APPENDIX A  VMM STANDARD LIBRARY SPECIFICATION

This appendix specifies the detailed behavior of a set of base and utility classes that can be used to implement a VMM-compliant verification environment and verification components. The actual implementation of these classes is left to each tool provider. Chapter 4 provides detailed guidelines on how to use these classes.

VMM_ENV

The class is a base class used to implement verification environments. The guidelines covering the implementation of verification environments based on this class can be found in section titled "Simulation Control" on page 124.

vmm_log log;
Message service interface for the verification environment. This property is set by the constructor using the specified environment name and may be modified at run time.

vmm_notify notify;
enum {GEN_CFG, BUILD, RESET_DUT, CFG_DUT, START, WAIT_FOR_END,
Notification service interface and predefined notifications used to indicate the progression of the verification environment. The predefined notifications are used to signal the start of the corresponding predefined virtual methods. All notifications are ON/OFF.

```plaintext
function new(string name = "Verif Env");
```

Creates an instance of the verification environment, with the specified name. The name is used as the name of the message service interface.

```plaintext
task run()
Runs all remaining steps of the simulation, including vmm_env::stop(), vmm_env::cleanup() and vmm_env::report(). This method must be explicitly invoked in the test programs.
```

```plaintext
virtual function void gen_cfg();
```

Randomizes the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the vmm_env::build() method.

```plaintext
virtual function void build();
```

Builds the verification environment according to the value of the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the vmm_env::reset_dut() method.

```plaintext
virtual task reset_dut();
```

Physically resets the DUT to make it ready for configuration. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the vmm_env::cfg_dut() method.

```plaintext
virtual task cfg_dut();
```

Configures the DUT according to the value of the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the vmm_env::start() method.
virtual task start();
Starts all the components of the verification environment to start the actual test. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the `vmm_env::wait_for_end()` method.

virtual task wait_for_end();
Waits for an indication that the test has reached completion or its objective—whatever these may be. When this task returns, it signals that the end of simulation condition has been detected. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the `vmm_env::stop()` method.

virtual task stop();
Stops all the components of the verification environment to terminate the simulation cleanly. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the `vmm_env::cleanup()` method.

virtual task cleanup();
Performs clean-up operations to let the simulation terminate gracefully. Clean-up operations may include letting the DUT drain off all buffered data, reading statistics registers in the DUT and sweeping the scoreboard for leftover expected responses. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the `vmm_env::run()` method.

virtual task report();
Reports final success or failure of the test and close all files. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the `vmm_env::run()` method.
The `vmm_log` class used implements an interface to the message service. The guidelines covering the usage of the message service can be found in section titled "Message Service" on page 134.

Several methods apply to multiple message service interfaces, not just the one where the method is invoked. All message service interfaces that match the specified name and instance name are affected by these methods. If the name or instance name is enclosed between slashes (e.g., "/.../"), then they are interpreted as `sed`-style regular expressions. If a value of "" is specified, then the name or instance name of the current message service interface is specified. If the `recurse` parameter is `TRUE` (i.e., non-zero), then all interfaces logically under the matching message service interfaces are also specified.

```cpp
function new( string name,
    string instance,
    vmm_log under = null);
```

Creates a new instance of a message service interface, with the specified interface name and instance name. Furthermore, a message service interface can optionally be specified as hierarchically below another message service instance to create a logical hierarchy of message service interfaces.

```cpp
virtual function void is_above(vmm_log log);
```

Specifies that this message service instance is hierarchically above the specified message service interface. This method is the corollary of the `under` argument of the constructor and need not be used if the specified message service interface has already been constructed as being under this message service interface.

```cpp
virtual function vmm_log copy(vmm_log to = null);
```

Copies the configuration of this message service interface to the specified message service interface (or a new interface if none is specified) and returns a reference to the interface copy. The current configuration of the message service interface is copied, except the hierarchical relationship information, which is not modified.

```cpp
virtual function string get_name();
virtual function string get_instance();
```

Returns the name and instance name of the message service interface.
virtual function void list(string name = "./.",
                        string instance = "./.",
                        bit recurse = 0);

Lists all message service interfaces that match the specified name and instance name. If the `recurse` parameter is TRUE (i.e., non-zero), then all interfaces logically under the matching message service interface are also listed.

```cpp
enum { FAILURE_TYP,
      NOTE_TYP,
      DEBUG_TYP,
      TIMING_TYP,
      XHANDLING_TYP,
      REPORT_TYP,
      PROTOCOL_TYP,
      TRANSACTION_TYP,
      COMMAND_TYP,
      CYCLE_TYP,
      INTERNAL_TYP,
      DEFAULT_TYP,
      ALL_TYPS};
```

Enumerated type defining symbolic values for message types used when specifying a message type in properties or method arguments. Table 4-1 on page 135 describes the purpose of the various message types. The `vmm_log::DEFAULT_TYP` and `vmm_log::ALL_TYPS` are special symbolic values usable only with some control methods and are not used to issue actual messages. Multiple message types can be specified to some control methods by combining the value of the required types using the bitwise-or or addition operator.

```cpp
enum {FATAL_SEV,
      ERROR_SEV,
      WARNING_SEV,
      NORMAL_SEV,
      TRACE_SEV,
      DEBUG_SEV,
      VERBOSE_SEV,
      DEFAULT_SEV,
      ALL_SEVS};
```

Enumerated type defining symbolic values for message severities used when specifying a message severity in properties or method arguments. Table 4-2 on page 136 describes the purpose of the various message severities. The `vmm_log::DEFAULT_SEV` and `vmm_log::ALL_SEVS` are special symbolic values usable only with some control methods and are not used to issue actual
messages. Multiple message severities can be specified to some control methods by combining the value of the required severities using the bitwise-or or addition operator.

```cpp
enum { IGNORE,
      CONTINUE,
      DUMP_STACK,
      STOP_PROMPT,
      DEBUGGER,
      COUNT_ERROR,
      ABORT_SIM,
      DEFAULT_HANDLING};
```

Enumerated type defining symbolic values for simulation handling used when specifying a new simulation handling when promoting or demoting a message using the `vmm_log::modify()` method.

Unless otherwise specified, message types are assigned the following default severity and simulation handling:

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Default Severity</th>
<th>Default Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAILURE_TYP</td>
<td>ERROR_SEV</td>
<td>COUNT_ERROR</td>
</tr>
<tr>
<td>NOTE_TYP</td>
<td>NORMAL_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>DEBUG_TYP</td>
<td>DEBUG_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>TIMING_TYP</td>
<td>WARNING_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>XHANDLING_TYP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSACTION_TYP</td>
<td>TRACE_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>COMMAND_TYP</td>
<td>DEBUG_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>REPORT_TYP</td>
<td>DEBUG_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>PROTOCOL_TYP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLE_TYP</td>
<td>VERBOSE_SEV</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>Any type</td>
<td>FATAL_SEV</td>
<td>ABORT_SIM</td>
</tr>
</tbody>
</table>
virtual function vmm_log_format
    set_format(vmm_log_format fmt);
Globally sets the message formatter to the specified message formatter instance. A reference to the previously used message formatter instance is returned. A default global message formatter is provided.

virtual function string set_typ_image(int typ,
    string image);
Globally replaces the image used to display the specified message type with the specified image. The previous image is returned. Default images are provided.

virtual function string set_sev_image(int sev,
    string image);
Globally replaces the image used to display the specified message severity with the specified image. The previous image is returned. Default images are provided.

Example A-1. Colorizing the severity display on ANSI terminals

    log.set_sev_image(vmm_log::WARNING,
        \033[33mWARNING\033[0m);
    log.set_sev_image(vmm_log::ERROR_SEV,
        \033[31mERROR\033[0m);
    log.set_sev_image(vmm_log::FATAL_SEV,
        \033[41m*FATAL*\033[0m);

virtual function bit start_msg( int typ,
    int sev = DEFAULT_SEV);
Prepares to issue a message of the specified type and severity. If the message service interface is configured to ignore messages of the specified type or severity, the function returns FALSE. It returns TRUE otherwise.

virtual function bit text(string msg = "");
Adds the specified text to the message being constructed. This method specifies a single line of message text and a newline character is automatically appended when the message issued. Additional lines of messages can be produced by calling this method multiple times, once per line. Each additional line is prefixed with the prefix specified in the vmm_log::format() method. If an empty string is specified as message text, all previously specified lines of text are flushed to the output, but the message is not terminated. This method may return FALSE if the message will be filtered out based on the text.
A message must be flushed and terminated by calling the \texttt{vmm\_log::end\_msg()} method to trigger the message display and the simulation handling. A message can be flushed multiple times by calling the \texttt{vmm\_log::text("")} method, but the simulation handling and notification will take effect on the message termination.

If additional lines are produced using the $\texttt{display()}$ system task or other display mechanisms, they will not be considered by the filters, nor included in explicit log files. They may also be displayed out of order if they are produced before the previous lines of the message are flushed.

For single-line messages, the \texttt{'vmm\_fatal()}, \texttt{'vmm\_error()}, \texttt{'vmm\_warning()}, \texttt{'vmm\_note()}, \texttt{'vmm\_trace()}, \texttt{'vmm\_debug()}, \texttt{'vmm\_verbose()}, \texttt{'vmm\_report()}, \texttt{'vmm\_command()}, \texttt{'vmm\_transaction()}, \texttt{'vmm\_protocol()} and \texttt{'vmm\_cycle()} macros can be used as a shorthand notation.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Message Type</th>
<th>Message Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>'vmm_fatal(vmm_log log, string txt);</td>
<td>Failure</td>
<td>Fatal</td>
</tr>
<tr>
<td>'vmm_error(vmm_log log, string txt);</td>
<td>Failure</td>
<td>Error</td>
</tr>
<tr>
<td>'vmm_warning(vmm_log log, string txt);</td>
<td>Failure</td>
<td>Warning</td>
</tr>
<tr>
<td>'vmm_note(vmm_log log, string txt);</td>
<td>Note</td>
<td>Default</td>
</tr>
<tr>
<td>'vmm_trace(vmm_log log, string txt);</td>
<td>Debug</td>
<td>Trace</td>
</tr>
<tr>
<td>'vmm_debug(vmm_log log, string txt);</td>
<td>Debug</td>
<td>Debug</td>
</tr>
<tr>
<td>'vmm_verbose(vmm_log log, string txt);</td>
<td>Debug</td>
<td>Verbose</td>
</tr>
<tr>
<td>'vmm_report(vmm_log log, string txt);</td>
<td>Report</td>
<td>Default</td>
</tr>
<tr>
<td>'vmm_command(vmm_log log, string txt);</td>
<td>Command</td>
<td>Default</td>
</tr>
</tbody>
</table>
virtual function void end_msg();
Flushes and terminates the current message and triggers the message display and the simulation handling. A message can be flushed multiple times using the `vmm_log::text("")` method, but the simulation handling and notification will only take effect on message termination.

virtual function void enable_types( int typs,
string name = ",",
string inst = ",",
bit recursive = 0);

virtual function void disable_types( int typs,
string name = ",",
string inst = ",",
bit recursive = 0);

Specifies the message types to be displayed/disabled by the specified message service interfaces. Message service interfaces are specified by value or regular expression for both the name and instance name. If no name and no instance are explicitly specified, this message service interface is implicitly specified.

If the name or instance named are specified between "/" characters, then the specification is interpreted as a regular expression that must be matched against all known names and instance names, respectively. Both names must match to consider a message service interface as specified. If the `recursive` argument is TRUE, all message service interface hierarchically below the specified message service interfaces are included in the specification, whether their name and instance name matches or not. A message service interface must exist to be specified.
The `types` argument specifies the messages types to enable or disable. Types are specified as the bitwise-or or sum of all relevant types.

By default, all message types are enabled.

```cpp
virtual function void set_verbosity(int severity,
    string name = "",
    string inst = "",
    bit recursive = 0);
```

Specify the minimum message severity to be displayed when sourced by the specified message service interfaces. See the documentation for the `vmm_enable_types()` method for the interpretation of the `name`, `inst` and `recursive` argument and how they are to specify message service interfaces.

The default minimum severity can be changed by using the `+vmm_log_default=<sev>` runtime command-line option, where `<sev>` is the desired minimum severity and is a one of the following: “error,” “warning,” “normal,” “trace,” “debug” or “verbose”. The default verbosity level can be later modified using this method.

The minimum severity level can be globally forced by using the `+vmm_force_verbosity=<sev>` runtime command-line option. The specified verbosity overrides the verbosity level specified using this method.

```cpp
virtual function int get_verbosity();
```

Returns the minimum message severity to be displayed when sourced by this message service interface.

```cpp
virtual function int
    modify( string name = "",
            string inst = "",
            bit recursive = 0,
            int typs = ALL_TYPS,
            int severity = ALL_SEVS,
            string text = "/./",
            int new_typ = UNCHANGED,
            int new_severity = UNCHANGED,
            int handling = UNCHANGED);
```

Modifies the specified message source by any of the specified message service interfaces with the new specified type, severity or simulation handling. The message
can be specified by type, severity, numeric ID or by text pattern. By default, messages of any type, severity, ID or text is specified. A message must match all specified criteria.

This method returns a unique message modifier identifier that can be used to remove it using the `vmm_log::unmodify()` method. All message modifiers are applied in the same order they were defined before a message is issued.

```c++
virtual function void unmodify( int mod_id = -1,
                               string name = "",
                               string instance = "",
                               bit recursive = 0);
```

Removes the specified message modification from the specified message service interfaces. By default, all message modifications are removed.

```c++
virtual function void log_start( int file,
                                string name = "",
                                string instance = "",
                                bit recurse = 0)
```

Appends all messages produced by the specified message service interfaces to the specified file. The `file` argument must be a file descriptor, as returned by the `$fopen()` system task. By default, all message service interfaces append their messages to the standard output. Specifying a new output file does not stop messages from being appended to previously specified files.

```c++
virtual function void log_stop( int file,
                                string name = "",
                                string instance = "",
                                bit recurse = 0);
```

Messages issued by the specified message service interfaces are no longer appended to the specified file. The `file` argument must be a file descriptor, as returned by the `$fopen()` system task. If the specified `file` argument is 0, messages are no longer sent to the standard simulation output and transcript. If the `file` argument is specified as –1, appending to all files, except the standard output, is stopped.

```c++
virtual function void stop_after_n_errors(int n);
```

Aborts the simulation after the specified number of messages with a simulation handling of `COUNT_ERROR` has been issued. This value is global and all messages from any message service interface count toward this limit. A zero or negative value
specifies no maximum. The default value is 10. The message specified by the \texttt{vmm\_log\_format::abort\_on\_error()} is displayed before the simulation is aborted.

\begin{verbatim}
virtual function int
    get_message_count( int severity = ALL_SEVS,
                      string name = "",
                      string instance = "",
                      bit recurse = 0);
\end{verbatim}

Returns the total number of messages of the specified severities that have been issued from the specified message service interfaces. Message severities can be specified as a sum of individual message severities to specify more than one severity.

\begin{verbatim}
virtual function int
    create_watchpoint(int types = ALL_TYPS,
                       int severity = ALL_SEVS,
                       string text = "",
                       logic issued = 1'bx);
\end{verbatim}

Creates a watchpoint descriptor that will be triggered when the specified message is used. The message can be specified by type, severity or by text pattern. By default, messages of all types, severities and text are specified. A message must match all specified criteria to trigger the watchpoint. The \texttt{issued} parameter specifies if the watchpoint is triggered when the message is physically issued (1'b1), physically not issued, i.e., filtered out (1'b0) or regardless if the message is physically issued or not (1'bx).

A watchpoint will be repeatedly triggered every time a message matching the watchpoint specification is issued by a message service interface associated with the watchpoint.
virtual function void

    add_watchpoint(int watchpoint_id,
                    string name = "",
                    string instance = "",
                    bit recurse = 0);

virtual function void

    remove_watchpoint(int watchpoint_id,
                       string name = "",
                       string instance = "",
                       bit recurse = 0);

Adds or removes the specified watchpoint to or from the specified message service interfaces. If a message matching the watchpoint specification is issued by one of the specified message service interfaces associated with the watchpoint, the watchpoint will be triggered.

virtual task wait_for_watchpoint(int watchpoint_id,
                                 ref vmm_log_msg msg);

Waits for the specified watchpoint to be triggered by a message issued by one of the message service interfaces attached to the watchpoint. A descriptor of the message that triggered the watchpoint will be returned.

virtual task wait_for_msg(string name = ",
                           string instance = ",
                           bit recurse = 0,
                           int typs = ALL_TYPS,
                           int severity = ALL_SEVS,
                           string text = ",
                           logic issued = 1'bx,
                           ref vmm_log_msg msg);

Sets up and waits for a one-time watchpoint for the specified message on the specified message service interface. The watchpoint is triggered only once and removed after being triggered.

virtual task report(string name = "./",
                    string instance = "./",
                    bit recurse = 0);

Reports a failure if any of the specified message service interfaces have issued any error or fatal messages. Reports a success otherwise. The text of the pass or fail message is specified using the vmm_log_format::pass_or_fail() method.
virtual function void prepend_callback(vmm_log_callbacks cb);
virtual function void append_callback(vmm_log_callbacks cb);

Globally prepends or appends the specified callback façade instance with the message service. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

virtual function void unregister_callback(vmm_log_callbacks cb);

Globally unregisters the specified callback façade instance with the message service. A warning is issued if the specified façade instance is not currently registered with the service. Callback façade instances can later be re-registered.

vmm_log_msg
This class describes a message issued by a message service interface that caused a watchpoint to be triggered. It is returned by the vmm_log::wait_for_watchpoint() and vmm_log::wait_for_msg() method.

vmm_log log;
A reference to the message reporting interface that has issued the message.

time timestamp;
The simulation time when the message was issued.

int original_typ;
Original message type as specified in the code creating the message.

int original_severity;
Original message severity as specified in the code creating the message.

int effective_typ;
Effective message type as potentially modified by the vmm_log::modify() method.
int effective_severity;
Effective message severity as potentially modified by the vmm_log::modify() method.

string text[$];
Formatted text of the message. Each element of the array contains one line of text as built by individual calls to the vmm_log::text() method.

bit issued;
Indicates if the message has been physically issued or not. If non-zero, then the message has been issued.

int handling;
The simulation handling after the message is physically issued.

vmm_log_format
This class is used to specify how messages are formatted before being displayed or logged to files. The default implementation of these methods produces the default message format.

virtual function string format_msg( string name,
        string instance,
        string msg_typ,
        string severity,
        string lines[$]);

This method is called by all message service interfaces to format a message on the first occurrence of a call to the vmm_log::end_msg() method or empty vmm_log::text(""") method call. Subsequent calls to the vmm_log::end_msg() method or empty vmm_log::text(""") method use the vmm_log_format::continue_msg() method.

The lines parameter contains one line of message text for each non-empty call to the vmm_log::text() method.
virtual function string continue_msg( string name,
        string instance,
        string msg_typ,
        string severity,
        string lines[]);

This method is called by all message service interfaces to format a message on the first occurrence of a call to the \texttt{vmm\_log::end\_msg()} method or empty \texttt{vmm\_log::text("")} method call. Subsequent calls to the \texttt{vmm\_log::end\_msg()} method or empty \texttt{vmm\_log::text("")} method use the \texttt{vmm\_log\_format::continue\_msg()} method.

