System Verilog Assertion API

João Geada

Change Log

Version	Date	Authors	Description
0.1	11/25/2002	Joao	First draft proposal for incorporating Synopsys assertions API donation into SV framework.

Table of Contents

1 Requirements

To provide an API into the SV assertion capabilities providing sufficient capabilities to:

1. enable user's C code to react to assertion events
2. enable 3rd party assertion "waveform" dumping tools to be written
3. enable 3rd party assertion coverage tools to be written
4. enable 3rd party assertion debug tools to be written

The interface should also be readily extensible/adaptable so that it can easily be kept in sync with progress made by the sv-ac committee.
In addition, this interface should not unnecessarily duplicate any existing PLI/VPI interfaces.

1.1 Naming conventions[1]

All elements provided by this interface must have the unique prefix "sva"
Types names will start with "sva" followed by Capitalized words with no separators, eg:

```
      svaAssertID
```

Function names will start with "sva_" followed by all lowercase words separated by '_', eg:

```
      sva_get_assertion_by_name()
In addition, enumeration entries for "actions" will end with the
character 'A', whereas enumeration entries for "events" will end with
'E'
```

2 Static Information

2.1 Obtaining assertion handles

1. Iterate over all assertions

   ```
   typedef enum { svaFirstItemI, svaNextItemI } svaIter;
   typedef void* svaAssertID;
   svaAssertID sva_iterate_assertions(svaClientID, svaIter);
   ```

   The iterator has to be initialized by making a call with svaFirstItem; if there are assertions in the design this returns the handle to the first assertion and initializes a client iterator data structure. Repeated calls with svaNextItem will return handles to other assertions in the design until all assertions have been visited. After the last handle has been returned, a further call with svaNextItem will return a NULL handle. It is undefined what happens if svaNextItem is invoked without a prior call to svaFirstItem.
   Only one assertion iterator can be in-progress for a given svcClientID
2. Iterate over all assertions in an instance

   ```
   svaAssertID sva_instance_assertions(svaClientID,
                                       const char *instance,
                                       svaIterEnum);
   ```

   Similar to global iterators above. instance is a fully qualified name of an instance in

[1] João comment: these rules are meant to be consistent with the naming conventions used by PLI and VPI interfaces

the design. Instance must be provided *only* when svaFirstItem is given, and on all successive calls with svaNextItem instance must be NULL.

3. Iterate over all assertions in a module
   no proposal
4. Obtain assertion by name
```
svcAssertID svc_get_assertion_by_name(svaClientID,
                                       const char *name);
```
   returns the handle for the named assertion or NULL. 'name' must be a fully qualified[2] assertion name.

Note that all assertion handles (svaAssertID) are to a *specific assertion* in a *specific instance*.

2.2 Obtaining static assertion information

The following information about an assertion is considered to be "static":

1. Assertion name
2. Instance in which the assertion occurs
3. Module definition containing the assertion
4. Assertion type[3]:
   a. check
   b. forbid
   c. event
   d. etc as necessary by assertion updates in sv-ac
5. Assertion directive[4]:
   a. property
   b. assume
   c. etc as necessary by assertion updates in sv-ac
6. Assertion source information
   file, line and column where assertion defined
7. Assertion clocking domain/expression[5]

Static information can be obtained directly from an svaAssertID without requiring any assertion attempts to be started or completed.

```
typedef enum { svaCheckType,
            svaForbidType,
            svaEventType } svaAssertType;
typedef enum { svaProperty, svaAssume } svaAssertDirective;
typedef struct { char *fileName;
            int startLine; int startColumn;
            int endLine; int endColumn;
} svaSourceInfo;
typedef struct { char *assertName;
            char *instanceName;
            char *moduleName;
```

[2] Using regular Verilog hierarchical 'dot' notation
[3] Exact types will have to be adjusted as per developments in the sv-ac committee
[4] Exact directives will have to be adjusted as per developments in the sv-ac committee
[5] Specific clocking domain info will have to be adjusted as per developments in the sv-ac committee

```
            char *clockDomain;
            svaAssertType assertType;
            svaAssertDirective assertDirective;
            svaSourceInfo sourceInfo;
} svaAssertStaticInfo;
int svaGetAssertStaticInfo(svaClientID,
                        svaAssertID,
                        svaAssertStaticInfo*);
```

This call is used to obtain all the static information associated with an assertion. **Note:** a single call returns all the information for efficiency reasons, as most clients will require most of the data; for efficiency one roundtrip through the API is better than multiple. The inputs are a valid svaClientID, svaAssertID and a pointer to an existing svaAssertStaticInfo datastructure. On success the function returns TRUE and the svaAssertStaticInfo datastructure will be filled in as appropriate. On failure, the function returns FALSE and the contents of the the static info datastructure are unpredictable.

