range for these call is 700 - 729.

```
#define vpiAssertion          700  /* assertion */
```

2.2.2 Object properties

This section lists the object property VPI calls. The VPI reserved range for these call is 700 - 729.

```

#define vpiAssertAssertion      701
#define vpiAssumeAssertion      702
#define vpiRestrictAssertion    703
#define vpiCoverAssertion       704
#define vpiCheckAssertion        705 /* inlined behavior assertion */
#define vpiAssertionDirective    706 /* method to obtain assertion directive */

```

2.2.3 Callbacks

This section lists the system callbacks. The VPI reserved range for these call is 700 - 719.

1) Assertion

```

#define cbAssertionStart        700
#define cbAssertionSuccess      701
#define cbAssertionFailure      702
#define cbAssertionStepSuccess  703
#define cbAssertionStepFailure  704
#define cbAssertionDisable      705
#define cbAssertionEnable       706
#define cbAssertionReset        707
#define cbAssertionKill         708

```

2) “Assertion system”

```

#define cbAssertionSysInitialized 709
#define cbAssertionSysStart      710
#define cbAssertionSysStop       711
#define cbAssertionSysEnd        712
#define cbAssertionSysReset      713

```

2.2.4 Control constants

This section lists the system control constant callbacks. The VPI reserved range for these call is 730 - 759.

1) Assertion

```

#define vpiAssertionDisable      730
#define vpiAssertionEnable       731
#define vpiAssertionReset        732
#define vpiAssertionKill         733
#define vpiAssertionEnableStep   734
#define vpiAssertionDisableStep  735

```

2) Assertion stepping

```

#define vpiAssertionClockSteps    736

```

3) “Assertion system”

```

#define vpiAssertionSysStart      737
#define vpiAssertionSysStop       738
#define vpiAssertionSysEnd        739
#define vpiAssertionSysReset      740

```

2.3 Static information

This section defines how to obtain assertion handles and other static assertion information.

2.3.1 Obtaining assertion handles

SystemVerilog 3.1/draft 2 extends the VPI module iterator model (i.e., the instance) to encompass assertions, as shown in Figure 2-1—. ****Revise this xref w/ Stu; also check/revise variable settings, etc.****

The following steps highlight how to obtain the assertion handles for named assertions.

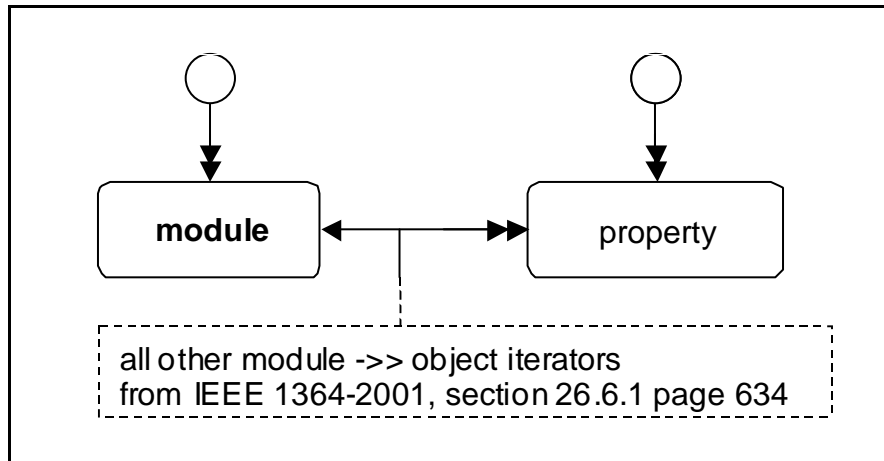


Figure 2-1—Encompassing assertions

- 1) Iterate all assertions in the design: use a NULL reference handle (ref) to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiAssertion, NULL);
while (assertion = vpi_scan(itr)) {
    /* process assertion */
}
```
- 2) Iterate all assertions in an instance: pass the appropriate instance handle as a reference handle to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiAssertion, instanceHandle);
while (assertion = vpi_scan(itr)) {
    /* process assertion */
}
```
- 3) Obtain the assertion by name: extend `vpi_handle_by_name` to also search for assertion names in the appropriate scope(s), e.g.,

```
vpiHandle = vpi_handle_by_name(assertName, scope)
```

NOTES

- 1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.
- 2—These iterators return both assertions and immediate non-temporal checks.
- 3—Unnamed assertions cannot be found by name.