The \texttt{lines} parameter contains one line of message text for each non-empty call to the \texttt{vmm\_log::text()} method since the last empty call to the \texttt{vmm\_log::text("")} method. It does not contain lines that were previously formatted in a prior call to this method or the \texttt{vmm\_log\_format::format\_msg()} method.

virtual function string abort_on_error( int count,
        int limit);

This method is called when the total number of COUNT\_ERROR messages exceed the error message threshold. The string returned by the method describes the cause of the simulation aborting. If \texttt{null} is returned, no explanation is displayed.

This method is called and the returned string is displayed before the \texttt{vmm\_log\_callbacks::pre\_abort()} callback methods are invoked.

virtual function string pass_or_fail( bit pass,
        string name,
        string instance,
        int fatals,
        int errors,
        int warnings,
        int dem\_errs,
        int dem\_warns);

This method is called by the \texttt{vmm\_log::report()} method to format the final pass/fail message at the end of simulation. The \texttt{pass} argument, if true, indicates that the simulation was successful. The \texttt{name} and \texttt{instance} arguments are the specified name and instance names specified to the \texttt{vmm\_log::report()} method. The \texttt{fatsals} argument is the total number of \texttt{vmm\_log::FATAL\_SEV} messages that were issued. The \texttt{errors} argument is the total number of \texttt{vmm\_log::ERROR\_SEV} messages that were issued. The \texttt{warnings} argument is
the total number of \texttt{vmm\_log::WARNING\_SEV} messages that were issued. The \texttt{dem\_errs} argument is the total number of \texttt{vmm\_log::ERROR\_SEV} messages that were demoted. The \texttt{dem\_warns} argument is the total number of \texttt{vmm\_log::WARNING\_SEV} messages that were demoted.

\textbf{vmm\_log\_callbacks}

This class provides a façade for the callback methods provided by the message service. Callbacks are associated with the message service itself, not a particular message service interface instance.

\textbf{virtual task pre\_abort(vmm\_log log);}

This callback method is invoked by the message service before the simulation is aborted because of an \texttt{ABORT} simulation handling or exceeded maximum number of \texttt{COUNT\_ERROR} messages. The message service instance provided as argument can be used to issue further messages.

\textbf{virtual task pre\_stop(vmm\_log log);}

This callback method is invoked by the message service before the simulation is stopped because of a \texttt{STOP} simulation handling. The message service instance provided as argument can be used to issue further messages.

\textbf{virtual task pre\_debug(vmm\_log log);}

This callback method is invoked by the message service before the breaking into the debugger because of a \texttt{DEBUGGER} simulation handling. The message service instance provided as argument can be used to issue further messages.

\textbf{Example A-2. Issuing a Simple Message}

\begin{verbatim}
program test;
    vmm\_log log = new("Test", "Singleton");

    initial begin
        if (log.start\_msg(vmm\_log::DEBUG\_TYP)) begin
            void'(log.text("Starting test"));
            log.end\_msg();
        end
    ...
end
endprogram
\end{verbatim}
Example A-3. Issuing a Simple Message using a Macro

```verilog
program test;
  vmm_log log = new("Test", "Singleton");

initial begin
  `vmm_debug(log, "Starting test");
  ...
end
endprogram
```

Example A-4. Issuing a Complex Message

```verilog
program test;
  vmm_log log = new("Test", "Singleton");

initial begin
  ...
  while (log.start_msg(vmm_log::FAILURE_TYP, vmm_log::WARNING_SEV)) begin
    string str;
    if (!log.text(...)) break;
    if (!log.text(transaction.psdisplay())) break;
    $sformat(str, ...);
    if (!log.text(str)) break;
    log.end_msg();
    break;
  end
  ...
end
endprogram
```

Example A-5. Pattern-Based Message Promotion

Demote all messages with an `ERROR_SEV` severity containing the pattern “abort” in all instances of message service interfaces named “AMBA AHB Interface Master” to a `WARNING_SEV` severity.

```verilog
program test
  verif_env env = new;

initial begin
  env.build();
  env.log.modify("AMBA AHB Interface Master", "/./", "/abort/", ", vmm_log::ERROR_SEV, ", vmm_log::WARNING_SEV);
  ...
end
endprogram
```
VMM_DATA

This base class is to be used as the basis for all transaction descriptors and data models. It provides a standard set of methods expected to be found in all descriptors. It also creates a common class—akin to C’s void type—that can be used to create generic components. The guidelines covering the development of data and transaction descriptors based on this class can be found in section titled "Data and Transactions" on page 140.

**function new(vmm_log log);**

Creates a new instance of this data model or transaction descriptor with the specified message service interface. The specified message service interface is used when constructing the `vmm_data::notify` property.

Because of the potentially large number of instances of data objects, a `class-static` message service interface should be used to minimize memory usage and to be able to control class-generic messages:

```plaintext
class eth_frame extends vmm_data {
    static vmm_log log = new("eth_frame", "class");
    function new()
        super.new(this.log);
    ...
    endfunction
endclass: eth_frame
```

**function vmm_log set_log(vmm_log log);**

Replaces the message service interface for this instance of a data model or transaction descriptor with the specified message service interface and returns a reference to the previous message service interface. Can be used to associate a descriptor with the message service interface of a transactor currently processing the transaction or to set the service when it was not available during initial construction.

```plaintext
int stream_id;
int scenario_id;
int data_id;
```

Unique identifiers for a data model or transaction descriptor instance. They specify the offset of the descriptor within a sequence and the sequence offset within a stream. These properties must be set by the transactor that instantiates the descriptor. They are set by the predefined generator before randomization so they can be used to specify conditional constraints to express instance-specific or stream-specific constraints.
vmm_notify notify;  
enum { EXECUTE;  
      STARTED;  
      ENDED}; 

A notification service interface with three pre-configured events. The EXECUTE notification is ON/OFF and indicated by default. It can be used to prevent the execution of a transaction or the transfer of data if reset. The STARTED and ENDED notifications are ON/OFF events and indicated by the transactor at the start and end of the transaction execution or data transfer. The meaning and timing of the notifications is specific to the transactor executing the transaction described by this instance.

function void display(string prefix = "");  
Displays the current value of the transaction or data described by this instance in a human-readable format on the standard output. Each line of the output will be prefixed with the specified prefix. This method prints the value returned by the psdisplay() method.

virtual function string psdisplay(string prefix = "");  
Returns an image of the current value of the transaction or data described by this instance in a human-readable format as a string. The string may contain newline characters to split the image across multiple lines. Each line of the output must be prefixed with the specified prefix.

virtual function bit is_valid(bit silent = 1,  
                          int kind = -1);  
Checks if the current value of the transaction or data described by this instance is valid and error-free, according to the optionally specified kind or format. Returns TRUE (i.e., non-zero) if the content of the object is valid. Returns FALSE otherwise. The meaning (and use) of the kind argument is descriptor-specific and defined by the user-extension of this method.

If silent is TRUE (i.e., non-zero), no error or warning messages are issued if the content is invalid. If silent is FALSE, warning or error messages may be issued if the content is invalid.

virtual function vmm_data allocate();  
Allocates a new instance of the same type as the object instance. Returns a reference to the new instance. Useful to implement class factories to create instances of user-defined derived class in generic code written using the base class type.
virtual function vmm_data copy(vmm_data to = null);
Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated. Returns a reference to the target instance.

Note that the following trivial implementation will not work. Constructor copying is a shallow copy. The objects instantiated in the object (such as those referenced by the log and notify properties) are not copied and both copies will share references to the same service interfaces. Furthermore, it will not properly handle the case when the to argument is not null.

Example A-6. Invalid Implementation of the vmm_data::copy() Method

```
function vmm_data atm_cell::copy(vmm_data to = null)
    copy = new this;
endfunction
```

The following implementation is usually preferable:

Example A-7. Proper Implementation of the vmm_data::copy() Method

```
function vmm_data atm_cell::copy(vmm_data to = null)
    atm_cell cpy;

    if (to != null) begin
        if (!$cast_assign(cpy, to)) begin
            'vmm_fatal(log, "Not a atm_cell instance");
            return;
        end
    end
    else cpy = new;

    this.copy_data(cpy);
    cpy.vpi = this.vpi;
    ...;
    copy = cpy;
endfunction: copy
```

The base-class implementation of this method must not be called as it contains error detection code of a derived class that forgot to supply an implementation. The vmm_data::copy_data() method should be called instead.

virtual protected function void copy_data(vmm_data to);
Copies the current value of all base-class data properties in the current data object into the specified data object instance. This method should be called by the
implementation of the `vmm_data::copy()` method in classes immediately derived from this base class.

```cpp
virtual function bit compare( input vmm_data to,
              output string diff,
              input int kind = -1);
```

Compares the current value of the object instance with the current value of the specified object instance, according to the specified kind. Returns TRUE (i.e., non-zero) if the value is identical. If the value is different, FALSE is returned and a descriptive text of the first difference found is returned in the specified `string` variable. The `kind` argument may be used to implement different comparison functions (e.g., full compare, comparison of `rand` properties only, comparison of all properties physically implemented in a protocol and so on.)

```cpp
virtual function int unsigned byte_pack(
              ref logic [7:0] bytes[],
              int unsigned offset = 0,
              int kind = -1);
```

Packs the content of the transaction or data into the specified dynamic array of bytes, starting at the specified offset in the array. The array is resized appropriately. Returns the number of bytes added to the array.

If the data can be interpreted or packed in different ways, the `kind` argument can be used to specify which interpretation or packing to use.

```cpp
virtual function int unsigned byte_unpack(
              const ref logic [7:0] bytes[],
              input int unsigned offset = 0,
              input int len = -1,
              input int kind = -1);
```

Unpacks the specified number of bytes of data from the specified offset in the specified dynamic array into this descriptor. If the number of bytes to unpack is specified as -1, the maximum number of bytes will be unpacked. Returns the number of bytes unpacked. If there is not enough data in the dynamic array to completely fill the descriptor, the remaining properties are set to unknown and a warning may be issued.

If the data can be interpreted or unpacked in different ways, the `kind` argument can be used to specify which interpretation or packing to use.
virtual function int unsigned byte_size(int kind = -1);
Returns the number of bytes required to pack the content of this descriptor. This method will be more efficient than \texttt{vmm\_data::byte\_pack()} for simply knowing how many bytes are required by the descriptor because no packing is actually done.

If the data can be interpreted or packed in different ways, the \textit{kind} argument can be used to specify which interpretation or packing to use.

virtual function int unsigned max_byte_size(
      int kind = -1);
Returns the maximum number of bytes that will ever be required to pack the content of any instance of this descriptor. A value of 0 indicates an unknown maximum size. Can be used to allocate memory buffers in the DUT or verification environment of suitable sizes.

If the data can be interpreted or packed in different ways, the \textit{kind} argument can be used to specify which interpretation or packing to use.

virtual function void save(int file);
Appends the content of this descriptor to the specified file. The format is user defined and may be binary. By default, simply packs the descriptor and saves the value of the bytes, in sequence, as binary values and terminated by a newline.

virtual function bit load(int file);
Sets the content of this descriptor from the data in the specified file. The format is user defined and may be binary. By default, interprets a complete line as binary byte data and unpacks it.

Should return FALSE (i.e., zero) if the loading operation was not successful.

\textbf{VMM\_CHANNEL}

This class implements a generic transaction-level interface mechanism. The guidelines covering the usage of the channel can be found in section titled "Transaction-Level Interfaces" on page 171.
Offset values, either accepted as arguments or returned values, are always interpreted the same way. A value of 0 indicates the head of the channel (first transaction descriptor added). A value of –1 indicates the tail of the channel (last transaction descriptor added). Positive offsets are interpreted from the head of the channel. Negative offsets are interpreted from the tail of the channel. For example, an offset value of –2 indicates the transaction descriptor just before the last transaction descriptor in the channel. It is illegal to specify a non-zero offset that does not correspond to a transaction descriptor already in the channel.

The channel includes an active slot that can be used to create more complex transactor interfaces. The active slot counts toward the number of transaction descriptors currently in the channel for control-flow purposes but cannot be accessed nor specified via an offset specification.

The implementation uses a macro to define a class named “<class_name>_chan” derived from the class named “vmm_channel” for any user-specified class named “class_name”.

`vmm_channel(class_name);
Defines a channel class to transport instances of the specified class. The transported class must be derived from the vmm_data class. This macro is typically invoked in the same file where the specified class is defined and implemented.

This macro creates an external class declaration and no implementation. It is typically invoked when the channel class must be visible to the compiler but the actual channel class declaration is not yet available.

function new( string name,
    string instance,
    int unsigned full = 1,
    int unsigned empty = 0,
    bit fill_as_bytes = 0);
Creates a new instance of a channel with the specified name, instance name and full and empty levels. If the fill_as_bytes argument is TRUE (i.e., non-zero) the full and empty levels and the fill level of the channel are interpreted as the number of bytes in the channel as computed by the sum of vmm_data::byte_size() of all transaction descriptors in the channel, not the number of objects in the channel. If the value is FALSE (i.e., zero), the full and empty levels and the fill level of the channel are interpreted as the number of transaction descriptors in the channel. It is illegal to configure a channel with a full level lower than the empty level.
vmm_channel

vmm_log log;
Message service interface for messages issued from within the channel instance.

function void reconfigure( int full = -1,
                           int empty = -1,
                           logic fill_as_bytes = 1’bx);
If not negative, reconfigure the full or empty levels of the channel to the specified levels. Reconfiguration may cause threads currently blocked on a vmm_channel::put() call to unblock. If the fill_as_bytes argument is specified as 1’b1 or 1’b0, the interpretation of the fill level of the channel is modified accordingly. Any other value leaves the interpretation of the fill level unchanged.

function int unsigned full_level();
function int unsigned empty_level();
Returns the currently configured full or empty level.

function int unsigned level();
Returns the current fill level of the channel. The interpretation of the fill level depends on the configuration of the channel instance.

function int unsigned size();
Returns the number of transaction descriptors currently in the channel, including the active slot, regardless of the interpretation of the fill level.

function bit is_full();
Returns TRUE (i.e., non-zero) if the fill level is greater than or equal to the currently configured full level. Returns FALSE otherwise.

vmm_notify notify
An event notification interface used to indicate the occurrence of significant events within the channel. The notifications shown in Table A-3 are pre-configured.
### Table A-3. Pre-Configured Notifications in `vmm_channel` Notifier Interface

<table>
<thead>
<tr>
<th>Symbolic Property</th>
<th>Corresponding Significant Event</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmm_channel::FULL</code></td>
<td>Channel has reached or surpassed its configured full level. This notification is configured ON/OFF. No status is returned.</td>
</tr>
<tr>
<td><code>vmm_channel::EMPTY</code></td>
<td>Channel has reached or underflowed the configured empty level. This event is configured ON/OFF. No status is returned.</td>
</tr>
<tr>
<td><code>vmm_channel::PUT</code></td>
<td>A new transaction descriptor has been added to the channel. This event is configured ONE_SHOT. The newly added transaction descriptor is available as status.</td>
</tr>
<tr>
<td><code>vmm_channel::GOT</code></td>
<td>A transaction descriptor has been removed from the channel. This event is configured ONE_SHOT. The newly removed transaction descriptor is available as status.</td>
</tr>
<tr>
<td><code>vmm_channel::PEEKED</code></td>
<td>A transaction descriptor has been peeked from the channel. This event is configured ONE_SHOT. The newly peeked transaction descriptor is available as status.</td>
</tr>
<tr>
<td><code>vmm_channel::ACTIVATE</code></td>
<td>A transaction descriptor has been transferred to the active slot. This notification also implies a <code>PEEKED</code> notification. This event is configured ONE_SHOT. The newly activated transaction descriptor is available as status.</td>
</tr>
<tr>
<td><code>vmm_channel::ACT_STARTED</code></td>
<td>The state of a transaction descriptor in the active slot has been updated to <code>STARTED</code>. This event is triggered ONE_SHOT. The currently active transaction descriptor is available as status.</td>
</tr>
<tr>
<td><code>vmm_channel::ACT_COMPLETED</code></td>
<td>The state of a transaction descriptor in the active slot has been updated to <code>COMPLETED</code>. This event is configured ONE_SHOT. The currently active transaction descriptor is available as status.</td>
</tr>
</tbody>
</table>
function void flush();
Flushes the content of the channel. Flushing will unblock any thread currently blocked in the `vmm_channel::put()` method. This method will cause the `FULL` notification to be reset or the `EMPTY` notification to be indicated. Flushing a channel unlocks all sources and consumers.

function void sink();
Flushes the content of the channel and sinks any further objects put into it. No transaction descriptors will accumulate in the channel while it is sunk. Any thread attempting to obtain a transaction descriptor from the channel will be blocked until the flow through the channel is restored using the `vmm_channel::flow()` method. This method will cause the `FULL` notification to be reset or the `EMPTY` notification to be indicated.

function void flow();
Restores the normal flow of transaction descriptors through the channel.

function void lock(bit [1:0] who);
function void unlock(bit [1:0] who);
Blocks any source (consumer) as if the channel was full (empty) until explicitly unlocked. The side that is to be locked or unlocked is specified using the sum of the symbolic values shown in Table A-4.
Table A-4. Channel Endpoint Identifiers

<table>
<thead>
<tr>
<th>Symbolic Property</th>
<th>Channel Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmm_channel::SOURCE</td>
<td>The producer side, i.e., any thread calling the vmm_channel::put() method</td>
</tr>
<tr>
<td>vmm_channel::SINK</td>
<td>The consumer side, i.e., any thread calling the vmm_channel::get() method</td>
</tr>
</tbody>
</table>

Locking a source does not indicate the `FULL` notification, nor does locking the sink indicate the `EMPTY` notification—although they have the same control-flow effect.

```plaintext
function bit is_locked(bit [1:0] who);
Returns TRUE (i.e., non-zero) if any of the specified sides is locked. If both sides are specified, returns TRUE if any side is locked.
```

Example A-8. Querying the Lock Status of a Channel

```
while (chan.is_locked(vmm_channel::SOURCE +
                      vmm_channel::SINK)) begin
  chan.notify.wait_for(vmm_channel::UNLOCKED);
end
```

```plaintext
task put( class_name obj, 
         int offset = -1);
Puts the specified transaction descriptor in the channel at the specified offset. If the fill level of the channel, including the active slot, is greater than or equal to the configured full level, or if the source is locked, the task will block until the fill level of the channel is less than or equal to the configured empty level and the source is unlocked.

It is an error to specify an offset that does not already exist in the channel.

This method may cause the `FULL` notification to be indicated and will cause the `EMPTY` notification to be reset.
```

```plaintext
function void sneak(class_name obj, 
                     int offset = -1)
Puts the specified transaction descriptor in the channel at the specified offset. This task will never block, regardless of the configured full level. Use only when a guaranteed non-blocking version of vmm_channel::put() is required—for
```
example, inside a function—and threads using this method have some other means of eventually blocking their execution.

It is an error to specify an offset that does not already exist in the channel or sneak a new transaction descriptor into a locked channel.

