Detecting end-of-test conditions- A Shutdown Manager implementation for an AVM based verification environment

Dr. Ambar Sarkar, Matt Trostel
Paradigm Works, Inc.

Abstract:

State-of-the-art functional verification environments are typically implemented using a distributed architecture, with a number of components such as transactors, monitors, checkers, scoreboards, and so on. All of these components, including the design under test, have complex interaction patterns. In addition, with constrained random stimuli becoming the norm, it is extremely hard to predict how long the test be allowed to run. If not run long enough, we end up with inconclusive results. If run too long, a lot of the simulation resources are wasted, reducing overall verification efficiency.

To efficiently detect the end of test in an AVM[(1)]-based verification environment, we need a Shutdown Manager(SM) utility which provides two services: 1) normal end of test detection, and 2) activity monitoring. The implementation is based on a simple objection-based scheme, where each component raises an objection when busy and lowers it to indicate it's done for the moment. This article gives an overview of the implementation of a SM as well as examples of using it in an AVM-based verification environment.

Introduction

With the adoption of constrained random-based methodology, verification environments are becoming increasingly sophisticated, containing of a number of concurrently active threads, with complex interaction patterns. When writing a test in such an environment, especially with random stimuli, it is quite a challenge for the verification engineer to determine when to declare the test completed and stop a simulation session.

A test may complete for any number of reasons. Examples include, but are not limited to: absolute time (specified in number of simulation cycles, or amount of time in ns, for example), the number of transactions to be issued, or when a functional coverage goal has been met. There are a number of possible conditions under which a test may complete, and the challenge is to determine when those conditions are being met. For simple, directed testbenches, a timeout based on the number of transactions can be applied to declare a test done. Another straightforward approach is to run the test for a fixed amount of cycles after the last transaction is generated. Such guesswork is inadequate when the number and type of transactions generated are large and random, especially for complex interaction patterns.

In this article, we describe a Shutdown Manager (SM) base class, written in SystemVerilog, that encapsulates the end-of-test detection functionality and provides a mechanism for notifying the testbench that a test has ended. This class was developed as part of SystemVerilog FrameWorks$^{TM(2)}$, a productivity enhancement tool that help verification teams get a jumpstart on creating and maintaining SystemVerilog based verification environments. Next, we provide examples to show how an instance of this class can be easily integrated into an AVM based verification environment.

Shutdown Manager

The Shutdown Manager base class is intended to be straightforward, flexible, and easy to understand. This base class provides the following two services:

Normal Test Completion Service: Provides notification of normal end-of-test conditions using raising and lowering of objections

Activity Monitor Service: Provides notification of abnormal end-of-test conditions based on a user-configurable design/environment inactivity watchdog timer value

Normal Test Completion Service

The normal test completion service is used for tests which complete gracefully and do not end due to errors or activity timeouts. An objection mechanism is used to implement this service. Any element of the verification environment, typically an instance of a class derived from *avm_named_component*, which initiates or processes a stimulus, is responsible for raising an objection to the test completing. When the verification element observes the expected response to the stimulus, it may then lower the objection to the test completion.

Objection Counter

The test completion service indicates test completion when there are no objections raised. The objection mechanism can be thought of simply as a counter. The counter is incremented each time an objection is raised, and decremented each time an objection is lowered. The current implementation allows for multiple kinds of objections, meaning there can be multiple counters concurrently active. The predefined enumerated type SVF_ObjectionKind_e is used to describe all objection types for which the objection mechanism is used. There is a predefined enumerated type SVF_ObjectionKind_e, initially set to TEST_DONE. TEST_DONE is the most basic objection type which may be used, although others could be envisioned that would enable the SM to manage progression of the verification components through different phases of test execution. A unique counter is maintained for each SVF_ObjectionKind_e.