2.3.2 Obtaining static assertion information

The following information about an assertion is considered to be “static”.

- Assertion name
- Instance in which the assertion occurs
- Module definition containing the assertion
- Assertion directive¹

1) *assert*

2) *check*

3) *assume*

4) *cover*

5) Any assertion updates from the *SV-AC*.

- Assertion source information: the file, line, and column where the assertion is defined.
- Assertion clocking domain/expression²

2.3.2.1 Using `vpi_get_assertion_info`

Static information can be obtained directly from an assertion handle by using `vpi_get_assertion_info`, as shown below.

```
typedef struct t_vpi_source_info {
    PLI_BYTE8 *fileName;
    PLI_INT32 startLine;
    PLI_INT32 startColumn;
    PLI_INT32 endLine;
    PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;
typedef struct t_vpi_assertion_info {
    PLI_BYTE8 *name; /* name of assertion */
    vpiHandle instance; /* instance containing assertion */
    PLI_BYTE8 modname; /* name of module/interface containing
                        assertion */

    vpiHandle clock; /* clocking expression */
    PLI_INT32 directive; /* vpiAssume, ... */
    s_vpi_source_info sourceInfo;
    s_vpi_assertion_info, *p_vpi_assertion_info;
} int vpi_get_assertion_info (assert_handle, p_vpi_assertion_info);
```

This call obtains all the static information associated with an assertion.

The inputs are a valid handle to an assertion and a pointer to an existing `s_vpi_assertion_info` data structure. On success, the function returns TRUE and the `s_vpi_assertion_info` data structure is filled in as appropriate. On failure, the function returns FALSE and the contents of the assertion data structure are unpredictable.

Assertions can occur in modules and interfaces: assertions defined in modules (by using VPI) shall have instance information; assertions in interfaces shall have a NULL instance handle. In either case, modname is

¹The exact directives need to be adjusted per developments in the *SV-AC* committee.

²Any specific clocking domain information needs to be adjusted per developments in the *SV-AC* committee.

the definition name.

NOTES

1—The assertion clock is an event expression supplied as the clocking expression to the assertion declaration, i.e., this is a handle to an arbitrary Verilog event expression.

2—A single call returns all the information for efficiency reasons.

2.3.2.2 Extending `vpi_get()` and `vpi_get_str()`

In addition to `vpi_get_assertion_info`, the following existing VPI functions are also extended:

`vpi_get()`, `vpi_get_str()`

`vpi_get()` can be used to query the following VPI properties from a handle to an assertion.

`vpiAssertionDirective`

returns one of `vpiAssertProperty` or `vpiCheckProperty`.

`vpiLineNo`

returns the line number where the assertion is declared.

`vpi_get_str()` can be used to obtain the following VPI properties from an assertion handle.

`vpiFileName`

returns the filename of the source file where the assertion was declared.

`vpiName`

returns the name of the assertion.

`vpiFullName`

returns the fully qualified name of the assertion.

2.4 Dynamic information

This section defines how to place assertion system and assertion callbacks.

2.4.1 Placing assertion “system” callbacks

Use `vpi_register_cb()`, setting the `cb_rtn` element to the function to be invoked and the reason element of the `s_cb_data` structure to one of the following values, to place an assertion system callback.

`cbAssertionSysInitialized`

occurs after the system has initialized. No assertion-specific actions can be performed until this callback completes. The assertion system can initialize before `cbStartOfSimulation` does or afterwards.