This method may cause the FULL notification to be indicated and will cause the EMPTY notification to be reset.

```verilog
function class_name unput(int offset = -1);
Removes the specified transaction descriptor from the channel. It is an error to specify an offset to a transaction descriptor that does not exist.

This method may cause the EMPTY notification to be indicated and will cause the FULL notification to be reset.
```

```verilog
task get(output class_name obj,
    input int offset = 0);
Retrieves the next transaction descriptor in the channel at the specified offset. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved. This method may cause the EMPTY notification to be indicated or the FULL notification to be reset.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or with a non-empty active slot.
```

```verilog
task peek(output class_name obj,
    input int offset = 0);
Gets a reference to the next transaction descriptor that will be retrieved from the channel at the specified offset without actually retrieving it. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or with a non-empty active slot.
```

```verilog
task activate(output class_name obj,
    input int offset = 0);
If the active slot is not empty, first removes the transaction descriptor currently in the active slot.
```
Move the transaction descriptor at the specified offset in the channel to the active slot and update the status of the active slot to `vmm_channel::PENDING`. If the channel is empty, this method will wait until a transaction descriptor becomes available. The transaction descriptor is still considered as being in the channel.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or to use this method with multiple concurrent consumer threads.

```plaintext
function class_name active_slot();
Returns the transaction descriptor currently in the active slot. Returns `null` if the active slot is empty.

function class_name start();
Updates the status of the active slot to `vmm_channel::STARTED`. The transaction descriptor remains in the active slot. It is an error to call this method if the active slot is empty. The `vmm_data::STARTED` notification of the transaction descriptor in the active slot is indicated.

function class_name complete(vmm_data status = null);
Updates the status of the active slot to `vmm_channel::COMPLETED`. The transaction descriptor remains in the active slot and may be restarted. It is an error to call this method if the active slot is empty. The `vmm_data::ENDED` notification of the transaction descriptor in the active slot is indicated with the optionally specified completion status descriptor.

function class_name remove();
Updates the status of the active slot to `vmm_channel::INACTIVE` and removes the transaction descriptor from the active slot from the channel. This method may cause the `EMPTY` notification to be indicated or the `FULL` notification to be reset. It an error to call this method with an active slot in the `vmm_channel::STARTED` state. The `vmm_data::ENDED` notification of the transaction descriptor in the active slot is indicated.

function active_status_e status();
Returns one of the enumerated values in Table A-5, indicating the status of the transaction descriptor in the active slot.
### Table A-5. Pre-Configured Notifications in `vmm_channel` Notifier Interface

<table>
<thead>
<tr>
<th>Symbolic Property</th>
<th>Corresponding Significant Event</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmm_channel::INACTIVE</code></td>
<td>No transaction descriptor is present in the active slot.</td>
</tr>
<tr>
<td><code>vmm_channel::PENDING</code></td>
<td>A transaction descriptor is present in the active slot but it has not been started yet.</td>
</tr>
<tr>
<td><code>vmm_channel::STARTED</code></td>
<td>A transaction descriptor is present in the active slot and it has been started, but it is not completed yet. The transaction is being processed by the downstream transactor</td>
</tr>
<tr>
<td><code>vmm_channel::COMPLETED</code></td>
<td>A transaction descriptor is present in the active slot and it has been processed by the downstream transactor, but it has not yet been removed from the active slot.</td>
</tr>
</tbody>
</table>

**task tee(output class_name obj);**

When the tee mode is ON, retrieve a copy of the transaction descriptor references that have been retrieved by the `get()` or `activate()` methods. The task will block until one of the `get()` or `activate()` methods successfully completes.

This method can be used to fork off a second stream of references to the transaction descriptor stream. Note that the transaction descriptors themselves are not copied. The references returned by this method are referring to the same transaction descriptor instances obtained by the `get()` and `activate()` methods.

**function bit tee_mode(bit is_on);**

Turn the tee mode ON or OFF for this channel. Returns TRUE if the tee mode was previously ON. A threads blocked on a call to the `vmm_channel::tee()` method will not unblock execution if the tee mode is turned OFF. If the stream of references is not drained via the `vmm_channel::tee()` method, data will accumulate in the secondary channel when the tee mode is ON.

**function void connect(vmm_channel downstream);**

Connect the output of this channel instance to the input of the specified channel instance. The connection is performed with a blocking model (see section titled "In-Order Atomic Execution Model" on page 177) to communicate the status of the downstream channel to the producer interface of the upstream channel. Flushing this
Channel will cause the downstream connected channel to be flushed as well. However, flushing the downstream channel will not flush this channel.

The effective full and empty levels of the combined channels is equal to the sum of their respective levels minus one. However, the detailed blocking behavior of the various interface methods will differ from using a single channel with an equivalent configuration. Additional zero-delay simulation cycles may be required while transaction descriptors are transferred from the upstream channel to the downstream channel.

Connected channels need not be of the same type but must carry compatible polymorphic data.

The connection of a channel into another one can be dynamically modified and broken by connection to a null reference. However, modifying the connection while there is data flowing through the channels may yield unpredictable behavior.

```
function class_name for_each(bit reset = 0);
```

Iterates over all of the transaction descriptors currently in the channel. The content of the active slot, if non-empty, is not included in the iteration. If the reset argument is TRUE, a reference to the first transaction descriptor in the channel is returned. Otherwise, a reference to the next transaction descriptor in the channel is returned. Returns null when the last transaction descriptor in the channel has been returned. It will keep returning null unless reset.

Modifying the content of the channel in the middle of an iteration will yield unexpected results.

```
function int unsigned for_each_offset();
```

Returns the offset of the last transaction descriptor returned by the `vmm_channel::for_each()` method. An offset of 0 indicates the first transaction descriptor in the channel.

```
function bit record(string filename);
```

Starts recording the flow of transaction descriptors added through the channel instance in the specified file. The `vmm_data::save()` method must be implemented for that transaction descriptor and defines the file format. A transaction descriptor is recorded when added to the channel by the `vmm_channel::put()` method.
vmm_broadcast

A null filename stops the recording process. Returns TRUE if the specified file was successfully opened.

**task bit playback(output bit success,**
  **input string filename,**
  **input vmm_data loader,**
  **input bit metered = 0);**

Locks all sources of the current channel and playback the transaction descriptors found in the specified file. The `vmm_data::load()` load method in the object specified by the `loader` argument is used and defines the file format. The transaction descriptors are added one by one in the order specified in the file. If the `metered` argument is TRUE, the transaction descriptors are added to the channel with the same relative time interval as they were originally put in when the file was recorded.

All consumers are locked out from the channel during playback. Normal operation resumes after the data has been entirely played back. Returns TRUE if the playback was successful.

**VMM_BROADCAST**

Channels are point-to-point data transfer mechanisms. If multiple consumers are extracting transaction descriptors from a channel, the transaction descriptors are distributed among the various consumers and each of the N consumers sees 1/N descriptors. If a point-to-multi-point mechanism is required, where all consumers must see all of the transaction descriptors in the stream, a `vmm_broadcast` component can be used to replicate the stream of transaction descriptors from a source channel to an arbitrary and dynamic number of output channels. If only two output channels are required, the `vmm_channel::tee()` method of the source channel may also be used.

Individual output channels can be configured to receive a copy of the reference to the source transaction descriptor (most efficient but the same descriptor instance is shared by the source and all like-configured output channels) or to use a new descriptor instance copied from the source object (least efficient but uses a separate instance that can be modified without affecting other channels or the original descriptor). A `vmm_broadcast` component can be configured to use references or copies in output channels by default.
In the As Fast As Possible (AFAP) mode, the full level of the output channels is ignored. Only the full level of the source channel will control the flow of data through the broadcaster. Output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next descriptor is removed from the source channel (if available) and added to all output channels, even if they are already full.

In the As Late As Possible (ALAP) mode, the slowest of the output or input channels controls the flow of data through the broadcaster. Only once all active output channels are empty, the next descriptor is removed from the source channel (if available) and added to all output channels.

If there are no active output channels, the input channel is continuously drained as transaction descriptors are added to it to avoid data accumulation.

This class is based on the vmm_xactor class.

VMM Standard Library Specification

vmm_log log;
Message service interface for this broadcaster. Set by the constructor and uses the name and instance name specified in the constructor.

function new( string name,
             string instance,
             vmm_channel source,
             bit use References = 1,
             bcast_mode_typ mode = AFAP);
Creates a new instance of a channel broadcaster object with the specified name, instance name, source channel and broadcasting mode. If use_references is TRUE (i.e., non-zero), references to the original source transaction descriptors are assigned to output channels by default (unless individual output channels are configured otherwise).

See the documentation for the broadcast_mode() method on page 399 for a description of the available modes.

virtual function void start_xactor();
Starts this vmm_broadcast instance. The broadcaster can be stopped. Any extension of this method must call super.start_xactor().
virtual function void stop_xactor();
Suspends this `vmm_broadcast` instance. The broadcaster can be restarted. Any extension of this method must call `super.stop_xactor()`.

virtual function void
    reset_xactor(reset_e rst_type = SOFT_RST);
Resets this `vmm_broadcast` instance. The broadcaster can be restarted. The input channel and all output channels are flushed.

virtual function void
    broadcast_mode(bcast_mode_e mode);
Changes the broadcasting mode to the specified mode. The new mode takes effect immediately. The available modes are specified by using one of the class-level enumerated symbolic values shown in Table A-6.

**Table A-6. Broadcasting Mode Enumerated Values**

<table>
<thead>
<tr>
<th>Enumerated Value</th>
<th>Broadcasting Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmm_broadcast::ALAP</code></td>
<td><em>As Late As Possible.</em> Data is broadcast only when all active output channels are empty. This delay ensures that data is not broadcast any faster than the slowest of all consumers can digest it.</td>
</tr>
<tr>
<td><code>vmm_broadcast::AFAP</code></td>
<td><em>As Fast As Possible.</em> Active output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next descriptor from the input channel (if available) is immediately broadcast to all active output channels, regardless of their fill level. This mode <em>must not</em> be used if the data source can produce data at a higher rate than the slowest data consumer and if broadcast data in all output channels are not consumed at the same average rate.</td>
</tr>
</tbody>
</table>
virtual function int
    new_output(vmm_channel channel,
               logic use_references = 1'bx);

Adds the specified channel instance as a new output channel to the broadcaster. If use_references is TRUE (i.e., non-zero), references to the original source transaction descriptor is added to the output channel. If FALSE (i.e., zero), a new instance copied from the original source descriptor is added to the output channel. If unknown (i.e., 1'bx), the default broadcaster configuration is used.

If there are no output channels, the data from the input channel is continuously drained to avoid data accumulation.

This method returns a unique identifier for the output channel that must be used to modify the configuration of the output channel.

Any user extension of this method must call super.new_output().

virtual function void bcast_on(int unsigned output_id);
virtual function void bcast_off(int unsigned output_id);

Turns broadcasting to the specified output channel on or off. By default, broadcasting to an output channel is on. When broadcasting is turned off, the output channel is flushed and the addition of new transaction descriptors from the source channel is inhibited. The addition of descriptors from the source channel is resumed as soon as broadcasting is turned on.

If all output channels are off, the input channel is continuously drained to avoid data accumulation.

Any user extension of these methods should call super.bcast_on() or super.bcast_off(), respectively.

virtual protected function bit
    add_to_output( int unsigned decision_id,
                   int unsigned output_id,
                   vmm_channel channel,
                   vmm_data obj);

Overloading this method allows the creation of broadcaster components with different broadcasting rules. If this function returns TRUE (i.e., non-zero), the transaction descriptor will be added to the specified output channel. If this function returns FALSE (i.e., zero), the descriptor is not added to the channel. If the output
channel is configured to use new descriptor instances, the \textit{dat} parameter is a reference to that new instance.

This method is not necessarily invoked in increasing order of output identifiers. It is only called for output channels currently configured as ON. If this method returns \texttt{FALSE} for all output channels for a given broadcasting cycle, lock-up may occur. The \texttt{decision_id} argument is reset to \texttt{0} at the start of every broadcasting cycle and is incremented after each call to this method in the same cycle. It can be used to identify the start of broadcasting cycles.

If transaction descriptors are manually added to output channels, it is important that the \texttt{vmm_channel::sneak()} method be used to prevent the execution thread from blocking. It is also important that \texttt{FALSE} be returned to prevent that descriptor from being added to that output channel by the default broadcast operations and thus from being duplicated into the output channel.

The default implementation of this method always returns \texttt{TRUE}.

\textbf{VMM_SCHEDULER}

Channels are point-to-point transaction descriptor transfer mechanisms. If multiple sources are adding descriptors to a single channel, the descriptors are interleaved with the descriptors from the other sources in a fair but uncontrollable way. If a multi-point-to-point mechanism is required to follow a specific scheduling algorithm, a \texttt{vmm_scheduler} component can be used to identify which source stream should next be forwarded to the output stream.

This class is based on the \texttt{vmm_xactor} class.

\texttt{vmm_log log;}
Message service interface for this scheduler. Set by the constructor and uses the name and instance name specified in the constructor.

\texttt{protected vmm_channel out_chan;}
Reference to the output channel. Set by the constructor.

\texttt{function new( string name,}
\texttt{    string instance,}
vmm_channel destination,
int instance_id = -1);

Creates a new instance of a channel scheduler object with the specified name, instance name, destination channel and optional instance identifier.

**virtual function void start_xactor();**

Starts this `vmm_scheduler` instance. The scheduler can be stopped. Any extension of this method must call `super.start_xactor()`.

**virtual function void stop_xactor();**

Suspends this `vmm_scheduler` instance. The scheduler can be restarted. Any extension of this method must call `super.stop_xactor()`.

**virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);**

Resets this `vmm_scheduler` instance. The output channel and all input channels are flushed. If a HARD_RST reset type is specified, the scheduler election factory instance in the randomized_sched property is replaced with a new default instance.

**virtual function int new_source(vmm_channel chan);**

Adds the specified channel instance as a new input channel to the scheduler. This method returns an identifier for the input channel that must be used to modify the configuration of the input channel or -1 if an error occurred.

Any user extension of this method must call `super.new_source()`.

**virtual function void sched_on(int unsigned input_id);**

**virtual function void sched_off(int unsigned input_id);**

Turns scheduling from the specified input channel on or off. By default, scheduling from an input channel is on. When scheduling is turned off, the input channel is not flushed and the scheduling of new transaction descriptors from that source channel is inhibited. The scheduling of descriptors from that source channel is resumed as soon as scheduling is turned on.

Any user extension of this method should call `super.sched_from_input()` or `super.sched_from_input()`, respectively.
virtual protected task 
    schedule(output vmm_data obj, 
               input vmm_channel sources[$], 
               int unsigned input_ids[$]);

Overloading this method allows the creation of scheduling components with different rules. It is invoked for each scheduling cycle. The transaction descriptor returned by this method in the \textit{obj} argument is added to the output channel. If this method returns \texttt{null}, no descriptor is added for this scheduling cycle. The input channels provided in the \textit{sources} argument are all the currently non-empty ON input channels. Their corresponding input identifier is found in the \textit{input_ids} argument.

New scheduling cycles are attempted whenever the output channel is not full. If no transaction descriptor is scheduled from any of the currently non-empty source channels, the next scheduling cycle will be delayed until an additional ON source channel becomes non-empty. If there are no empty input channels and no OFF channels, lock-up will occur.

The default implementation of this method randomizes the instance found in the \textit{randomized_sched} property.

virtual protected task get_object(output vmm_data obj, 
                                  vmm_channel source, 
                                  int unsigned input_id, 
                                  int offset);

This method is invoked by the default implementation of the \texttt{vmm\_scheduler::schedule()} method to extract the next scheduled transaction descriptor from the specified input channel at the specified offset within the channel. Overloading this method allows access to or replacement of the object that is about to be scheduled. User-defined extensions can be used to introduce errors by modifying the object, interfere with the scheduling algorithm by substituting a different object or recording of the schedule into a functional coverage model.

Any object that is returned by this method via the \textit{obj} argument must either have been internally created or physically removed from the input source using the \texttt{vmm\_channel::get()} method. If a reference to the object remains in the input channel (e.g., by using the \texttt{vmm\_channel::peek()} or \texttt{vmm\_channel::activate()} method), it is liable to be scheduled more than once as the mere presence of an instance in any of the input channel makes it available to the scheduler.
vmm_scheduler_election randomized_sched;
Factory instance randomized by the default implementation of the vmm_scheduler::schedule() method. Can be replaced with user-defined extensions to modify the election rules.

vmm_scheduler_election
This class implements the random election rules for the next scheduling cycle. The election is performed by randomizing an instance of this class. The default implementation provides a round-robin election process.

int instance_id;
Instance identifier of the vmm_scheduler class instance that is randomizing this object instance. Can be used to specified instance-specific constraints.

int unsigned election_id;
Incremented by the vmm_scheduler instance that is randomizing this object instance before every election cycle. Can be used to specified election-specific constraints.

int unsigned n_sources;
Number of sources. Equal to vmm_scheduler_election::sources.size().

vmm_channel sources[$];
Input source channels with transaction descriptors available to be scheduled.

int unsigned ids[$];
Unique input identifiers corresponding to the source channels at the same index in the sources array.

int unsigned id_history[$];
A queue of the (up to) 10 last input identifiers that were elected.

vmm_data obj_history[$];
A list of the (up to) 10 last transaction descriptors that were elected.


**vmm_notify**

**rand int unsigned source_idx;**
Index in the `sources` array of the elected source channel. An index of -1 indicates no election. The `default_round_robin` constraint block constrains this property to be in the 0 to `sources.size() - 1` range.

**rand int unsigned obj_offset;**
Offset, within the source channel indicated by the `source_idx` property, of the elected transaction descriptor within the elected source channel. The `default_round_robin` constraint block constrains this property to be equal to 0.

**constraint default_round_robin;**
Constraints required by the default round-robin election process.

**function void post_randomize();**
The default implementation of this method helps performs the round-robin election.

**VMM_NOTIFY**

The `vmm_notify` class implements an interface to the notification service. The notification service provides a synchronization mechanism for concurrent threads or transactors. Unlike `event` variables, the operation of the notification is define at configuration time. Furthermore, notification can have status and timestamp information attached to their indication.

**function new(vmm_log log);**
Creates a new instance of this class, using the specified message service interface to issue error and debug messages.

**virtual function vmm_notify copy(vmm_notify to = null);**
Copies the current configuration of this notification service interface to the specified instance. If no instance is specified, a new one is allocated using the same message service interface as the original one. A reference to the target instance copied is returned.

Only the notification configuration information is copied and merged with any pre-configured notification in the destination instance. Copied notification configuration
will replace any pre-existing configuration for the same notification identifier. Status and timestamp information is **not** copied.

```cpp
virtual function int
    configure( int notification_id = -1,
                sync_e sync = ONE_SHOT);
```

Defines a new notification associated with the specified unique identifier. If a negative identifier value is specified, a new, unique identifier greater than 1,000,000 is returned. The thread synchronization mode of a notification is defined when the notification is configured, not when it is triggered or waited upon, using one of the `vmm_notify::ONE_SHOT`, `vmm_notify::BLAST`, or `vmm_notify::ON_OFF` synchronization types. This definition timing prevents a notification from being misused by the triggering or waiting threads.

### Table A-7. Notification Synchronization Mode Enumerated Values

<table>
<thead>
<tr>
<th>Enumerated Value</th>
<th>Broadcasting Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmm_notify::ONE_SHOT</code></td>
<td>Only threads currently waiting for the notification to be indicated are notified.</td>
</tr>
<tr>
<td><code>vmm_notify::BLAST</code></td>
<td>All threads waiting for the notification to be indicated in the same timestep at the indication are notified. This mode eliminates certain types of race conditions.</td>
</tr>
<tr>
<td><code>vmm_notify::ON_OFF</code></td>
<td>The notification is level-sensitive. Notifications remain notified until explicitly reset. Threads waiting for a notification that is still notified will not wait. This mode eliminates certain types of race conditions.</td>
</tr>
</tbody>
</table>

A warning may be issued if a notification is configured more than once.

Notification identifiers numbered from 1,000,000 and up are reserved for automatically generated notification identifiers. Predefined notification identifiers in the VMM base classes use identifiers 999,999 and down. User-defined notification identifiers can thus use values 0 and up.