Example:
typedef enum [TEST_DONE] SVF_ObjectionKind_e;

The Shutdown Manager objection mechanism also provides debugging aids. A user may find it useful to determine the component which raised or lowered an objection. To facilitate this, any component using the Shutdown Manager is required to register itself with the SM. This registration is implemented via a SM function call which returns a unique tag to the component. This tag could optionally simply be the hierarchical instance name of the component, as managed in the avm_named_component base class. This tag can then be used for all subsequent invocations of raise/dropObjection (see table 3.1 for a description of the SM objection user interface). A unique counter is maintained for each tag registered with the SM. In addition, the simulation timestamp is recorded each time an objection is raised or lowered.

The Table 3.1 enumerates the user interface methods provided by this base class.

Method	Description
task new ( )	Constructor which creates an instance of the Shutdown Manager object.
function string registerSM()	Registers a component with the SM and returns a unique tag (as a string).
task raiseObjection (string a_tag, SVF_ObjectionKind_e a_objectionKind)	Raises an objection to the predefined enumerated type passed via the a_objectionKind parameter.
task dropObjection (string a_tag, SVF_ObjectionKind_e a_objectionKind)	Drops an objection to the predefined enumerated type passed via the a_objectionKind parameter. An error is issued if dropObjection is called and there is no corresponding raiseObjection for a_objectionKind.
function integer getObjectionCount (SVF_ObjectionKind_e a_objectionKind)	Returns the objection count for the current number of objections to a_objectionKind.
event testComplete_ev;	Event which is triggered when all completions have been dropped.
task resetObjections ( )	Clears all objections pending.

Table 0.1: Methods of the SVF Shutdown Manager base class normal test completion service

Activity Monitor Service

The activity monitor service provides a mechanism to indicate when the design has been inactive for a user-specified period of time. This service is useful for detecting, for example, when the design has entered a deadlock condition and ceases all activity on its interfaces.

The activity monitor service keeps track of a timeout counter, which is loaded to a default value if the user does not specify it. This counter is reset every time new activity is detected, or when the reset( ) method is called. The startActivity( ) method is provided to load the counter with the user-specified timeout value. Once the counter has been loaded via startActivity( ), the count will decrement on every clock tick. When the activity counter reaches a value of zero, an event is triggered to notify the testbench of an activity timeout. It is left up to the testbench to determine the subsequent course of action. Table 3.2 enumerates the methods provided by the activity monitor service.

Method/Property	Description
task new ( )	Constructor which creates an instance of the SVF Shutdown Manager object.
task startActivity ( )	Loads the timeout counter with the value specified in numTimeoutCycles (i.e., when there is design activity).
task stopActivity ( )	Disables the activity monitor from decrementing the activity cycle counter. The activity monitor may be re-started via startActivity ( ).
integer numTimeoutCycles = 20000;	User-constrainable field to control count value which gets loaded for every call to reset( ) or startActivity( );
event activityTimeout_ev;	Event which is triggered when the activity counter has reached 0, indicating an activity timeout.

Table 0.2: Methods of the SVF Shutdown Manager base class activity monitor service

.

Examples

Here are two examples of using the SM class in a typical verification environment. The first shows how to instantiate and use a SM. It also shows how the top-level environment can detect the end of the test. In the second example, we show how "execution gating" can be implemented, where the individual phases of the various verification components can be made to progress through individual stages in a synchronized manner.

Example 1. Creating an instance of shutdown manager

The following code snippets show how various components in the verification environment can use the shutdown manager. Basically, the top-level environment object creates an instance of the SM object and passes it to various components during their