`cbAssertionSysStart`

the assertion system has become active and starts processing assertion attempts. This always occur after `cbAssertionSysInitialized`. By default, the assertion system is “started” on simulation startup, but the user can delay this by using assertion system control actions.

`cbAssertionSysStop`

the assertion system has been temporarily suspended. While stopped no assertion attempts are processed and no assertion-related callbacks occur. The assertion system can be stopped and resumed an arbitrary number of times during a single simulation run.

`cbAssertionSysEnd`

occurs when all assertions have completed and no new attempts will start. Once this callback occurs no

more assertion-related callbacks shall occur and assertion-related actions shall have no further effect. This typically occurs after the end of simulation.

`cbAssertionSysReset`

occurs when the assertion system is reset, e.g., due to a system control action.

The callback routine invoked follows the normal VPI callback prototype and is passed an `s_cb_data` containing the callback reason and any user data provided to the `vpi_register_cb()` call.

2.4.2 Placing assertions callbacks

Use `vpi_register_assertion_cb()` to place an assertion callback; the prototype is:

```
vpiHandle vpi_register_assertion_cb(
    vpiHandle, /* handle to assertion */
    PLI_INT32 event, /* event for which callbacks needed */
    PLI_INT32 (*cb_rtn)( /* callback function */
        PLI_INT32 event,
        vpiHandle assertion,
        p_vpi_attempt_info info,
        PLI_BYTE8 *userData),
    PLI_BYTE8 *user_data /* user data to be supplied to cb */
);
typedef struct t_vpi_assertion_step_info {
    PLI_INT32 matched_expression_count;
    vpiHandle *matched_exprs; /* array of expressions */
    p_vpi_source_info *exprs_source_info; /* array of source info */
    PLI_INT32 stateFrom, stateTo; /* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;
typedef struct t_vpi_attempt_info {
    union {
        vpiHandle failExpr;
        p_vpi_assertion_step_info step;
    } detail;
    s_vpi_time attemptTime,
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

where event is any of the following.

`cbAssertionStart`

an assertion attempt has started. For most assertions one attempt starts each and every clock tick.

`cbAssertionSuccess`

when an assertion attempt reaches a success state.

`cbAssertionFailure`

when an assertion attempt fails to reach a success state.

`cbAssertionStepSuccess`

the progress of one “thread” along an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis, rather than on a per-assertion basis.

`cbAssertionStepFailure`

failure to progress along one “thread” along an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis, rather than on a per-assertion basis.

`cbAssertionDisable`

whenever the assertion is disabled (e.g., as a result of a control action).

`cbAssertionEnable`

whenever the assertion is enabled.

`cbAssertionReset`
whenever the assertion is reset.

`cbAssertionKill`
when an attempt is killed (e.g., as a result of a control action).

These callbacks are specific to a given assertion; placing such a callback on one assertion does not cause the callback to trigger on an event occurring on a different assertion. If the callback is successfully placed, a handle to the callback is returned. This handle can be used to remove the callback via `vpi_remove_cb()`. If there were errors on placing the callback, a `NULL` handle is returned. As with all other calls, invoking this function with invalid arguments has unpredictable effects.

Once the callback is placed, the user-supplied function shall be called each time the specified event occurs on the given assertion. The callback shall continue to be called whenever the event occurs until the callback is removed.

The callback function shall be supplied the following arguments:

- 1) the event that caused the callback
- 2) the handle for the assertion
- 3) a pointer to an attempt information structure
- 4) a reference to the user data supplied when the callback was placed.

The *attempt information structure* contains details relevant to the specific event that occurred.

- On disable, enable, reset and kill events, the `info` field is absent (a `NULL` pointer is given as the value of `info`).
- On start and success events, only the *attempt time* field is valid.
- On a failure event, the *attempt time* and `detail.failExpr` are valid.
- On a step callback, the *attempt time* and `detail.step` elements are valid.