```cpp
virtual function int is_configured(int notification_id);
```

Checks if the specified notification is currently configured. If this method returns 0, the notification is not configured. Otherwise, it returns an integer value corresponding
vmm_notify

to the current vmm_notify::ONE_SHOT, vmm_notify::ONE_BLAST or vmm_notify::ON_OFF configuration.

virtual function bit is_on(int notification_id);
Checks if the specified vmm_notify::ON_OFF notification is currently in the notify state. If this method returns TRUE, the notification is in the notify state and any call to the vmm_notify::wait_for() method will not block. A warning is issued if this method is called on any other types of notifications.

virtual task wait_for(int notification_id);
Suspends the execution thread until the specified notification is notified. It is an error to specify an unconfigured notification. Use the vmm_notify::status() function to retrieve any status descriptor attached to the indicated notification.

virtual task wait_for_off(int notification_id);
Suspends the execution thread until the specified vmm_notify::ON_OFF notification is reset. It is an error to specify an unconfigured or a non-ON/OFF notification. The status returned by subsequent calls to the vmm_notify::status() function is undefined.

virtual function bit is_waited_for(int notification_id);
Checks if a thread is currently waiting for the specified notification, including waiting for an ON/OFF notification to be reset. It is an error to specify an unconfigured notification. The function returns TRUE if there is a thread known to be waiting for the specified notification.

Note that the knowledge about the number of threads waiting for a particular notification is not definitive and may be out of date. As threads call the vmm_notify::wait_for() method, the fact that they are waiting for the notification is recorded. Once the notification is indicated and each thread returns from the method call, the fact that they are no longer waiting is also recorded. But if the threads are externally terminated via the disable statement or a timeout, the fact that they are no longer waiting cannot be recorded. In this case, it is up to the terminated threads to report that they are no longer waiting by calling the vmm_notify::terminated() method.

When a notification is reset with a hard reset, no threads are assumed to be waiting for any notification.
virtual function void terminated(int notification_id);
Indicates to the notification service interface that a thread waiting for the specified notification has been disabled and is no longer waiting.

virtual function vmm_data status(int notification_id);
Returns the status descriptor associated with the specified notification when it was last indicated. It is an error to specify an unconfigured notification.

virtual function time timestamp(int notification_id);
Returns the simulation time when the specified notification was last indicated. It is an error to specify an unconfigured notification.

virtual function void indicate( int notification_id,
   vmm_data status = null);
Indicates the specified notification with the optional status descriptor.

virtual function void
   set_notification( int notification_id,
   vmm_notification ntfy = null);
Defines the specified notification using the specified notification descriptor. If the descriptor is null, the notification is undefined and can only be indicated using the vmm_notify::indicate() method. If a notification is already defined, the new definition replaces the previous definition.

virtual function vmm_notification
   get_notification(int notification_id);
Gets the notification descriptor associated with the specified notification, if any. If no notification descriptor is associated with the specified notification, null is returned.

virtual function void reset( int notification_id = -1,
   reset_e rst_typ = SOFT);
Resets the specified notification. A vmm_notify::SOFT reset clears the specified ON_OFF notification and restarts the vmm_notification::indicate() and vmm_notification::reset() methods on any attached notification descriptor. A vmm_notify::HARD reset clears all status information and attached
notification descriptor on the specified event and further assumes that no threads are waiting for that notification. If no notification is specified, all notifications are reset.


```verilog
class bus_mon extends vmm_xactor;
static int EVENT_A = 0;
static int EVENT_B = 1;
static int EVENT_C = 2;

function new(...);
    super.new(...);
    super.notify.configure(this.EVENT_A);
    super.notify.configure(this.EVENT_B, vmm_notify::ON_OFF);
    super.notify.configure(this.EVENT_C, vmm_notify::BLAST);
endfunction
endclass: bus_mon
```

vmm_notification

This class is used to describe a notification that can be autonomously indicated or reset based on a user-defined behavior, such as the composition of other notifications or external events. Notification descriptors are attached to notifications using the `vmm_notify::set_notification()` method.

`virtual task indicate(ref vmm_data status);`

Defines a method that, when it returns, causes the notification attached to the descriptor to be indicated. The value of the `status` argument is used as the indicated notification status descriptor. This method is automatically invoked by the notification service interface when a notification descriptor is attached to a notification using the `vmm_notify::set_notification()` method.

This method must be overloaded in user-defined class extensions. It can be used to implement arbitrary notification mechanisms, such as notifications based on a complex composition of other indications (e.g., notification expressions) or external events.

`virtual task reset();`

Defines a method that, when it returns, causes the ON/OFF notification attached to the notification descriptor to be reset. This method is automatically invoked by the notification service interface when a notification definition is attached to a `vmm_notify::ON_OFF` notification.
This method must be overloaded in user-defined class extensions.

**Example A-10.** Notification Indicated When Two Other Notifications Are Indicated

```systemverilog
class notify_a_and_b extends vmm_notification;
    local vmm_notify notify;
    local int    a;
    local int    b;

    function new(vmm_notify notify,
                 int    a,
                 int    b) {
        this.notify = notify;
        this.a      = a;
        this.b      = b;
    }

    virtual task indicate(ref vmm_data status)
        fork
            void = this.notify.wait_for(a);
            void = this.notify.wait_for(b);
        join
    endtask
endclass: notify_a_and_b

class bus_mon extends vmm_xactor;

    static int EVENT_A = 0;
    static int EVENT_B = 1;
    static int EVENT_C = 2;

    function new(...);
        super.new(....);
        super.notify.configure(this.EVENT_A);
        super.notify.configure(this.EVENT_B,
                                vmm_notify::ON_OFF);
        super.notify.configure(this.EVENT_C,
                                vmm_notify::BLAST);

        begin
            notify_a_and_b AB = new(super.notify,
                                      this.EVENT_A,
                                      this.EVENT_B);
                                      super.notify.set_notification(this.EVENT_C,
                                                                     AB);
        end
    endfunction
endclass: bus_mon
```
VMM_XACTOR

This base class is to be used as the basis for all transactors, including bus-functional models, monitors and generators. It provides a standard control mechanism expected to be found in all transactors. The guidelines covering the development of transactors based on this class can be found in section titled "Transactors" on page 161.

```cpp
function new( string name,
             string instance,
             int stream_id = -1);
```

Creates an instance of the transactor base class, with the specified name, instance name and optional stream identifier. The name and instance name are used to create the message service interface in the `vmm_xactor::log` property and the specified stream identifier is used to initialize the `vmm_xactor::stream_id` property.

```cpp
virtual function string get_name();
virtual function string get_instance();
```

Returns the name and instance name of this transactor respectively.

```cpp
vmm_log log;
```

Message service interface for messages issued from within this transactor instance.

```cpp
int stream_id;
```

Unique identifier for the stream of transaction and data descriptors flowing through this transactor instance. It should be used to set the `vmm_data::stream_id` property of the descriptors as they are received or randomized by this transactor.

```cpp
virtual function void
    prepend_callback(vmm_xactor_callbacks cb);
virtual function void
    append_callback(vmm_xactor_callbacks cb);
```

Prepends or appends the specified callback façade instance with this instance of the transactor. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once with the same transactor. A façade instance can be registered with more than one transactor. Callback façade instances can be unregistered and re-registered dynamically.
virtual function void unregister_callback(vmm_xactor_callbacks cb);
Unregisters the specified callback façade instance for this transactor instance. A warning is issued if the specified façade instance is not currently registered with the transactor. Callback façade instances can later be re-registered with the same or another transactor.

vmm_notify notify;
enum {XACTOR_IDLE;
    XACTOR_BUSY;
    XACTOR_STARTED;
    XACTOR_STOPPED;
    XACTOR_RESET};
Notification service interface and pre-configures notifications to indicate the state and state transitions of the transactor. The vmm_xactor::XACTOR_IDLE and vmm_xactor::XACTOR_BUSY notifications are vmm_notify::ON_OFF. All other events are vmm_notify::ONE_SHOT.

virtual function void start_xactor();
Starts the execution threads in this transactor instance. The transactor can later be stopped. Any extension of this method must call super.start_xactor(). The base class indicates the vmm_xactor::XACTOR_STARTED and vmm_xactor::XACTOR_BUSY notifications and resets the vmm_xactor::XACTOR_IDLE notification.

virtual function void stop_xactor();
Stops the execution threads in this transactor instance. The transactor can later be restarted. Any extension of this method must call super.stop_xactor(). The transactor will actually stop when the vmm_xactor::wait_if_stopped() or vmm_xactor::wait_if_stopped_or_empty() method is called. It is calls to these methods that define the granularity of stopping a transactor.

virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
Resets the state and terminates the execution threads in this transactor instance, according to the specified reset type. The base class indicates the vmm_xactor::XACTOR_RESET and vmm_xactor::XACTOR_IDLE notifications and resets the vmm_xactor::XACTOR_BUSY notification.
To facilitate the implementation of this method, the actual values associated with these symbolic properties are of increasing magnitude (e.g., `vmm_xactor::FIRM_RST` is greater than `vmm_xactor::SOFT_RST`). Not all reset types may be implemented by all transactors. Any extension of this method must call `super.reset_xactor(rst_type)` first to terminate the `vmm_xactor::main()` method, reset the notifications and reset the main thread seed according to the specified reset type. Calling `super.reset_xactor()` with a reset type of `vmm_xactor::PROTOCOL_RST` is functionally equivalent to `vmm_xactor::SOFT_RST`.

<table>
<thead>
<tr>
<th>Enumerated Value</th>
<th>Broadcasting Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmm_xactor::SOFT_RST</code></td>
<td>Clears the content of all channels, resets all ON_OFF notifications and terminates all execution threads but maintains the current configuration, notification service and random number generation state information. The transactor must be restarted. This reset type must be implemented.</td>
</tr>
<tr>
<td><code>vmm_xactor::PROTOCOL_RST</code></td>
<td>Equivalent to a reset signaled via the physical interface. The information affected by this reset is user defined.</td>
</tr>
<tr>
<td><code>vmm_xactor::FIRM_RST</code></td>
<td>Like <code>SOFT_RST</code>, but resets all notification service interface and random-number-generation state information. This reset type must be implemented.</td>
</tr>
<tr>
<td><code>vmm_xactor::HARD_RST</code></td>
<td>Resets the transactor to the same state found after construction. The registered callbacks are unregistered.</td>
</tr>
</tbody>
</table>

**protected task wait_if_stopped()**

**protected task wait_if_stopped_or_empty(vmm_channel chan)**

Blocks the thread execution if the transactor has been stopped via the `stop_xactor()` method or if the specified input channel is currently empty. These methods will indicate the `vmm_xactor::XACTOR_STOPPED` and `vmm_xactor::XACTOR_IDLE` notifications and reset the
vmm\_xactor::XACTOR\_BUSY notification. The tasks will return once the transactor has been restarted using the \texttt{start\_xactor()} method and the specified input channel is not empty. These methods do not block if the transactor is not stopped and the specified input channel is not empty.

Calls to these methods define the granularity by which the transactor can be stopped without violating the protocol. If a transaction can be suspended in the middle of its execution, the \texttt{wait\_if\_stopped()} method should be called at every opportunity. If a transaction cannot be suspended, the \texttt{wait\_if\_stopped\_or\_empty()} method should only be called after the current transaction has been completed, before fetching the next transaction descriptor for the input channel.

\textbf{Example A-11. Stopping a Transactor Execution at Appropriate Points}

\begin{verbatim}
protected virtual task main();
   fork
      super.main();
   join none
   while (1) begin
      transaction tr;
      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.get(tr);
      ...
      this.wait_if_stopped();
      ...
   end
endtask: main
\end{verbatim}

\texttt{protected virtual task main();}
This task is forked off whenever the \texttt{start\_xactor()} method is called. It is terminated whenever the \texttt{reset\_xactor()} method is called. The functionality of a user-defined transactor must be implemented in this method. Any additional subthreads must be started within this method, not in the constructor. It can have a blocking or non-blocking implementation.

Any extension of this method must first fork a call to \texttt{super.main()}. 

\texttt{virtual function void save\_rng\_state();}
This method should save, in local properties, the state of all random generators associated with this transactor instance.
virtual function void restore_rng_state();
This method should restore, from local properties, the state of all random generators
associated with this transactor instance.

virtual function void xactor_status(string prefix = "");
Displays the current status of the transactor instance in a human-readable format
using the message service interface found in the vmm_log::log property, using
vmm_log::NOTE_TYP messages. Each line of the status information is prefixed
with the specified prefix.

`vmm_callback(callback_class_name, method(args));
This macro simplifies the syntax of invoking callback methods in a transactor. For
example, instead of:

    foreach (this.callbacks[i]) begin
        ahb_master_callbacks cb;
        if ($cast_assign(cb, this.callbacks[i])) continue;
        cb.ptr_tr(this, tr, drop);
    end

Use:

    `vmm_callback(ahb_master_callbacks, ptr_tr(this, tr, drop));

vmm_xactor_callbacks
This class implements a pure virtual base class for callback containments. See the
documentation for the vmm_xactor::append_callback() method on
page 411.

VMM_ATOMIC_GEN
A macro is used to define a class named <class_name>_atomic_gen for any
user-specified class derived from vmm_data\(^1\), using a process similar to the
`vmm_channel macro.

\(^1\) With a constructor callable without any arguments.
The atomic generator class is an extension of the \texttt{vmm\_xactor} class and as such, inherits all of the public interface elements provided in the base class.

```
\texttt{\textasciitilde vmm\_atomic\_gen(class\_name, \textasciitilde Class Description\textasciitilde);}
```

Defines an atomic generator class named \texttt{<class\_name>\_atomic\_gen} to generate instances of the specified class. The generated class must be derived from the \texttt{vmm\_data} class and the \texttt{<class\_name>\_channel} class must exist.

```
function new( string instance,
    int stream\_id = -1,
    <class\_name>\_channel out\_chan = null);
```

Creates a new instance of the \texttt{<class\_name>\_atomic\_gen} class with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel instance is specified, one will be created internally in the \texttt{<class\_name>\_atomic\_gen::out\_chan} property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with “\textit{Atomic Generator}”.

```
<class\_name>\_channel out\_chan;
```

References the output channel for the instances generated by this transactor. The output channel may have been specified via the constructor. If no output channel instances were specified, a new instance is automatically created. This reference in this property may be dynamically replaced but the generator should be stopped during the replacement.

```
int unsigned stop\_after\_n\_insts;
```

The generator will stop after the specified number of object instances has been generated and consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own.

The default value of this property is 0.

```
<class\_name> randomized\_obj;
```

Transaction or data descriptor instance that is repeatedly randomized to create the random content of the output descriptor stream. The individual instances of the output
stream are copied from this instance, after randomization, using the vmm_data::copy() method.

The atomic generator uses a class factory pattern to generate the output stream instances. The generated stream can be constrained using various techniques on this property.

The vmm_data::stream_id property of this instance is set to the generator’s stream identifier before each randomization. The vmm_data::data_id property of this instance is also set before each randomization. It will be reset to 0 when the generator is reset and after the specified maximum number of instances has been generated.

**enum {GENERATED};**  
Notification identifier for the notification service interface in the vmm_xactor::notify property provided by the vmm_xactor base class. It is configured as a vmm_xactor::ONE_SHOT notification and is indicated immediately before an instance is added to the output channel. The generated instance is specified as the status of the notification.

**enum {DONE};**  
Notification identifier for the notification service interface in the vmm_xactor::notify property provided by the vmm_xactor base class. It is configured as a vmm_xactor::ON_OFF notification and is indicated when the generator stops because the specified number of instances has been generated. No status information is specified.

**virtual task inject(<class_name> data, ref bit dropped);**  
Injects the specified transaction or data descriptor in the output stream. Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances generated by this generator and will be subjected to the callback methods. The method returns once the instance has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.
This class implements a façade for atomic generator, transactor, callback methods. This class is automatically declared and implemented for any user-specified class by the atomic generator macro.

Callback method invoked by the generator after a new transaction or data descriptor has been created and randomized but before it is added to the output channel.

The gen argument refers to the generator instance that is invoking the callback method (in case the same callback extension instance is registered with more than one transactor instance). The data argument refers to the newly generated descriptor—which can be modified. If the value of the drop argument is set to non-zero, the generated descriptor will not be forwarded to the output channel, but the remaining registered callbacks will still be invoked.

A macro is used to define a class named <class_name>_scenario_gen for any user-specified class derived from vmm_data, using a process similar to the `vmm_scenario_gen` macro.

The scenario generator class is an extension of the vmm_xactor class and as such, inherits all of the public interface elements provided in the base class.

Defines a scenario generator class to generate sequences of related instances of the specified class. The specified class must be derived from the vmm_data class and the <class_name>_channel class must exist. It must also have a constructor with no arguments or that has default values for all of its arguments.

2. With a constructor callable without any arguments.
vmm_scenario_gen

The macro defines classes named `<class_name>_scenario_gen`, `<class_name>_scenario`, `<class_name>_scenario_election` and `<class_name>_scenario_gen_callbacks`.

```plaintext
function new( string instance,
             int stream_id = -1,
             <class_name>_channel out_chan = null);
```

Creates a new instance of a scenario generator transactor with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel is specified, one will be created internally in the `<class_name>_scenario_gen::out_chan` property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with “Scenario Generator”.

```plaintext
<class_name>_channel out_chan;
```

References the output channel for the instances generated by this transactor. The output channel may have been specified via the constructor. If no output channel was specified, a new instance is automatically created. The reference in this property may be dynamically replaced but the generator should be stopped during the replacement.

```plaintext
int unsigned stop_after_n_insts;
```

The generator will stop after the specified number of transaction or data descriptor instances have been generated and consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own based on the number of generated instances (but may still stop based on the number of generated scenarios).

The default value of this property is 0.