2.2.1 Additional static information

There are additional items that could be obtained statically from assertion, including:

1. Structure of assertion
2. Set of HDL variables used by assertion
3. Set of HDL expressions used by assertion

No proposal for these items.

3 Dynamic Information

3.1 Placing assertion "system" callbacks

1. Assertion system initializing
   Occurs before assertion system has initialized. No assertion specific actions can be performed at this time
2. Assertion system started
   Assertion system has completed initialization and is about to become active. Typically will occurs just before start of simulation
3. Assertion system finish
   Occurs when all assertions have completed and no new attempts will start. Typically occurs after the end of simulation.
4. Assertion system reset
   Occurs when the assertion system is reset, eg due to a system control action
5. Assertion system terminated
   Occurs if assertion system is terminated, eg due to a system control action

```
typedef enum {
      svaSystemInitializingE,
      svaSystemStartedE,
      svaSystemFinishE,
      svaSystemResetE,
      svaSystemTerminateE,
} svaSystemEvent;
typedef void (*svaSystemCallback)(svaSystemEvent,
                                  void *userData);
```

```
int svaAddSystemCallback(svaClientID,
                        svaSystemEvent,
                        svaSystemCallback,
                        void *userData);
int svaDeleteSystemCallback(svaClientID,
                              svaSystemEvent,
                              svaSystemCallback,
                              void **userData);
```

3.2 Placing assertions callbacks

Callbacks can be placed on any occurrence of the following per-assertion events:

1. Assertion attempt start
   An assertion attempt has started. For most assertions one attempt will start each and every clock tick
2. Assertion success
   When an assertion attempt reaches a success state
3. Assertion failure
   When an assertion attempt fails to reach a success state
4. Successful transition
   progress of one “thread” along an attempt
5. Failed transition
   failure of one “thread” to progress along the attempt
6. Assertion disabled
   Whenever the assertion is disabled (eg as a result of a control action)
7. Assertion enabled
   Whenever the assertion is enabled
8. Assertion reset
   Whenever the assertion is reset
9. Assertion killed
   When an attempt is killed (eg as a result of a control action)

These callbacks are specific to a given assertion; placing such a callback on one assertion will not cause the callback to trigger on an event occurring on a different assertion

```
typedef enum {
      svaAttemptStartE,
      svaAttemptSuccessE,
      svaAttemptFailureE,
      svaSuccessfullTransitionE,
      svaFailedTransitionE,
      svaAssertionDisabledE,
      svaAssertionEnabledE,
      svaAssertionResetE,
      svaAssertionKilledE,
      svaAssertionAllE,       /* composite event = all events */
} svaAssertEvent;
typedef void (*svaAssertCallback)(svaAssertEvent,
                                    svaAssertID,
                                    svaAssertAttemptID,
                                    void *userData);
int svaAddAssertCallback(svaClientID,
                        svaAssertID,
                        svaAssertEvent,
                        svaAssertCallback,
                        void *userData);
int svaDeleteAssertCallback(svaClientID,
                              svaAssertID,
                              svaAssertEvent,
                              svaAssertCallback,
                              void **userData);
```

This call places an assertion callback. The call return TRUE if successful, FALSE otherwise. Attempting to place the same callback (ie same callback function) on the same assertion by the same client repeatedly has no effect other than changing the userData associated with that callback. Supplying invalid IDs or an invalid event to this call may have unpredictable effects.

Transition callbacks will occur for every transition along each attempt in the FSM representing the assertion. Note that multiple transitions may occur as a result of a single clocking event.

Once the callback is placed the user-supplied function will be called each time this event occurs on the given assertion. The callback will continue to be called whenever the event occurs until the callback is removed. The callback function will be supplied the event that caused the callback, the handle for the assertion, a handle to the specific assertion attempt (if appropriate, NULL otherwise), and a reference to the user data supplied when the callback was placed.

A callback can be removed once placed by the svaDeleteAssertCallback() function by supplying the appr. arguments. Attempting to remove a non-existent callback has no effect. If a callback is successfully removed, TRUE is returned and the userData pointer will be set to the userData supplied when the callback was placed. If no callback is removed FALSE is returned. As with all other calls, invoking this function with invalid arguments will have unpredictable effects.