On a step callback, the `detail` describes the set of expressions matched in satisfying a step along the assertion, along with the corresponding source references. In addition, the `step` also identifies the source and destination “states” needed to uniquely identify the path being taken through the assertion. *State ids* are just integers, with 0 identifying the origin state, 1 identifying an accepting state, and any other number representing some intermediate point in the assertion. It is possible for the number of expressions in a step to be 0 (zero), which represents an unconditional transition. In the case of a failing transition, the information provided is just as that for a successful one, but the last expression in the array represents the expression where the transition failed.

NOTES

- 1—In a failing transition, there shall always be at least one element in the expression array.
- 2—Placing a step callback results in the same callback function being invoked for both success and failure steps.

2.5 Control functions

This section defines how to obtain assertion system control and assertion control information.

2.5.1 Assertion system control

Use `vpi_control()`, with one of the following operators and no other arguments, to obtain assertion system control information.

Usage example: `vpi_control(vpiAssertionSysReset)`

`vpiAssertionSysReset`

discards all attempts in progress for all assertions and restore the entire assertion system to its initial state.

Usage example: `vpi_control(vpiAssertionSysStop)`

`vpiAssertionSysStop`

considers all attempts in progress as unterminated and disable any further assertions from being started.

Usage example: `vpi_control(vpiAssertionSysStart)`

`vpiAssertionSysStart`

restarts the assertion system after it was stopped (e.g., due to `vpiAssertionSysStop`). Once started, attempts shall resume on all assertions.

Usage example: `vpi_control(vpiAssertionSysEnd)`

`vpiAssertionSysEnd`

discard all attempts in progress and disable any further assertions from starting.

2.5.2 Assertion control

Use `vpi_control()`, with one of the following operators, to obtain assertion control information.

- For the following operators, the second argument shall be a valid assertion handle.

Usage example: `vpi_control(vpiAssertionReset, assertionHandle)`

`vpiAssertionReset`

discards all current attempts in progress for this assertion and resets this assertion to its initial state.

Usage example: `vpi_control(vpiAssertionDisable, assertionHandle)`

`vpiAssertionDisable`

disables the starting of any new attempts for this assertion. This has no effect on any existing attempts. or if the assertion already disabled. By default, all assertions are enabled.

Usage example: `vpi_control(vpiAssertionEnable, assertionHandle)`

`vpiAssertionEnable`

enables starting new attempts for this assertion. This has no effect if assertion already enabled or on any existing attempts.

- For the following operators, the second argument shall be a valid assertion handle and the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure).

Usage example: `vpi_control(vpiAssertionKill, assertionHandle, attempt)`

`vpiAssertionKill`

discards the given attempts, but leaves the assertion enabled and does not reset any state used by this assertion (e.g., `past()` sampling).

Usage example: `vpi_control(vpiAssertionDisableStep, assertionHandle, attempt)`

`vpiAssertionDisableStep`

disables step callbacks for this assertion. This has no effect if stepping not enabled or it is already disabled.

- For the following operator, the second argument shall be a valid assertion handle, the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure) and the fourth argument shall be a “step control” constant.

Usage example: `vpi_control(vpiAssertionEnableStep, assertionHandle, attempt,
vpiAssertionClockSteps)`

`vpiAssertionEnableStep`

enables step callbacks to occur for this assertion attempt. By default, stepping is disabled for all assertions. This call has no effect if stepping is already enabled for this assertion and attempt, other than possibly changing the stepping mode for the attempt if the attempt has not occurred yet. The stepping mode of any particular attempt cannot be modified after the assertion attempt in question has started.

NOTE—In this release, the only step control constant available is `vpiAssertionClockSteps`, indicating callbacks on a per assertion/clock-tick basis. The assertion clock is the event expression supplied as the clocking expression to the assertion declaration. The assertion shall “advance” whenever this event occurs and, when stepping is enabled, such events shall also cause step callbacks to occur.