```plaintext
int unsigned stop_after_n_scenarios;
```

The generator will stop after the specified number of scenarios have been generated and entirely consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own based on the number of generated scenarios (but may still stop based on the number of generated instances).

The default value of this property is 0.
Set of available scenario descriptors that may be repeatedly randomized to create the random content of the output stream. The \texttt{<class_name>_scenario_gen::select_scenario} property is used to determine which scenario descriptor, out of the available set of descriptors, is randomized next. The individual instances of the output stream are then created by calling the \texttt{<class_name>_scenario::apply()} method of the randomized scenario descriptor.

By default, this property contains one instance of the atomic scenario descriptor \texttt{<class_name>atomic_scenario}. Out of the box, the scenario generator will generate individual random descriptors.

The \texttt{vmm_data::stream_id} property of the randomized instance is assigned the value of the generator’s stream identifier before randomization. The \texttt{vmm_data::scenario_id} property of the randomized instance is assigned a unique value before randomization. It will be reset to 0 when the generator is reset and after the specified number of instances or scenarios has been generated.

References the scenario descriptor selector that is repeatedly randomized to determine which scenario descriptor, out of the available set of scenario descriptors, will be randomized next.

By default, a round-robin selection process is used. The constraint blocks or randomized properties in this instance can be turned off or the instance can be replaced with a user-defined extension to modify the election rules.

\textbf{enum \{GENERATED\};}
Notification identifier for the \texttt{vmm_xactor::notify} notification service interface provided by the \texttt{vmm_xactor} base class. It is configured as a \texttt{vmm_notify::ONE_SHOT} notification and is indicated immediately before a scenario is applied to the output channel. The randomized scenario is specified as the status of the notification.

\textbf{enum \{DONE\};}
Notification identifier for the \texttt{vmm_xactor::notify} notification service interface provided by the \texttt{vmm_xactor} base class. It is configured as a \texttt{vmm_notify::ON_OFF} notification and is indicated when the generator stops.
because the specified number of instances or scenarios has been generated. No status information is specified.

**virtual task inject_obj(<class_name> obj);**
Injests the specified descriptor in the output stream. Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods as an atomic scenario. The method returns once the descriptor has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

**virtual task inject(<class_name>_scenario scenario);**
Injests the specified scenario descriptor in the output stream. Unlike injecting the descriptors directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods. The method returns once the scenario has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

**<class_name>_scenario**
This class implements a base class for describing scenarios or sequences of transaction descriptors. This class named `<class_name>_scenario` is automatically declared and implemented for any user-specified class named “`class_name`” by the scenario generator macro, using a process similar to the `vmm_channel` macro.

**static vmm_log log;**
Message service interface to be used to issue generic messages when the message service interface of the scenario generator is not available or in scope.

**int stream_id;**
Stream identifier. It is set by the scenario generator before the scenario descriptor is randomized. Can be used to express stream-specific constraints.
int scenario_id;
Scenario identifier within the stream. It is set by the scenario generator before the
scenario descriptor is randomized and incremented after each randomization. Can be
used to express scenario-specific constraints. The scenario identifier is reset to 0
when the scenario generator is reset or when the specified number of scenarios has
been generated.

function int unsigned
define_scenario( string name,
               int unsigned max_len);
Defines a new scenario with the specified name and the specified maximum number
of transactions or data descriptors. Returns a unique scenario identifier that should be
assigned to an int unsigned property.

function void
redefine_scenario( int unsigned scenario_kind,
                  string name,
                  int unsigned max_len);
Redefines the name and maximum number of descriptors in a previously defined
scenario. Used to redefine an existing scenario instead of creating a new one and
constraining the original scenario out of existence.

function string
scenario_name(int unsigned scenario_kind);
Returns the name associated with the specified scenario identifier.

rand int unsigned scenario_kind;
When randomized, selects the identifier of the scenario that is generated. Constrained
to the known scenario identifiers defined using the
<class_name>_scenario::define_scenario() method. Can be
constrained to modify the distribution of generated scenarios.

rand int unsigned length;
Randomized number of items in the scenario. Defines how many instances in the
<class_name>_scenario::items[] property are part of the scenario.
vmm_scenario_gen

**rand** `<class_name>` *items[]*;

Instances of user-specified `<class_name>` that are randomized to form the scenarios. Only elements from index 0 to `<class_name>:_scenario::length`-1 are part of the scenario.

The constraint blocks and **rand** attributes of the instances in the randomized array may be turned **ON** or **OFF** to modify the constraints on scenario items. They can also be replaced with extensions.

By default, the output stream is formed by **copying** the values of the items in this array onto the output channel.

`<class_name>` *using*;

Instance used in the default implementation of the **pre_randomize()** method when invoking the **fill_scenario()** method. Set to **null** by default. Can be replaced by an instance of a derived class to subject the items of the scenario to different constraints or content.

**rand int unsigned repeated**;

Number of times the items in the scenario are applied. The repeated instances in the scenario count toward the total number of instances generated but only one scenario is considered generated, regardless of the number of times it is repeated.

This property is unconstrained by default. To avoid accidentally repeating a scenario many times, a warning message will be issued if the value of this property is greater than the value specified in the `repeat_thresh` property.

**static int unsigned repeat_thresh**;

To avoid accidentally repeating a scenario many times because the **repeated** property was left unconstrained, a warning message will be issued if the value of the `repeated` property is greater than the value specified in this property. The default value is 100.

**function void**

`allocate_scenario(<class_name> using = null);`

Allocates a new set of instances in the *items* property, up to the maximum number of items in the maximum-length scenario. Any instance previously located in the *items* array is replaced. If a reference to an instance is specified in the *using*
argument, the array is filled by calling `vmm_data::copy()` on the specified instance. Otherwise, the array is filled with a new instance of `<class_name>` class.

```c
function void fill_scenario(<class_name> using = null);
```

Allocates new instances in the `items` property, up to the maximum number of items in the maximum-length scenario in any `null` element of the array. Any instance previously located in the `items` array is left untouched. If a reference to an instance is specified in the `using` argument, the array is filled by calling `vmm_data::copy()` on the specified instance. Otherwise, the array is filled with a new instance of `<class_name>` class.

```c
virtual task apply( <class_name>_channel channel,
    ref int unsigned n_insts);
```

Applies the items in the scenario descriptor to the specified output channel and returns when they have all been consumed by the channel. The `n_insts` argument is set to the number of instances that were consumed by the channel. By default, copies the values of the `items` array using their `vmm_data::copy()` method.

This method may be overloaded to define procedural scenarios.

**<class_name>._atomic_scenario**

This class implements a predefined atomic scenario descriptor. An atomic scenario is composed of a single unconstrained transaction or data descriptor. This class named `<class_name>._atomic_scenario` is automatically implemented for any user-specified class named “<class_name>” by the scenario generator macro, using a process similar to the `vmm_channel` macro.

```c
int unsigned ATOMIC;
```

Symbolic scenario identifier for the atomic scenario described by this descriptor. The atomic scenario is a single, random, unconstrained, transaction descriptor (i.e., an atomic descriptor).

```c
constraint atomic_scenario;
```

Specifies the constraints of the atomic scenario. By default, the atomic scenario is a single unrepeated unconstrained item. This constraint block may be overridden to redefine the atomic scenario.
<class_name>_scenario_election

This class implements a random selection process for selecting the next scenario descriptor, from a set of available descriptors, to be randomized next. This class named <class_name>_scenario_election is automatically implemented for any user-specified class named “class_name” by the scenario generator macros, using a process similar to the vmm_channel macro.

int stream_id;
Stream identifier. It is set by the scenario generator to the value of the generator stream identifier before the scenario selector is randomized. Can be used to express stream-specific constraints.

int scenario_id;
Scenario identifier within the stream. It is set by the scenario generator before the scenario selector is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset or when the specified number of scenarios has been generated.

int unsigned n_scenarios;
Number of available scenario descriptors in the scenario set. The final value of the select property must be in the [0:n_scenarios-1] range.

int unsigned last_selected[$];
A history (maximum of 10) of the last scenario selections. Can be used to express constraints based on the historical distribution of the selected scenarios (e.g., “never select the same scenario twice in a row”).

int unsigned next_in_set;
The next scenario descriptor index that would be selected in a round-robin selection process. Used by the round_robin constraint block.

<class_name>_scenario scenario_set[$];
The available set of scenario descriptors. Can be used to procedurally determine which scenario to select or to express constraints based on the scenario descriptors.
rand int select;
The index, within the scenario_set array, of the selected scenario descriptor to be randomized next.

constraint round_robin;
Constrains the random scenario selection process to a round-robin selection. This constraint block may be turned off to produce a random scenario selection process or allow a different constraint block to define a different scenario selection process.

$class_name>_scenario_gen_callbacks
This class implements a façade for callback containments for the scenario generator transactor. This class named $class_name>_scenario_gen_callbacks is automatically implemented for any user-specified class named “class_name” by the scenario generator macro, using a process similar to the ‘vmm_channel macro.

virtual task pre_scenario_randomize(
  $class_name>_scenario_gen gen,
  ref $class_name>_scenario scenario);
Callback method invoked by the generator after a new scenario has been selected but before it is randomized. The gen argument refers to the generator instance that is invoking the callback method. The scenario argument refers to the newly selected scenario descriptor which can be modified. Note that any modifications of the randomization state of the scenario descriptor—such as turning constraint blocks ON or OFF—will remain in effect the next time the scenario descriptor is selected to be randomized. If the reference to the scenario descriptor is set to null, the scenario will not be randomized and a new scenario will be selected.

To minimize memory allocation and collection, it is possible that the elements of the scenarios may not be allocated. Use the
$class_name>_scenario::allocate_scenario() or $class_name>_scenario::fill_scenario() to allocate the elements of the scenario if necessary.

virtual task post_scenario_gen(
  $class_name>_scenario_gen gen,
  $class_name>_scenario scenario,
  ref bit dropped);
Callback method invoked by the generator after a new scenario has been randomized but before it is applied to the output channel. The gen argument refers to the
generator instance that is invoking the callback method. The \texttt{scenario} argument refers to the newly randomized scenario that can be modified. Note that any modifications of the randomization state of the scenario descriptor—such as turning constraint blocks ON or OFF—will remain in effect the next time the scenario descriptor is selected to be randomized. If the value of the \texttt{dropped} argument is set to non-zero, the generated instance will not be applied to the output channel.
This appendix describes the checkers currently available in the VMM Checker Library. In the first group, there are 31 checkers that are equivalent to the checkers in the Accellera OVL but contain extensions for coverage. A second group of 19 checkers verify more complex behaviors than those in the first group.

Assertions are enabled globally by defining the symbol `ASSERT_ON`. If this symbol is not defined, the code for all checkers is physically removed at compile-time.

All the checkers contain coverage statements that can be globally enabled by defining the `COVER_ON` symbol. In addition, three coverage levels can be independently enabled on a per-instance basis. Level 1 coverage provides an indication of the coverage of the trigger conditions of the checker and, in some cases, of the basic functionality. Level 2 coverage collects data on the profiles of delay or data values observed during the simulation. Finally, Level 3 coverage provides information on the occurrence of corner cases such as hitting the user-specified minimum and maximum values on delays and value ranges.

**OVL-EQUIVALENT CHECKERS (SVL)**

This section describes the 31 OVL-like checkers. All checkers, except `assert_proposition`, are triggered at the positive edge of a triggering signal or expression `clk`. A clock cycle is defined as the duration between two consecutive positive edges of the clock signal.
The values of all actual signals or expressions in the ports of the checkers are sampled just before the positive edge of \( \text{clk} \). Therefore, any pulses happening on the signals or expressions between consecutive positive edges of \( \text{clk} \) are not observed by the checkers.

Moreover, whenever an edge of a signal or expression on a port of a checker other than the clock is used in the checker, it is the sampled from the edge, as detected by looking at two consecutive samples of the signal.

The checker \textit{assert\_proposition} monitors an expression at all times but fires only when the \textit{test\_expr} undergoes a falling transition (from 1 or x or z to 0 (false)).

In the following descriptions, an assertion that \textit{fires} means that an error condition has been detected.

\texttt{assert\_always} — Continuously monitors \textit{test\_expr} at every positive edge of clock signal \( \text{clk} \). \textit{test\_expr} must always evaluate to \textit{true}. If \textit{test\_expr} evaluates to \textit{false}, the assertion fires. The \textit{test\_expr} can be any valid expression.

\texttt{assert\_always\_on\_edge} — Continuously monitors the \textit{test\_expr} at every specified edge of the \textit{sampling\_event}. \textit{test\_expr} must always evaluate \textit{true} at the \textit{sampling\_event}. If \textit{test\_expr} evaluates to \textit{false}, the assertion fires. Note that the transition on the sampling event is determined by sampling \textit{sampling\_event} at two consecutive positive edges of the clock signal \( \text{clk} \).

\texttt{assert\_change} — Continuously monitors the \textit{start\_event} at every positive edge of the clock signal \( \text{clk} \). When \textit{start\_event} is \textit{true}, the checker ensures that the expression, \textit{test\_expr} changes values on a clock edge at some point within the next \textit{num\_cks} number of clocks.

\texttt{assert\_cycle\_sequence} — Verifies the following conditions:

- When \textit{necessary\_condition} = 0, if all \textit{num\_cks}-1 first bits of a vector of Boolean events (\textit{event\_sequence}[num\_cks-1:1]) are \textit{true} (1) in consecutive clock cycles, the last Boolean (\textit{event\_sequence}[0]) must be \textit{true} in the next clock cycle.

- When \textit{necessary\_condition} = 1, if the first bit of a vector of (\textit{event\_sequence}[num\_cks-1]) is \textit{true}, then all the remaining \textit{event\_sequence}[num\_cks-2:0] bits must become true in the subsequent \textit{num\_cks}-1 clock cycles.
assert_decrement — Continuously monitors test_expr at every positive edge of the clock signal clk. It checks that test_expr always decreases by the value specified by value. The test_expr can be any valid expression. The checker will not start until the first clock edge after reset_n is asserted.

assert_delta — Continuously monitors test_expr at every positive edge of clock signal clk. It verifies that test_expr always changes value by a value greater than or equal to min and less than or equal to max value. The test_expr can be any valid expression. The checker will not start until the first clock edge after reset_n is asserted.

assert_even_parity — Ensures that the variable, test_expr, has an even number of bits set to 1 at any positive edge of the clock signal clk.

assert_fifo_index — Ensures that a FIFO element a) never overflows or underflows b) allows/disallows simultaneous push and pop operations.

assert_frame — Validates proper cycle timing relationships between two events in the design. When a start_event (a bit) evaluates true, then test_expr must evaluate true within a minimum and maximum number of clock cycles.

assert_handshake — Continuously monitors the req and ack signals at every positive edge of the clock signal clk. It ensures that ack occurs after req within a specified minimum and maximum number of clock cycles. Both req and ack must go inactive prior to starting a new cycle. Verifying that req is persistent until ack arrives and that it remains active for some cycle after ack is controlled by checker parameters.

assert_implication — Continuously monitors antecedent_expr. If it evaluates to true, then it verifies that the consequent_expr is true. When antecedent_expr evaluates to false, then consequent_expr expression will not be checked at all and the implication is satisfied.

assert_increment — Continuously monitors test_expr at every positive edge of the clock signal clk. It verifies that test_expr increases by the value specified by value. The test_expr can be any valid expression. The checker will not start until the first clock edge after reset_n is asserted.

assert_never — Continuously monitors test_expr at every positive edge of the clock signal clk. It verifies that test_expr never evaluates true. The test_expr can be any valid expression. When test_expr evaluates true, this checker fires.
**assert_next** — Validates proper cycle timing relationships between two events in the design. When a `start_event` evaluates *true*, then the `test_expr` must evaluate *true* exactly `num_cks` number of clock cycles later. This checker supports overlapping sequences.

**assert_no_overflow** — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. It verifies that a specified `test_expr` will never:

- Change value from a `max` value (default is `(2**width) - 1`) to a value greater than `max`, or
- Change value from a `max` value (default is `(2**width) - 1`) to a value less than or equal to a `min` value (default is 0).

**assert_no_transition** — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. When it evaluates to the value of `start_state`, it ensures that `test_expr` will never transition to the value of `next_state`. The `width` parameter defines the number of bits in `test_expr`.

**assert_no_underflow** — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. This checker verifies that `test_expr` will never:

- Change value from a `min` value (default is 0) to a value less than `min`, or
- Change to a value greater than or equal to `max` (default is `(2**width) - 1`).

**assert_odd_parity** — Ensures that the variable, `test_expr`, has an odd number of bits set to 1 at any positive edge of the clock signal `clk`.

**assert_one_cold** — Ensures that the variable, `test_expr`, has only one bit set to 0 at any positive clock edge when the checker is configured for no inactive states. The checker can also be configured to accept all bits equal to either 0 or 1 as the inactive level.

**assert_one_hot** — Ensures that the variable, `test_expr`, has only one bit set to 1 at any positive edge of the clock signal `clk`.

**assert_proposition** — Continuously monitors `test_expr` and verifies that `test_expr` always evaluate *true*. If `test_expr` transits from *true* to *false* while `reset_n` is 1, the checker fires. Unlike **assert_always**, `test_expr` is not sampled by a clock.
OVL-Equivalent Checkers (SVL)

**assert_quiescent_state** — continuously monitors `state_expr` at every positive edge of the sampling event `sample_event` and verifies that the value `state_expr` is equal to the value `check_value` and optionally at the end of simulation.

**assert_range** — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. The checker ensures that the `test_expr` is always within the `min` and `max` value range.

**assert_time** — Continuously monitors `start_expr`. When it evaluates `true`, the checker ensures that `test_expr` evaluates to `true` for the next `num_cks` number of clock cycles.

**assert_transition** — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. When `test_expr` evaluates to the value `start_state`, the checker ensures that `test_expr` will always change to the value of `next_state`. The `width` parameter defines the number of bits in `test_expr`.

**assert_unchange** — Continuously monitors `start_event` at every positive edge of the clock signal `clk`. When `start_event` evaluates `true`, the checker ensures that `test_expr` will not change value within the next `num_cks` number of clock cycles.

**assert_width** — Continuously monitors `test_expr`. When `test_expr` evaluates `true`, it ensures that `test_expr` evaluates to `true` for a specified `minimum` number of clock cycles and does not exceed a `maximum` number of clock cycles.

**assert_win_change** — Continuously monitors `start_event` at every positive edge of the clock signal `clk`. When `start_event` evaluates `true`, it ensures that `test_expr` changes values prior to and including the occurrence of `end_event`.

**assert_win_unchange** — Continuously monitors `start_event` at every positive edge of the clock signal `clk`. When `start_event` evaluates `true`, it ensures that `test_expr` will not change in value up to and including `end_event` becoming `true`.

**assert_window** — Continuously monitors `start_event` at every positive edge of the clock signal `clk`. When `start_event` evaluates `true`, it ensures that the `test_expr` evaluates `true` at every successive positive clock edge of `clk` up to and including the `end_event` expression becoming `true`. This checker does not evaluate
VMM Checker Library

test_expr on start_event. It begins evaluating test_expr at the next positive clock edge of clk.

assert_zero_one_hot — Continuously monitors test_expr at every positive edge of the clock signal clk. It verifies that test_expr has exactly one bit asserted or no bit asserted.

ADVANCED CHECKERS

This section describes 19 advanced checkers. These advanced checkers use the same controls as the OVL-equivalent checkers described in the previous section. In addition, they have a clock edge selection parameter, edge_expr, that lets the user select posedge or negedge clock edge selection for sampling in the assertions and cover statements.

assert_arbiter — Ensures that a resource arbiter provides grants to corresponding requests within min_lat and max_lat cycles. reqs and grants are vectors of size [no_chnl-1:0] where the bits correspond to the individual channels. They are assumed to be 1 when active. The checker can verify a priority arbitration scheme alone or in conjunction with (as a secondary criterion) round-robin, FIFO or LRU selection algorithms. The checks are not enabled unless reset_n evaluates true.

assert_bits — Ensures that the value of exp has between min and max number of bits that are asserted or deasserted as indicated by the deasserted flag. The check is not enabled unless reset_n evaluates true.

assert_code_distance — Ensures that when exp changes, the number of bits that are different compared to exp2—the Hamming distance—are at least min but no more than max in number. The check is not enabled unless reset_n evaluates true.

assert_data_used — Ensures that data from src[sleft:sright] appears in dest[dleft:dright] within the window specified as start cycles from after the time trigger is asserted until finish number of cycles after trigger is asserted.

assert_driven — Ensures that all bits of exp are driven (i.e., none are ‘Z’ or ‘X’). The check is not enabled unless reset_n evaluates true.
assert_dual_clk_fifo — Checker for a dual-clock, single-input and single-output FIFO. It assumes that enqueuing is enabled when $enq$ is asserted at the active clock edge of $enq_{clk}$ and effectively occurs $enq_{lat}$ cycles later. Dequeuing is enabled when $deq$ is asserted at the active edge of $deq_{clk}$ and effectively occurs $deq_{lat}$ cycles later. It can verify that neither overflow or underflow of the FIFO occurs, that it reaches a watermark and that the enqueued data value is the correct one upon dequeue.

assert_fifo — Checker for a single-clock, single-input and single-output FIFO. All signals are sampled at the active edge of the clock signal $clk$. It assumes that enqueuing is enabled when $enq$ is asserted and effectively occurs $enq_{lat}$ cycles later. Dequeuing is enabled when $deq$ is asserted and effectively occurs $deq_{lat}$ cycles later. It can verify that neither overflow or underflow of the FIFO occurs, that it reaches a watermark and that the enqueued data value is the correct one upon dequeue. Also, if $pass_{thru}$ is 1, it allows simultaneous enqueue and dequeue of data on empty or full queue. Otherwise a dequeue on an empty queue will report an underflow.

assert_hold_value — Ensures that $exp$ of width $bw$ remains at $value$ for $min$ to $max$ number of cycles. That is, it must stay at $value$ for $min$ cycles, then it may change and after $max$ cycles it must change to some other value. The check is not enabled unless $reset_{n}$ evaluates true.

assert_memory_async — Ensures the integrity of an asynchronous memory content and access. When $addr_{chk}$ evaluates true, it ensures that $start_{addr} \leq raddr \leq end_{addr}$ as sampled by the negedge of $ren$, and that $start_{addr} \leq waddr \leq end_{addr}$ as sampled by the negedge of $wen$. All other checks apply only if the address is valid. There is no clock other than the $ren$ and $wen$ expressions that indicate when each operation is to take place by their falling edges.

Checks can also be enabled to verify that memory locations are written into before being read, that there is at least one read between two consecutive writes to an address, or similarly that there is at least one write between two consecutive reads to an address. The checker can also verify that the value written last to a memory location is the one being read out later.

assert_memory_sync — Ensures the integrity of a synchronous memory content and access. When $addr_{chk}$ evaluates true, it ensures that $start_{addr} \leq raddr \leq end_{addr}$ when $ren$ is true as sampled by the active edge of $rclk$, and that $start_{addr} \leq waddr \leq end_{addr}$ when $wen$ is true at the active edge of $wclk$. All other checks apply only if the address is valid.
Checks can also be enabled to verify that memory locations are written into before being read, that there is at least one read between two consecutive writes to an address, or similarly that there is at least one write between two consecutive reads to an address. The occurrence of simultaneous read and write operation when rclk is the same as wclk can be verified. The checker can also verify that the value written last to a memory location is the one being read out later or at the same time if pass_thru is enabled.

**assert_multiport_fifo** — Checker for a single-clock, multi-input and multi-output FIFO. *enq* and *deq* are bit vectors of equal size *no_ports*. Each pair of corresponding bits in these vectors defines the enqueue and dequeue enable signals for a FIFO port. Bit 0 has the lowest priority, while the highest-order bit *no_ports−1* has the highest priority. The enqueue port and the dequeue port of the highest priority are processed at every active *clk* edge.

*enq_data* is a concatenation of the data from the different ports, dimensioned as [no_ports*elem_size−1:0], with data vectors appearing in the same order as the *enq* requests. Whenever a bit in *enq* is asserted 1, the corresponding data port in *enq_data* must be valid after *enq_lat* clock cycles. Only the highest-priority data is actually enqueued.

*deq_data* is a concatenation of the data from the different ports. It is assumed that it is dimensioned the same way as *enq_data*, with data vectors appearing in the same order as the *deq* requests. Whenever a bit in *deq* is asserted 1, the corresponding data port in *deq_data* must be valid after *deq_lat* clock cycles. Only the data of the highest-priority dequeue request is compared with the reference data when *value_chk* is 1. Overflow, underflow, watermark, value and pass-thru checks can be enabled as in the *assert_fifo* checker.

**assert_mutex** — Ensures that *a* and *b* never evaluate *true* at the same time. The checker is not enabled unless *reset_n* evaluates *true*.

**assert_next_state** — Ensures that, when *exp* is in current state *cs*, *exp* will transition to one of the specified legal next states in *ns*. *no_ns* specifies the number of legal next states. *ns* is a bit vector of the concatenated legal state values that *exp* can transition to from *cs*.

**assert_no_contention** — Ensures that *bus* always has a single active driver and that there is no ‘X’ or ‘Z’ on the bus when driven (*en_vector != 0*). The total number of *en_vector* bits that are asserted can be at most 1. *min_quiet* and *max_quiet* define and interval in the number of clock cycles within when the bus may remain quiet, i.e., no diver enabled.
assert_regLoaded — Ensures that the register dst_reg is loaded with src data. The check for dst_reg holding the memorized value of src starts with delay cycles (minimum 1, which is default) after the trigger condition evaluates true and within end_cycle cycles after the trigger evaluates true or when stop becomes true (whichever occurs first).

assert_req_ack_unique — Verifies that each req receives an ack within the specified interval min_time and max_time active clock edges of clk. The arriving ack’s are attributed to req’s in a FIFO order.

assert_stack — Verifies operations of a stack. When push is asserted 1, it ensures that there is no stack overflow. push_lat specifies the number of clock cycles between the assertion of push and when push_data is valid. Similarly, when pop is asserted 1, it ensures that the stack is not empty. pop_lat specifies the number of clock cycles between the assertion of pop and when pop_data must be valid. Data value, stack empty, full, watermark and pass-thru checks can be selectively enabled.

assert_valid_id — The signal issued_sig asserted 1 validates a request identified by the value in issued_id. This request is expected to be acknowledged by ret_id validated by ret_sig asserted 1 within [min_lat:max_lat] latency. A reset_sig asserted true with reset_id value of one of the currently issued and still outstanding IDs resets that outstanding ID to empty, i.e., a ret_sig asserted for the ID is then considered as invalid until newly issued.

The bit width id_bw of the IDs can be any value supported by the tool; however, the maximum number of outstanding IDs at any time is limited by the value of the parameter max_ids. For a given ID, there can be at most max_out_per_id outstanding issues. The arriving returns of that ID are matched in a FIFO manner to the requests when verifying the latency of the return (similarly as in the assert_req_ack_unique checker).

assert_value — Ensures that exp can only be one of the specified values in a set. no_vals indicates the number of values in the set, which is defined by a bit vector vals of width [bw*no_vals-1 : 0] of the concatenated values of bw bits each that exp must evaluate to.
This appendix specifies the detailed behavior of a set of base and utility classes that can be used to implement an XVC-compliant verification environment and verification components. The actual implementation of these classes is left to each tool provider.

XVC_MANAGER

This class is a base class for implementing XVC management functions, as described in “XVC Manager” on page 316. A predefined XVC manager, as described in “Pre-defined XVC Manager” on page 317 is specified in section titled “vmm_xvc_manager” on page 444.

vmm_log log;
Message service interface used to issue all messages from the XVC manager. The name is specified as “XVC Manager” and the instance name is the instance name of the xvc_manager instance, as specified in the constructor.

vmm_log trace;
Message service interface for execution trace messages that may be routed differently than the generic messages issued through the message service instance in the log class property.
vmm_notify notify;
Notification service interface for the global notifications. The event identifier is the
same as the global notification identifier specified in the test file. All events corre-
sponding to global notifications are triggered ONE_SHOT.

function new(string instance = "Main");
Creates an instance of the XVC manager with the specified instance name.

function bit add_xvc(xvc_xactor xvc);
function bit remove_xvc(xvc_xactor xvc);
Puts or removes the specified XVC instance under the control of this XVC manager.
Returns non-zero if the operation is successful and error-free. An XVC instance can-
not be under the control of more than one XVC manager at any given time.

XVC instances can only be added or removed from the control of an XVC manager
when a manager is not running a test.

function bit split(string command,
   ref string argv[]);
Splits the specified command into blank-separated tokens, suitable for the
xvc_xactor::parse() method. Quotes and escaped characters are interpreted
like the C shell when splitting arguments into main’s argv array.

protected xvc_xactor xvcQ[];
Array of XVC instances under the control of this XVC manager instance. The content
of this array is managed using the add_xvc() and remove_xvc() methods.

**XVC_XACTOR**
This class is a base class for implementing XVC-compliant transactors, as described
in “Extensible Verification Components” on page 306. This base class is derived from
the vmm_xactor class and offers the following additional interface elements:

vmm_log trace;
Message service interface for execution trace messages that may be routed differently
than the generic messages issued through the message service instance in the
vmm_xactor::log class property.
vmm_notify notify;
Notification service interface for the local notifications. The event identifier is the same as the local notification identifier specified in the test file. All events corresponding to local notifications are triggered ONE_SHOT.

function new(string name, string instance, int stream_id = -1, xvc_action_channel action_chan = null, xvc_action_channel interrupt_chan = null);
Creates an instance of the XVC transactor with the specified name and instance name and optional stream identifier. The input action and interrupt channels are optionally connected to the specified channel instances. Action and interrupt channels, if specified, are reconfigured to a full level of 1 and 64 k respectively. The name, instance name and stream identifier are used as the vmm_xactor name, instance name and stream identifier, respectively. The name and instance name will be used to configure the vmm_log instance found in the vmm_xactor base class.

function void add_action(xvc_action action);
Adds the specified XVC action descriptor to the known actions of the XVC transactor. New action definitions may hide previous definitions as they are considered by the parse() method in the reverse order of registration.

function xvc_action parse(string argv[]);
Parses the specified action command and returns the corresponding action descriptor. If the action is not known to the XVC transactor, null is returned. The command is specified as an array of string tokens similar to argv in C’s main() function argument.

xvc_action_channel action_chan;
Input channel for actions to be executed by the XVC transactor. If no channel instance was specified in the constructor, a new instance is internally allocated. The XVC transactor uses an in-order, blocking completion model, as described in section titled "In-Order Atomic Execution Model" on page 177.
**XVC Standard Library Specification**

**xvc_action_channel interrupt_chan;**
Input channel for interrupt actions to be executed by the XVC transactor at the earliest opportunity. If no channel instance was specified in the constructor, a new instance is internally allocated. An interrupt action will be executed after the current action execution completes or when the current action invokes the `wait_if_interrupted()` method. The XVC transactor uses an in-order, non-blocking completion model for interrupt actions, as described in section titled "Out-of-Order Atomic Execution Model" on page 182.

**protected task wait_if_interrupted();**
Suspends the execution thread if an interrupt action is waiting to be executed by the XVC. This method must only be called from within an implementation of the `xvc_action::execute()` method.

**protected vmm_channel exec_chan;**
Channel that must be used to execute the actions in the XVC.

**protected vmm_xactor xactors[];**
Lower-level transactors used by this XVC. Actions may require the registration of callback extensions to implement their execution.

**XVC_ACTION**

This class is a base class to implement XVC action descriptors. An action descriptor defines the command used to invoke it and how to execute it. Actions are XVC-specific and cannot be executed on different XVCs. This base class is derived from the `vmm_data` class and offers the following additional interface elements:

**function new( string name,**