3.3 Obtaining dynamic information

Success: attempt start time

Failure: attempt start time, expression at which assertion failed

Transition: expression(s) causing transition, from state, to state, next state expression

```
typedef struct {
        char *failExpression;           /* expression causing assert to fail */
        long long attemptTime;          /* time when assert attempt started (in fs) */
} svaAssertAttemptInfo;
int svaGetAttemptInfo(svaClientID,
                      svaAssertID,
                      svaAssertAttemptID,
                      svaAssertAttemptInfo*);
```

```
typedef struct {
      int expressionCount;
      char **expressionsEvaluated;
      int stateFrom;
      int stateTo;
      char *nextExpression;
} svaTransitionInfo;
```

```
int svaGetTransitionInfo(svaClientID,
                         svaAssertID,
                         svaAssertAttemptID,
                         svaTransitionInfo*);
```

This function gets the information related to a transition. The input are a valid svaClientID, svaAssertID, svaAssertAttemptID (obtained from a transition callback), and a pointer to an existing svaTransitionInfo datastructure, which will be filled in by the API on successful completion of the call. If the call is successful it returns TRUE, FALSE otherwise.
Neither the stateFrom nor stateTo ids are meaningful other than for uniquely identifying states in the FSM from which transitions occur. The expressionCount has the number of expressions evaluated by the assertion in the processing of this transition. expressionsEvaluated is the array of expressions evaluated (dimension == expressionCount). nextExpression is the expression that will be evaluated in the next clock for this attempt and thread. **Note** that this expression can be a union of the expressions that will actually be used during the evaluation of the next transition.
Note that the expression array co by this function will not persist across any subsequent call of this same function. If user requires persistence of this information it is their responsibility to copy it.
Note this callback and information is primarily intended to support debuggers and runtime analysis of assertions, rather than for user applications.

4 Control Functions

4.1 Assertion System control

1. Initialize the API and register a user of the API
   ```
   typedef void *svaClientID;
   svaClientID svaRegisterClient();
   ```

   This call must be made prior to any use of the API by a client application. The returned client identifier must be supplied to all other API calls. The client handle is used to preserve client specific information including:
   - userData per callback
   - state of iterators
   - callbacks placed
2. Control the assertion system
   - Reset assertion engine
     discard all attempts in progress for all assertions and restore the entire assertion system to its initial state.
   - Finish assertion system
     consider all attempts in progress as unterminated and disable any further assertions from being started.
   - Terminate assertion system
     discard all attempts in progress and disable any further assertions from starting.

```
typedef enum {
      svaResetSysA,
```

```
      svaFinishSysA,
   svaTerminateSysA } svaSysAction;
int svaDoSysAction(svaClientID, svaSysAction, void *userData);
```

The above call performs a control action upon the whole assertion system. The call returns TRUE if action was successful, FALSE otherwise. Action to be performed is specified by providing the appropriate enum. Only one action can be performed per call. Supplying a value other than one of the legal enum values above may have unpredictable consequences. The userData field will be propagated to any callbacks waiting on assertion system events.

4.2 Assertion control

- Reset assertion
  discards all current attempts in progress for this assertion and resets this assertion to its initial state
- Disable new attempts
  disables the starting of any new attempts for this assertion. Has no effect on any existing attempts. No effect if assertion already disabled. Note that by default all assertions are enabled.
- Enable new attempts
  Enables starting new attempts for this assertion. No effect if assertion already enabled. No effect on any existing attempts.
- Kill existing attempts
  Discards any existing attempts but leaves assertion enabled and does not reset any state used by this assertion (eg past() sampling)
- Enable assertion tracing
  Enables state transition callbacks to occur for this assertion. No effect if tracing already enabled. Note that by default tracing is disabled for all assertions.
- Disable assertion tracing
  Disables state transition callbacks for this assertion. No effect if tracing already disabled

```
typedef enum {
      svaResetAssertA,
      svaDisableAttemptsAssertA,
      svaEnableAttemptsAssertA,
      svaKillAssertA,
      svaEnableTraceAssertA,
      svaDisableTraceAssertA } svaAssertAction;
int svaDoAssertAction(svaClientID, svaAssertAction,
                        svaAssertID, void *userData);
```

The above call performs a control action upon a specific assertion, identified by its svaAssertID. The call returns TRUE if action was successful, FALSE otherwise. Action to be performed is specified by providing the appropriate enum. Only one action can be performed per call. Supplying a value other than one of the legal enum values above or supplying an invalid svaAssertID may have unpredictable consequences. The userData field will be propagated to any callbacks waiting on events for the given assertion.