initialization. The individual components then raise and lower the objections as needed. Finally, the top-level environment object waits for all the objections to be dropped before declaring the test completed.

```
// A typical verification component using the shutdown manager
class my_component extends avm_verification_component;
        pw_svf_shutdown_mgr m_sm; // Keep a reference to the SM object

        function new(       string name,
                            avm_named_component parent = null,
                            pw_svf_ shutdown_mgr sm); // Pass a handle to the SM object
                super.new(name, parent);
                this.m_sm = sm;
        endfunction:new;

        // Typical call to execute this component
        task run;
                do
                        begin
                                // Raise the objection. The test will
                                // not be complete until this objection is dropped
                                m_sm.raiseObjection(this.name); // Raise objection

                                // Complete the task for this iteration
                                ….
                                m_sm.dropObjection(); // Drop objection
                        end
                while (1);
        endtask;run;
endclass;

// Create an instance of sm in the top-level environment
class verify_env extends avm_env;
        …
        // Declare the instance
        pw_svf_shutdown_mgr m_sm;

        // Various components in this interface
        my_component m_cmp

        // Creates a new instance of the environment
        // Creates an instance of the SM
        // Passes it on to various participating components
        function new(string name);
                …
                this.m_sm = new ();
                m_cmp = new(.name("Component"), .sm(this.m_sm));
                …
        endfunction:new;

        task execute;
                // Fork off the main execution thread
                fork
                        // do the main work
                        do_main_t();
                join_none

                // Wait until all the components are done
                // Or there is an activity timeout
                fork
                        begin
                                @(m_sm.testComplete_ev);
```

```
                                        avm_report_message("Test completed normally!");
                                end
                                begin
                                        @(m_sm.activityTimeout_ev);
                                        avm_report_error("Activity timeout!!!!");
                                end
                        join_any

        endtask:execute;
endclass
```

Example 2. Execution gating.

It is also possible to implement an "execution gating" scenario, where the top level environment can ensure that the various phases of the execution proceed only when all the components are done with their respective phases. In the following code, a possible implementation is shown, where each participating component is expected to raise and lower their corresponding objections related to each phase. The phases gated are the configuration and execution phases.

```
// Create an instance of sm in the top-level environment
class verify_env_with_gating extends avm_env;
…
        task configure;
                // Reset objections before starting this phase
                m_sm.resetObjections();
                fork
                        // Spawn off various configuration tasks
                        …
                join_none

                // Wait until all the components are done with this phase
                // Or there is an activity timeout
                fork
                        begin
                                @(m_sm.testComplete_ev);
                                avm_report_message("verify_env_with_gating::configure completed
normally!");
                        end
                        begin
                                @(m_sm.activityTimeout_ev);
                                avm_report_error("verify_env_with_gating::configure activity timeout!!!!");
                        end
                join_any
        endtask:configure

        task execute;
                // Reset objections before starting this phase
                m_sm.resetObjections();
                fork
                        // Spawn off various execute tasks
                        …
                join_none

                // Wait until all the components are done with this phase
                // Or there is an activity timeout
                fork
                        begin
                                @(m_sm.testComplete_ev);
```

```
                                        avm_report_message("verify_env_with_gating::execute completed
normally!");
                                end
                                begin
                                        @(m_sm.activityTimeout_ev);
                                        avm_report_error("verify_env_with_gating::execute activity timeout!!!!");
                                end
                        join_any
                endtask:connect
                …

endclass
```

Conclusion

The Shutdown Manager class offers an efficient, easy to use approach to detect the end of tests, or the completion of individual test phases in sophisticated verification environments. We have shown how the SM can be instantiated in the AVM environment to coordinate the execution of individual verification components, and how the components themselves can be designed to utilize the SM to report their progress. The flexibility of the AVM allows you to define a library of project- or company-specific library components that can be layered on top of the AVM base classes, and also used in conjunction with the AVM, to customize your environment for your particular needs. The open-source nature of the AVM, and Mentor's willingness to adapt, enhance and propagate the AVM, means that components such as the SM can easily be contributed by third-party providers and promulgated to a wide audience in future versions of the AVM. The authors are looking forward to participating in this process.

References

1) Verification Cookbook, AVM Library Documentation, Mentor Graphics
2) SystemVerilog FrameWorks™ User Guide, Paradigm Works