```
    vmm_log log);
```

Creates a new instance of an action descriptor. The action is named using the specified name and the specified message interface is passed to the `vmm_log::new()` method.

**function string get_name();**
Returns the name of the action, as specified in the constructor.
virtual function xvc_action parse(string argv[]);

Parses the specified command and returns a new instance of the action descriptor that corresponds action descriptor. Returns null without issuing any error or warning messages if the command is not recognized.

virtual task execute(vmm_channel exec_chan,
                    xvc_xactor xvc);

Executes the action described by this instance of the action descriptor, through the specified input channel. The action is executed by generating and putting the necessary transaction descriptor in the specified input channel.

At appropriate points during the execution of the action, the xvc_xactor::wait_if_interrupted() method should be called to let interrupt actions be executed. The calls to this method define the granularity of the action execution. For example, an atomic action would never call xvc_xactor::wait_if_interrupted().

vmm_xactor_callbacks callbacks[];

Transactor callbacks extensions that must be registered with the transactors in the XVC to properly execute the action described by this instance of the action descriptor, prior to invoking the execute() method. If not null, the callback extension instance is prepended to the registered callbacks of the corresponding lower-level transactor in the XVC before the action is executed, then unregistered upon completion of the execution.

virtual function int unsigned byte_pack(ref logic [7:0] bytes[],
                                       input int unsigned offset = 0,
                                       input int kind = -1);

The default implementation packs the name of the action descriptor into the specified dynamic array of bytes, starting at the specified offset in the array and ending with a byte set to 8'h00. The array is resized appropriately. Returns the number of bytes added to the array.
virtual function int unsigned
  byte_unpack( const ref logic [7:0] bytes[],
  input int unsigned offset = 0,
  input int len = -1,
  input int kind = -1);

The default implementation unpacks the name of the action descriptor from the specified offset in the specified dynamic array until a byte set to 8'h00, the specified number of bytes have been unpacked or the end of the array is encountered, whichever comes first. Returns the number of bytes unpacked.

virtual function int unsigned byte_size(int kind = -1);
The default implementation returns the length of the action descriptor name plus one.

virtual function int unsigned
  max_byte_size(int kind = -1);
The default implementation returns the length of the action descriptor name plus one.

VMM_XVC_MANAGER

The class implements the predefined XVC manager as described in “Predefined XVC Manager” on page 317. It is implemented as an extension of the xvc_manager base class and provides the following additional elements.

task run(string testfile);
Starts all of the XVCs under the control of the manager and runs the test in the specified command file on the XVC manager instance. This task returns once the test has completed as defined in the test itself. The XVCs are not stopped nor reset when the test completes and are still running.

Notifications
The XVC manager base class provides a notification service interface in its xvc_manager::notify class property. The predefined XVC manager uses notifications to coordinate actions and XVCs. XVCs also use notifications in their respective notification service interface in their vmm_xactor::notify class property to coordinate with the predefined XVC manager.
Events are used in a scenario description in place of decision-making constructs. This is partly to reduce the complexity of the scenario description syntax, but more importantly to keep the scenario description as a portable top-level entity. Any specific or complex decision-making logic can easily be implemented as an XVC action, which in turn could indicate more events.

Event indications can be used to control the execution of a scenario. Users may specify a trigger event for any action within a scenario description. The predefined XVC manager will wait for that event to be indicated before causing an XVC to execute the corresponding action. For example, the XVC manager can instruct XVC “A” to execute actions A, B and C; then wait for event 1 to be indicated before executing action D.

Scenario events are defined as local to a test scenario or global to all scenarios. A scenario event is mapped to a \texttt{vmm\_notify::ONE\_SHOT} notification with the same numerical identifier in the predefined XVC manager notification service interface and in all XVC notification service interfaces. When a scenario event is indicated, the corresponding notification is indicated in the XVC manager and in all XVC instances.

XVC notifications can be mapped onto scenario events using the \texttt{MAPEVENT} command. When an XVC indicates a notification mapped to a scenario event, the indication will be propagated to the XVC manager and all other XVCs.

A combination of scenario events can be mapped onto a single scenario event using the \texttt{MAP} command.

The remainder of this section describes the syntax of the predefined XVC manager test scenario description language used to implement XVC tests. Table C-1. summarizes the convention used to describe the syntax of the various commands. All other textual elements as specified as-is.

File Structure
A file is composed of commands, comments and blank lines, each terminated by the newline or end-of-file character. A command may span multiple lines if the newline is escaped using a backslash (\) character.
Table C-1. Grammar Notation

<table>
<thead>
<tr>
<th>Simulation Handling</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>TOKEN</em></td>
<td>Case-insensitive, predefined token</td>
</tr>
<tr>
<td>&lt;token&gt;</td>
<td>Required user-defined token</td>
</tr>
<tr>
<td>[token]</td>
<td>Optional token</td>
</tr>
<tr>
<td>{token}</td>
<td>Optional token that can be specified 0 or more times</td>
</tr>
<tr>
<td>(tokens</td>
<td>tokens)</td>
</tr>
</tbody>
</table>

Lines are composed of tokens. Tokens are blank-separated strings. Tokens may contain environment variable substitutions. Blanks may be included in a token by quoting it using double quotes. A quoted string may contain a double-quote character by escaping it using a backslash.

The commands `VERBOSITY`, `STOPONERROR` and `STOPONEVENT` must appear, in any order or number, before any other commands.

The commands `ACTION` and `INTERRUPT` must appear after a `SCENARIO` command.

The `EXECUTE` commands must be the last commands.

All other commands can appear anywhere in the command sequence.

User-defined tokens, such as XVC names or action descriptions, are case-sensitive. All predefined tokens, such as `DISPLAY` or `WAIT`, are case insensitive.

**Example C-1. Typical Command Sequence**

```
VERBOSITY  
STOPONERROR  

LOG  

MAPEVENT  GLOBAL  
EVENT  GLOBAL  

SCENARIO  
EVENT  
ACTION  
ACTION  
```
vmm_xvc_manager

INTERRUPT ...

SCENARIO ...
  EVENT ...
  MAPEVENT ...
  ACTION

EXECUTE ...

Environment Variables
Any token can contain environment variable substitution.

Syntax:

  ... ${<env_name>}

where <env_name> is the name of an environment variable. It is an error if the variable is not set.

The content of the environment variable is included in the token. An environment variable containing blank characters is not interpreted as multiple tokens. A dollar sign ($) can be specified by escaping it using a double dollar sign ($$).

Commands

Comments
Specifies arbitrary text that is ignored by the XVC manager.

Syntax:

  // {<string>}

A comment is terminated by the next newline character. Any character following the “//” is ignored. Commands may be specified before the “//” characters.

Example:

  // This is a comment line, followed by a blank line
  ACTION xvc a "write 0x400 0x55AA" // Comment on action
#include
Includes a scenario definition file in the current scenario definition file.

Syntax:
   #INCLUDE <filename>

where <filename> is the name of a file that will be included as if its entire content had been specified instead of the #include command. Included files may include other files. If the filename is a relative path, the path is interpreted as relative to the location of the file containing the #include directive, not necessarily the current working directory.

It is illegal for an included file to contain X, QUITON or LOG commands or global notification definitions.

#define
Defines a symbolic name for a numeric identifier to aid the readability and maintainability of test command files.

Syntax:
   #define <symbol> (<nid>|<sid>|<sev>|<sid>.<sev>|<gev>)

where <symbol> is an alphanumeric string with no spaces or control characters.

<nid>, <sid>, <sev> and <gev> are unsigned integer values referring to a local XVC notification identifier, scenario identifier, scenario event or global event, respectively.

VERBOSITY
Sets the global message verbosity level.

Syntax:
   VERBOSITY (<instance>|ALL) <severity_level>

where <instance> is a string or regular expression specifying the instance names of the XVCs from which messages of the specified severity or higher should be displayed. Specifying ALL is identical to specifying /./.
<severity_level> specifies the minimum severity levels of messages to be displayed. Messages of a lower severity level will not be displayed. For further information, see the `vmm_log::set_verbosity()` method on page 374.

**LOG**

Writes messages issued from specified XVC instances into a user-specified file. This includes messages issued through the trace message service interfaces.

Syntax:

```
LOG (<instance>|ALL) <filename>
```

where `<instance>` is a string or regular expression specifying the instance names of the XVC from which messages should be logged to the specified file. Specifying `ALL` is identical to specifying `/./`. A single XVC instance can log messages to more than one file.

`<filename>` is the name of the file to which the messages will be written. If the name is prefixed with a `+' character, the messages are appended to the file. If the name is prefixed with a '-', the messages are no longer written to the specified file.

**Examples:**

```
LOG ALL "Messages.log"
LOG UART "+UART.log"
... 
LOG UART "-UART.log"
```

**LOG NONE**

Stops writing messages to files and closes all log files. Applies to all XVC instances and all files. This action includes messages issued through the trace message service interfaces.

Syntax:

```
LOG NONE
```
XVC Standard Library Specification

**XVCTRACE**
Writes messages issued from specified XVC instances via their `xvc_xactor::trace` message service interface or the XVC manager via its `xvc_manager::trace` message service interface into a user-specified file.

Syntax:

```
XVCTRACE (<instance>|MANAGER|ALL) <filename>
XVCTRACE NONE
```

where `<instance>` is a string or regular expression specifying the instance names of the XVC from which trace messages should be logged to the specified file. Specifying `ALL` is identical to specifying `/./`. `MANAGER` specifies the XVC manager itself. If `NONE` is specified, trace messages are no longer logged to a file.

**COVFILE**
Specifies the name of the functional coverage database file.

Syntax:

```
COVFILE (<dbname>|NONE)
```

where `<filename>` is the name of the database to which coverage information will be written. If `NONE` is specified, no functional coverage is to be collected during simulation.

**STOPONERROR**
Stops the simulation after `<count>` error messages have been issued. By default, the simulation stops after issuing 10 messages with a severity of `ERROR_SEV`. The simulation always stops when a message with a `FATAL_SEV` severity level is issued.

Syntax:

```
STOPONERROR <count>
```
STOPONEVENT

Stops the simulation if the specified circumstance occurs before the natural end of the
test.

Syntax:

\[
\text{STOPONEVENT } [<sid>.]<sev> <\text{(IMMEDIATE|GRACEFUL)}> [\text{<count>}] \\
\]

where \([<sid>.]<sev>\) specifies a a global or local event.

If IMMEDIATE is specified, the simulation is stopped as soon as the specified event is
indicated. If GRACEFUL is specified, the XVC manager will delay the end of the
simulation until all actions executing in the current scenario have completed.

\text{<count>} is the number of times the specified events must be indicated before the
simulation is stopped.

SCENARIO

Defines a new test scenario and implicitly terminates the definition of a previous test
scenario.

Syntax:

\[
\text{SCENARIO } <sid> [\text{<description>}] \\
\]

where \(<sid>\) is a unique unsigned integer identification number for the scenario to
be defined. Two scenarios cannot have the same identification number in the same
simulation.

\text{<description>} is an arbitrary description string that will be displayed to the
simulation output and log file(s) whenever that scenario is executed.

This command starts the definition of a new scenario. Any subsequent \text{EVENT,}
\text{MAPEVENT, ACTION or INTERRUPT} commands will define this scenario. A
subsequent \text{SCENARIO} or \text{EXECUTE} command terminates the scenario.

Actions may not execute in the sequence they are specified in the scenario. Actions
targeted to different XVCs will execute concurrently, as soon as the target XVC can
execute the next action. Only actions targeted to the same XVC will execute in the
sequence specified in the scenario.

It is an error to define a scenario that does not contain at least one action.
**EVENT**
Defines a local or global event. It allows actions of different XVCs to be coordinated within a scenario. Only global events can be defined outside of a scenario.

Syntax:

```
EVENT [ONESHOT] [(LOCAL|GLOBAL)] (<sev>|<gev>) IS
  [<sid>.,]<sev>{(,|+)[sid.]<sev>} [<descr>]
```

where `<sev>` or `<gev>` is a unique unsigned identification number of the local or global event to be defined. Global events must have globally-unique identifiers. Scenario events must have scenario-unique identifiers. A scenario event with the same identifier as a global event will hide the global event within that scenario.

If `ONESHOT` is specified, the event will be indicated only once during the entire test, even though the criteria for notification may occur multiple times. By default, the event is notified each time the criteria occurs.

If `LOCAL` is specified, the event is local to the current scenario. If `GLOBAL` is specified, the event is global to all scenarios. By default, events are local. Global events can be defined either outside or inside a scenario.

The event is notified when the specified notifications have been observed to be notified. `<sev>{+<sev>}` specifies that the defined event is indicated when all of the specified events are indicated, in any order. `<sev>{,<sev>}` specifies that the defined event is indicated when any of the specified events are indicated. The `+` operator has precedence over the `,` operator. When defining global events, events local to scenarios can be referred to by prefixing them with the appropriate scenario identifier using the `<sid>` notation.

`<descr>` is an optional string message, which will be displayed in a `DEBUG_TYP, TRACE_SEV` trace message if and when the notification is indicated. If no description is specified, a default description is used.

Example:
```
EVENT 3 IS 2 + 1
```

Declares local scenario event 3 that will be indicated following the occurrence of the events 2 and 1.

```
E GLOBAL 6 IS 2,3.1 "Setup complete"
```
Declares a global event 6 that will be indicated, with the message “Setup complete”
displayed, as soon as either global event 2 or the local event 1 in scenario 3 have been
indicated.

MAPEVENT
Maps a local XVC event to scenario or global event. It allows the actions of different
XVCs to be coordinated within a scenario.

Syntax:

```
M[APEVENT] [ONESHOT] [(LOCAL|GLOBAL)] (<sev>|<gev>) IS \n  <xid> E[VENT] <nid> [<descr>]
```

where `<sev>` or `<gev>` is a unique unsigned identification number of the local or
global event to be defined. Global events must have globally-unique identifiers.
Scenario events must have scenario-unique identifiers. A scenario event with the
same identifier as a global event will hide the global event within that scenario.

If `ONESHOT` is specified, the event will be indicated only once during the entire test,
even though the criteria for notification may occur multiple times. By default, the
event is notified each time the criteria occurs.

If `LOCAL` is specified, the event is local to the current scenario. If `GLOBAL` is
specified, the event is global to all scenarios. By default, events are local. Global
events can be defined either outside or inside a scenario.

`<xid>` is a string or regular expression specifying the instance name of the XVCs
that is the source or cause of this event.

`<nid>` identifies the local event using the notification identifier in the
`xvc_xactor::notify` notification service interface that, when indicated, will
indicate this event.

`<descr>` is an optional string message, which will be displayed in a `DEBUG_TYP,
TRACE_SEV` trace message if and when the event is indicated. If no description is
specified, a default description is used.

Example:

```
Mapevent 3 is "CLCD" event 1
```

Defines a local event 3 that will be indicated whenever the XVC instance named
“CLCD” indicates its local notification 1.
MAPEVENT GLOBAL 5 is /^AHB/ event 5 "Abort"

Defines a global event 5 that will be indicated whenever any one of the XVCs with an instance name matching the regular expression indicates its local notification 5.

**ACTION**

Adds an action execution to the scenario definition.

Syntax:

```
ACTION <instance> <action> \n    [WAIT <wait_for>] [EMIT <notification>]
```

where `<instance>` is the instance name of the XVC for which the action is intended. Regular expressions cannot be used because actions cannot be targeted to multiple XVC instances.

`<action>` is the string that defines the action to be executed. The syntax of this token is action-specific and is defined in the `xvc_action::parse()` method of the relevant action descriptor.

The `<wait_for>` and `<notification>` tokens are described in the `WAIT` and `EMIT` options sections.

This command adds an action to the action list of the specified XVC instance. The `WAIT` option lets the start of the action be synchronized with the indication of one or more specified scenarios or global events(s). The `EVENT` options lets the completion of that action indicate the specified local notification.

Examples:

```
ACTION ARM_DMA "5 True enable 5 9" WAIT 1
```

The XVC instance named `ARM_DMA` waits for the next indication of the scenario or global event ‘1’, then it will execute the specified action.

```
ACTION uart "enable master mode" EMIT 4
```

The XVC instance named `uart` will execute the specified action. When the action completes, the local XVC notification ‘4’ will be indicated.
**INTERRUPT**
Add a high-priority action execution to the scenario definition.

Syntax:

```
I[NTERRUPT] [ONESHOT] <instance> <action> \\
[WAIT <wait_for>] [EMIT <notification>]
```

where `<instance>` is the instance name of the XVC for which the interrupt action is intended. Regular expressions cannot be used because actions cannot be targeted to multiple XVC instances.

 `<action>` is the string that defines the interrupt action to be executed. The syntax of this token is action-specific and is defined in the `xvc_action::parse()` method of the relevant action descriptor.

If `ONESHOT` is specified, the interrupt action will only execute one on the first occurrence of the specified event. Subsequent occurrences of the event will not cause the interrupt action to re-execute.

The `<wait_for>` and `<notification>` tokens are described in the `WAIT` and `EMIT` options sections.

Schedule the specified action for execution whenever the specified `WAIT` condition is observed. If no `WAIT` condition is specified, the interrupt action is scheduled once immediately. The interrupt action will interrupt the execution of any action that is currently executing by the target XVC instance. The granularity of the interruption is defined by the implementation of the to-be-interrupted action in the `xvc_action::execute()` method by invoking the `xvc_xactor::wait_if_interrupted()` method. Interrupt actions cannot be interrupted.

If two interrupt actions are scheduled to execute on the same XVC instance at the same time, they will be executed in a non-deterministic order.

Examples:

```
INTERRUPT ARM_DMA "Reconfigure Generation" WAIT 1
```

The XVC instance named `ARM_DMA` will execute the specified action whenever global or scenario event ‘1’ is indicated, at the earliest possible action interruption point.
The XVC instance named ARM_DMA will immediately execute the specified interrupt action at the next interruption opportunity. When the action completes, the local XVC notification ‘4’ will be indicated.

... WAIT <wait_for>
This is an option for the ACTION and INTERRUPT command, not a stand-alone command. It is used to specify a global or scenario event indication for which the XVC will wait before starting the action.

Syntax:

... W[AIT] <sev>|<gev>

where <sev> or <gev> is the identifier for a scenario or global event.

The ACTION or INTERRUPT command will be delayed until the specified event has been indicated.

Example:

ACTION ... W 1

... EMIT <indication>
This is an option for the ACTION and INTERRUPT command, not a stand-alone command. It is used to specify a xvc_xactor::notify notification that is indicated when the XVC completes the execution of an action.

Syntax:

... E[MIT] <nid>

where <nid> is the identifier for a notification in the xvc_xactor::notify notification service interface of the XVC executing the action.

Example:

ACTION ... EVENT 1

When the execution of the action completes, notifications ‘1’ will be indicated, potentially allowing other XVC actions to be executed if they are mapped to global or scenario events.
EXECUTE
Executes the specified scenarios, in sequence.

Syntax:

```
[EX][ECUTE] <sid>{ <sid>}
```

where `<sid>` is the unique identifier of a scenario to execute.

**Example C-2. Example Test Scenario Description File**

```
// Test Set-up
//---------------------------------------------------
VERBOSITY ALL TRACE
LOG ALL "Trace.txt"
COVFILE NONE
STOPONERROR 1

// Global events
MAPEVENT GLOBAL 1 IS a EVENT 1

// Scenario definitions
//---------------------------------------------------
scenario 1 "Demonstrating XVCs"

    action a "Action 1"
    action b "Action 1"

scenario 2 "My second scenario - action end events"

    action a "Action 2" E 1
    action b "Action 2" W 1

scenario 3 "My third scenario - interrupt actions"

    action xvc a "Action 1" E 1
    interrupt xvc b "Action 1" W 1

// Execution
//---------------------------------------------------
x 1 2
x 3
```
APPENDIX D SOFTWARE TEST FRAMEWORK

This appendix specifies the detailed behavior of a C library that can be used to implement a VMM-compliant software verification environment and tests. The actual implementation of these classes is left to each tool or platform provider.

BASIC TYPES

The following are definitions of standard types, with explicit bit widths for clarity:

```c
typedef unsigned int        BOOL;
typedef signed char        BYTE8;
typedef unsigned char       UBYTE8;
typedef signed short       HWD16;
typedef unsigned short      UHWD16;
typedef signed int         WORD32;
typedef unsigned int       UWORD32;
typedef signed long long   LLONG64;
typedef unsigned long long ULLONG64;
```

A word is defined to be 32-bits long.
SYSTEM DESCRIPTOR

The system descriptor is an array of peripheral device descriptors.

```c
const svSYS_SystemElement svSYS_SystemDescriptor[] = {...}
```

A symbol must exist that specifies the index of a peripheral descriptor in the system descriptor. The symbol must be named `svSYS_Element_XXXX_N` where `XXXX` is the name of the peripheral and `N` is the instance number.

Peripheral Descriptor

A `svSYS_SystemElement` is a descriptor for an instance of a peripheral device. Each peripheral instance is described using the following structure:

```c
typedef const struct SystemElement svSYS_SystemElement;
struct SystemElement {
    UWORD32 Tested;
    svSYS_SystemID DeviceID;
    UWORD32 BaseAddress;
    svSYS_SystemInterrupts Interrupts;
    svSYS_SystemClocks Clocks;
    svSYS_SystemDMA DMA;
    svSYS_ActionSheetItem * pActionSheet;
    svSYS_CheckState pCheckState;
    UWORD32 Padding[x];
};
```
The following is an example of a fully specified peripheral descriptor in a system descriptor:

```c
const svSYS_SystemElement svSYS_SystemDescriptor[] = {
    ...
    // Data for P123 UART instance #0
    {
        0x1,       // Selected for testing
        "P123",     // Device ID "P123"
        0x20000000, // Base Address
        {{{svSYS_GET_SYS_DATA(P123, 0), 12}},
        // One interrupt at
        // controller 0, source 12
        {40000000},   // One clock at 4MHz
        {{{&svSYS_GET_SYS_DATA(P456, 0), 1, 4}},,
        // One DMA at
        // controller 0, channel 4
        // accessible via port 0
        ActionSheet_P456, // Address of action sheet
        svP456_CheckState  // State checking function
    },
    ...
}

typedef enum {
    ...
    svSYS_Element_P123_0,
    ...
    svSYS_Element_P456_0,
    ...
}
```

**Tested**

This structure property identifies peripheral sets this peripheral belongs to. A peripheral belongs to peripheral set $N$ if bit $N$ of this structure property is set. A peripheral can thus belong to more than one peripheral set.

**DeviceID**

This structure property specifies the name of the peripheral, for example “P123”. It is a null-terminated string padded with zeros if it is less than seven characters.

```c
typedef unsigned char svSYS_SystemID[8];
```

**BaseAddress**

This structure property specifies the hardware base address of the peripheral.
Interrupts
This structure property specifies the interrupts generated by the peripheral. A peripheral can generate interrupts to up to eight different interrupt controllers.

```c
typedef svSYS_SystemInterrupts svSYS_SystemInterrupts[8];
```

Each entry in the array describes a single interrupt connection from the device to a controller. See “Interrupt Descriptor” on page 463 for a description of the interrupt descriptor.

Clocks
This structure defines the frequency of the clocks signals supplied to the peripheral. Each element of the array is a clock rate in Hz. If a clock signal is not used, the frequency value is specified as 0.

```c
typedef UWORD32 svSYS_SystemClocks[4];
```

DMA
This structure property describes up to four DMA channels connected to the peripheral.

```c
typedef SYS_SystemDMA svSYS_SystemDMA[4];
```

Each entry in the array describes a single DMA connection from the device to a controller. See “DMA Channel Descriptor” on page 464 for a description of the DMA channel descriptor.

pActionSheet
This structure property is a pointer to zero-terminated array of actions called an action sheet. These actions—and only these actions—are available for execution on this peripheral. Multiple instances of the same peripheral may refer to the same action sheet.

```c
const svSYS_ActionSheetItem ActionSheet_P123 [] = {
    {&MyP123FirstAction, &MyP123FirstAction_Level, 
    &MyP123FirstAction_Name},
    {&MyP123SecondAction, &MyP123SecondAction_Level, 
    &MyP123SecondAction_Name},
    ...
    {0} //Must be zero terminated
};
```
System Descriptor

See “svSYS_ActionSheetItem” on page 467 for a specification of the action sheet entry.

**pCheckState**
This structure property is a pointer to a function that checks if the peripheral is in an idle state.

```c
typedef svSYS_eTestResponse(* svSYS_CheckState)(svSYS_SystemElement * pPeriph)
```

The function returns `svTestPassed` if the peripheral is currently in an idle state and `svTestFailed` otherwise. The meaning of *idle* is specific to each peripheral but in general, *idle* will mean that the peripheral is disabled and has no interrupts flagged.

See “svSYS_eTestResponse” on page 466 for the specification of `svSYS_eTestResponse`.

**Padding**
This structure property pads the value of the structure to a power-of-two number of words if needed. It will allow rapid iteration through the array of peripheral descriptors.

**Interrupt Descriptor**
This structure describes a single interrupt connection from a peripheral to an interrupt controller. Peripheral interrupts are described by the `Interrupts` struct property of that peripheral’s descriptor, as specified in “Peripheral Descriptor” on page 460.

```c
typedef struct {
    svSYS_SystemElement  * pController;
    UWORD32                Source;
} svSYS_SystemInterrupts;
```

**pController**
This structure property is a pointer to the peripheral descriptor for the interrupt controller. If the value is `NULL`, there is no interrupt described and this descriptor is to be ignored.
Source
The interrupt number, in the interrupt controller, of the interrupt that is generated by the peripheral.

DMA Channel Descriptor
This structure describes a single DMA channel connection between a peripheral and a DMA controller. The DMA channels for a peripherals are specified by the DMA struct property in that peripheral's descriptor. See “Peripheral Descriptor” on page 460 for more details.

typedef struct {
    svSYS_SystemElement  * pController;
    UHWD16                 Masters;
    UHWD16                 Channel;
} svSYS_SystemDMA;

pController
This structure property is a pointer to the peripheral descriptor for the DMA controller. If the value is NULL, there is no DMA channel described and the descriptor must be ignored.

Masters
This structure property indicates which of the master ports of the DMA controller can access the DMA channel. Master ports are identified by their corresponding bit position. For example, a value of 0x5 specified that the DMA channel to the peripheral is accessible from master ports 0 and 2.

Channel
The DMA channel number, in the DMA controller, of the DMA channel to the peripheral.
TEST ACTIONS

The following macros, declarations and functions are available to support the specification and execution of test actions on specific peripherals.

svTEST_NAME()
This macro is used to specify an arbitrary string to describe a test action. The string is accessible as an externally visible character array named <actionname>_Name. It is used to identify the running test action in simulation messages and the debug channel.

```c
#define svTEST_NAME(action, descr) const char action##_Name[] = descr;
```

The action parameter is the name of the function implementing the test action. The descr parameter is a string literal which describes or identifier the test action. For example:

```c
svTEST_NAME(svP123_FirstAction, "Test Interrupts on P123")
```

svTEST_LEVEL()
This macro is used to specify a complexity-level value to a test action. The value is accessible as an externally visible unsigned integer named <actionname>_Level.

```c
#define svTEST_LEVEL(action, level) \
 const unsigned int action##_Level = level;
```

The action parameter is the name of the function implementing the test action. The level parameter is an integer value specifying the complexity level of the test action. For example:

```c
svTEST_LEVEL(svP123_FirstAction, 3)
```

svSYS_GET_SYS_DATA()
This macro returns the address of the peripheral descriptor corresponding to the specified instance of the specified peripheral. The macro uses the specified instance and peripheral name to construct the symbolic name of the index of the peripheral descriptor in the system descriptor.

```c
#define svSYS_GET_SYS_DATA(Device, DeviceNum) (svSYS_SystemDescriptor + svSYS_Element_##Device##_##DeviceNum)
```
Software Test Framework

**svSYS_eTestResponse**
The *svSYS_eTestResponse* type is used to specify the result for a test action, as described in Table D-1.

```c
typedef enum {
    svSYS_TestSkipped    = -1,
    svSYS_TestFailed     = 0,
    svSYS_TestPassed     = 1,
    svSYS_TestException  = 2,
    svSYS_TestAvailable  = 3
} svSYS_eTestResponse;
```

**Table D-1 Test Status Codes**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svSYS_TestSkipped</td>
<td>Test not performed</td>
</tr>
<tr>
<td>svSYS_TestFailed</td>
<td>Test failed</td>
</tr>
<tr>
<td>svSYS_TestPassed</td>
<td>Test passed</td>
</tr>
<tr>
<td>svSYS_TestException</td>
<td>Test generated an exception</td>
</tr>
<tr>
<td>svSYS_TestAvailable</td>
<td>Test marked as available</td>
</tr>
</tbody>
</table>

**svSYS_SeqTest**
This pointer-to-function type is a prototype for test action functions.

```c
typedef svSYS_eTestResponse (*svSYS_SeqTest)(svSYS_SystemElement * pPeriph);
```

The *Periph* parameter is a pointer to the system descriptor for the peripheral targeted by this test action. The action is responsible for retrieving and using the fields of interest.

**svSYS_ActionRun()**
Runs the specified test action on the specified peripheral.

```c
void SYS_ActionRun (  
    char                * pTestName,  
    char                * pDeviceName,  
    svSYS_SeqTest       Action,  
    svSYS_SystemElement * pPeriph  
);
```

The *pTestName* parameter is a pointer to the description or name of the test action to be executed. The *pDeviceName* parameter is a pointer to the descriptor or name
of the target peripheral. The *Action parameter is a pointer to the test action function to be executed. The *Periph parameter is a pointer to the peripheral descriptor of the target peripheral.

This function standardizes the behavior and output of actions and performs common pre- and post-action operations:

- Outputs a debug message containing *pTestName, *pDeviceName and *Periph->BaseAddress
- Configures the exception handler to use a test-specific routine
- Runs the *Periph->pCheckState() function to check that the peripheral is in an idle state and outputs a debug message describing the result
- Calls the test action, passing in the *Periph pointer
- Restores the original exception handler
- Outputs a debug message describing the test result
- Adds a new entry to the test summary report

**svSYS_ActionSheetItem**

This structure defines a test action that is part of a test action sheet. A test action sheet is specified as a zero-terminated array of *svSYS_ActionSheetItem. See “pActionSheet” on page 462.

```c
typedef struct {
    svSYS_SeqTest    TestFunction;
    WORD32    * pLevel;
    char    * pName;
} svSYS_ActionSheetItem;
```

The *TestFunction structure property is a pointer to the function implementing the test action. The *pLevel structure property is a pointer to the test complexity-level value for the test action. The *pName structure property is a pointer to the description or name of the test action.
Software Test Framework

svSYS_ActionSheetRun()

This function is used to execute the test actions found in a specific test action sheet.

```c
void SYS_ActionSheetRun(
    char                * pDeviceName,
    UWORD32               Sequence,
    svSYS_eTestOrder      TestOrder,
    svSYS_SystemElement * pPeriph
);
```

The `pDeviceName` parameter is a pointer to the descriptor or name of the target peripheral. The `Sequence` parameter specifies the complexity levels that are to be executed. The `TestOrder` parameter specifies the order in which to execute the test actions found in the action sheet. The `Periph` parameter is a pointer to the peripheral descriptor of the target peripheral.

This function will iterate through all test actions found in the action sheet `pPeriph->ActionSheet` in the order specified by the `TestOrder` parameter. If the bit in the `Sequence` value corresponding to the complexity level of the test is set, the test is executed. Otherwise, the test is skipped and a new entry is added to the test summary report.

svSYS_eTestOrder

This enumerated type is used to specify the order of execution of tests in an action sheet.

```c
typedef enum {
    svSYS_TestSequence,
    svSYS_TestReverse,
    svSYS_TestRandom
} svSYS_eTestOrder;
```

The execution order specified by each value is described in the following table:
Test Actions

Table D-2  Test Sequencing Codes

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>svSYS_TestSequence</td>
<td>Execute tests in order specified in the action sheet</td>
</tr>
<tr>
<td>svSYS_TestReverse</td>
<td>Execute tests in the reverse order specified in the action sheet</td>
</tr>
<tr>
<td>svSYS_TestRandom</td>
<td>Execute tests in random order</td>
</tr>
</tbody>
</table>

svSYS_AllActionSheetsRun()

This function iterates through all peripherals in the system descriptor in a specified order and executes all tests in the test action sheet for each peripheral.

```c
void svSYS_AllActionSheetsRun(
    UWORD32 Peripherals,
    UWORD32 Sequence,
    svSYS_eTestOrder PeriphTestOrder,
    svSYS_eTestOrder ActionTestOrder);
```

The `Peripherals` parameter is specifies which peripherals should be tested in the run. The `Sequence` parameter specifies the complexity levels that are to be executed. The `PeriphTestOrder` parameter specifies the order in which to test the peripherals found in the system descriptor. The `ActionTestOrder` parameter specifies the order in which to execute the test actions found in the action sheet.

Peripherals are selected by and-ing the value of the peripherals parameter to the peripheral's tested value. If the result is non-zero, the action sheet associated with the peripheral is run. If the result is zero, the peripheral is skipped. This mechanism creates 32 different sets of peripherals that can be included or excluded from a test run. If a peripheral is selected, its action is sheet is run.

```c
UWORD32 svSYS_Peripherals;
UWORD32 svSYS_Sequence;
```

These global variables may be polled by the software test framework after every execution of `svSYS_AllActionSheetsRun()`. If the value of either variable has changed since the last poll, the `svSYS_AllActionSheetsRun()` is invoked again. If either is zero or the values have not changed, the polling continues until a change is detected.
These variables lets different peripherals or tests be dynamically selected using a debugger. If the execution of the software test is stopped and new values inserted, a new set of tests will be run after the completion of the current test run.

**LOW-LEVEL SERVICES**

**svSYS_DebugLevel**

Global variable used to specify the current debug level for the software test. A value of `svSYS_FATAL_SEV` is invalid and interpreted as `svSYS_ERROR_SEV`. The value of this variable can be modified at run time or via a debugger.

```c
svSYS_eMsgLevel svSYS_DebugLevel = svSYS_NORMAL_SEV;

typedef enum {
    svSYS_FATAL_SEV,
    svSYS_ERROR_SEV,
    svSYS_WARNING_SEV,
    svSYS_NORMAL_SEV,
    svSYS_TRACE_SEV,
    svSYS_DEBUG_SEV,
    svSYS_VERBOSE_SEV
} svSYS_eMsgLevel;
```

**svSYS_Printf()**

This function is used to issue all messages from a software test.

```c
void svSYS_Printf(svSYS_eMsgLevel     Level,
                 svSYS_ePrintType    Type,
                 void              * pParam);
```

This function will print the specified message to the defined output channel if the `Level` parameter is less than or equal to the current verbosity level defined by the `svSYS_DebugLevel` variable.

The `Type` parameter specifies the format of the data pointed to by the `pParam` parameter, using one of the following enumerated values:

```c
typedef enum {
    svSYS_PrintString,
    svSYS_PrintHex,
    svSYS_PrintDec,
    svSYS_PrintBoolean
} svSYS_ePrintType;
```
Low-Level Services

svSYS_EnvironmentGet()
This function identifies the software verification environment used to execute the software tests.

\[
\text{svSYS\_eEnvironment svSYS\_EnvironmentGet(void);}
\]

The value returned by this function is one of the following enumerated values:

<table>
<thead>
<tr>
<th>\text{svSYS_EnvSimulator}</th>
<th>Simulation-based environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{svSYS_EnvFPGA}</td>
<td>FPGA-based environment</td>
</tr>
</tbody>
</table>

svSYS_Ignore()
This function verifies if the current verification environment matches one of the specified unsupported verification environments. It is used to skip tests that cannot be executed on specific verification environments.

\[
\text{BOOL svSYS\_Ignore(UWORD32 TestMask);} \]

The \text{TestMask} parameter is the bitwise-OR of the enumerated values corresponding to the unsupported environments. If the current environment matches one of the specified unsupported environments, a \text{svSYS\_NORMAL\_SEV “Test Skipped”} message is issued and \text{TRUE} is returned. Otherwise, \text{FALSE} is return. The test action code is responsible for aborting the test if it is to be ignored.

\[
\text{if (svSYS\_Ignore(svSYS\_EnvFPGA | svSYS\_EnvPV)) return;}
\]

svSYS_HARDCODED()
This macro has no effect on the code but identifies potential porting issues.

\[
\text{#define svSYS\_HARDCODED(code) code}
\]

svSYS_Malloc()
svSYS_Free()
The software framework does not implement a full heap, due to the overheads associated with managing system resources. Instead, it offers the following functions:

\[
\text{void * svSYS\_Malloc(unsigned int size);}
\]
\[
\text{void svSYS\_Free(void *);} \]
There must be a block of data for dynamic memory allocation, which is identified in a linker definition file as execution region heap. The heap would be located from a heap base address constant to a heap limit-1.

Where memory leakage is not an issue, a simple heap management process can be used. The heap manager maintains a pointer to the top of the current heap. When `svSYS_Malloc()` is called, the heap is checked for the requested space, then the current pointer is returned and the pointer is incremented by the requested space size. Freed dynamic memory is never reclaimed and the `svSYS_Free()` function may be implemented as a blank macro.

In platforms and environments where memory leak is an issue, these functions may implement a more complex dynamic memory management system, including garbage collection.

```
svIO_BYTE_READ()
svIO_BYTE_WRITE()
svIO_WORD_READ()
svIO_WORD_WRITE()
```

The software framework provides macros for accessing memory-mapped registers, given the base address and an offset in bytes.

For word-width (32-bit) registers, the following macros are available:

```c
typedef volatile UWORD32 svRegister;

#define svIO_WORD_READ(_Addr, _Offset) \ 
    *((svRegister*) ((UWORD32)_Addr + \ 
    (UWORD32)_Offset))

#define svIO_WORD_WRITE(_Addr, _Offset, _Value) \ 
    *((svRegister*) ((UWORD32)_Addr + \ 
    (UWORD32)_Offset)) = \ 
    (UWORD32)_Value;
```
For byte-width registers the following are available:

```c
typedef volatile UBYTE8 svRegisterByteAccess;

#define svIO_BYTE_READ(_Addr, _Offset)     
    *((svRegisterByteAccess*) ((UWORD32)_Addr + 
                                   (UWORD32)_Offset))

#define svIO_BYTE_WRITE(_Addr, _Offset, _Value) 
    if(_Value != (_Value & 0xFF)) SYS_swi();\ 
    *((svRegisterByteAccess*) \ 
      ((UWORD32)_Addr + (UWORD32)_Offset)) = \ 
      (UBYTE8)_Value;
```

**svSYS_RAND()**

This function will generate and return a pseudo-random number. It is used by the `svSYS_AllActionSheetsRun()` function for sequencing tests in random order. It is also available for use within test actions to create random tests.

```c
UWORD32 svSYS_RAND(UWORD32 Limit)
```

The returned value will be between 0 and `Limit`-1. It is recommended that `Limit` be a power of 2 for runtime efficiency. The implementation of this function may invoke the standard C `rand()` function. The seed is set during initialization by the bootstrap module.

**svSYS_BlockCopy()**

This function copies a block of 32-bit words from one location to another.

```c
void svSYS_BlockCopy(UWORD32 * pSourceAddr, 
                     UWORD32 * pDestAddr, 
                     UWORD32 NumWords);
```

This function may be implemented using processor instructions or DMA channels.

**svSYS_ThrowException()**

This function is used to throw a software interrupt (SWI) exception.

```c
svSYS_ThrowException()
```

This function hides the system-specific way of generating a processor exception and catches this exception.
svSYS_ASSERT()

This macro is used to verify that a condition holds true and throws an SWI exception otherwise.

    svSYS_ASSERT(BOOL Assertion)

If the value of the Assertion parameter is FALSE, an SWI exception is thrown.

Cache Lockdown

Some processors have the ability to lock down and free either or both the instruction and data cache. This ability ensures that there will be no instruction or data bus activity while a cache is locked down.

Cache lockdown is implemented using the following functions:

svSYS_CacheLock()

This function locks all addresses between the start and end addresses into cache. The start address is included. The end address is normally excluded but may be locked down if cache line size requires it.

typedef UWORD32 svSYS_CacheLines;
typedef enum {
    svSYS/InstructionCache,
    svSYS/DataCache
} svCacheType;
svSYS_CacheLines svSYS_CacheLock(
    svSYS_CacheType eCache,
    void * pStartAddress,
    void * pEndAddress);

The return value that describes the block of locked cache lines is specified Table D-4.

If there is insufficient cache space to lock the requested address range, the maximum available number of cache lines should be locked, leaving at least one free cache line. It is recommended that a debug message will be generated to indicate this occurrence.

It is assumed that cache locking and unlocking will remain the responsibility of the user.
### svSYS_CacheUnlock()

This function unlocks a set of cache lines.

```c
void svSYS_CacheUnlock(svCacheLines LockedLines);
```

The descriptor for the cache lines to unlock, specified by the `LockedLines` parameter, is identical to the descriptor returned by the `svSYS_CacheLock()` function.

### svSYS_CACHE_BLOCK_START() and svSYS_CACHE_BLOCK_END()

Macros used to name a block of instructions that can be locked in the instruction cache.

```c
#define svSYS_CACHE_BLOCK_END(name) \
    void name##_CacheBlockEnd(void) {}
#define svSYS_CACHE_BLOCK_END(name) \
    void name##_CacheBlockEnd(void) {}
```

These macros create symbols that can be used to specify start and end addresses to the `svSYS_CacheLock()` function.

### Interrupt Controller

Software test actions can configure interrupt controllers to verify the interrupt sourcing features of a peripheral. Note that this is not done to test the interrupt controller, where controller-specific actions will be needed, but to test a peripheral generating an interrupt to the interrupt controller.

The normal process of using interrupts is to bind an interrupt handler to an interrupt. An interrupt handler is a software routine that is executed when the interrupt occurs.
The following example illustrates how to use some of the interrupt support routines to enable interrupts before executing the test and disables them afterwards:

```c
void MyTestHandler(UWORD32 Source,
                    svSYS_SystemElement * pPeriph)
{
    ... // Interrupt handler during test action */
}

svSYS_eTestResponse
P123_InterruptTest(svSYS_SystemElement * pPeriph)
{
    svSYS_InterruptAllBind(&MyTestHandler, pPeriph);
    ... // Test action using interrupts
    svSYS_InterruptAllUnBind(&MyTestHandler, pPeriph);
}
```

The following functions are available to configure interrupt controllers:

### svSYS_InterruptInit()

Initializes the specified interrupt controller.

```c
void svSYS_InterruptInit(
    svSYS_SystemElement  * pController,
    BOOL                    Enabled);
```

The `pController` parameter specifies the interrupt controller to be initialized. This function must be invoked before tests that use the interrupt controller.

If the `Enabled` parameter is `FALSE`, all interrupts will be disabled at the interrupt handler. If `TRUE`, all interrupts not already asserted will be enabled. After initialization, the handler for all interrupt sources is set to the following routine:

```c
void SYS_BadInterrupt(UWORD32 Source);
```

This routine will generate an exception, indicating that an unexpected interrupt has occurred.

### svSYS_InterruptHandler

The interrupt handler routine must comply with the following prototype:

```c
typedef __irq void (*svSYS_InterruptHandler)(
    UWORD32 Source,
    svSYS_SystemElement  * pPeriph);
```
The *Source* parameter is the number of the interrupt, in the controller, that is being handled. The *pPeriph* parameter specifies the peripheral descriptor for the peripheral generating the interrupt.

**svSYS_InterruptBind()**
This function binds a handler to an individual interrupt and enables the interrupt.

```c
void svSYS_InterruptBind(
    svSYS_InterruptHandler   Handler,
    svSYS_SystemElement    * pPeriph,
    UWORD32                  IntIndex,
    svSYS_eInterruptType     IntType,
    UBYTE8                    priority);
```

The *Handler* parameter specifies the interrupt handler routine to be run when the interrupt occurs. The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupt. The *IntIndex* parameter specifies the interrupt number in the peripheral's list of interrupts. The *IntType* parameter specifies the type of interrupt (either `svSYS_IRQ` or `svSYS_FIQ`). And the *Priority* parameter specifies the priority level of the interrupt on a scale 0 (low) to 15 (high).

**svSYS_InterruptUnBind()**
This function unbinds an interrupt handler routine from an individual interrupt, restoring the unexpected interrupt handler and leaving the enable status unchanged.

```c
void svSYS_InterruptUnBind(
    svSYS_SystemElement *  pPeriph,
    UWORD32                IntIndex);
```

The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupt. The *IntIndex* parameter specifies the interrupt number in the peripheral's list of interrupts.

**svSYS_InterruptAllBind()**
**svSYS_InterruptAllUnBind()**
These functions will bind all the interrupt sources specified for a peripheral onto a specified interrupt handler routine and enable the associated interrupts, or unbind these interrupts.

```c
void svSYS_InterruptAllBind(
    svSYS_InterruptHandler   Handler,
    svSYS_SystemElement *   pPeriph);

void svSYS_InterruptAllUnBind(
    svSYS_SystemElement * pPeriph);
```
Software Test Framework

The Handler parameter specifies the interrupt handler routine to be run when any of the interrupts occur. The pPeriph parameter specifies the peripheral descriptor of the peripheral generating the interrupts.

svSYS_InterruptEnable()
svSYS_InterruptDisable()

These functions enable or disable an individual interrupt.

```c
void svSYS_InterruptEnable(
    svSYS_SystemElement *  pPeriph,
    UWORD32                IntIndex);
void svSYS_InterruptDisable(
    svSYS_SystemElement *  pPeriph,
    UWORD32                 IntIndex);
```

The pPeriph parameter specifies the peripheral descriptor of the peripheral generating the interrupt. The IntIndex parameter specifies the interrupt number in the peripheral's list of interrupts.

svSYS_InterruptAllEnable()
svSYS_InterruptAllDisable()

These functions will enable or disable all the interrupt sources specified for a peripheral.

```c
void svSYS_InterruptAllEnable(
    svSYS_SystemElement *  pPeriph);
void svSYS_InterruptAllDisable(
    svSYS_SystemElement *  pPeriph);
```

The pPeriph parameter specifies the peripheral descriptor of the peripheral generating the interrupt.

Software-XVC Connectivity

For some test scenarios, it may be required to synchronize the execution of software routines with specific hardware events. For example, it may be necessary to observe a software stack in response to data packets being driven into a system peripheral. The software framework, rather than driving the entire system test, can act as a slave waiting for the XVC manager to send commands via a software-interface XVC.

Two global variables are used to hold a pair of base addresses through which data from a software-interface XVC can be read, and to where data from the software verification framework can be written.
UWORD32 svSYS_DebugXVCBase;
UWORD32 svSYS_DebugXVCSWBase;

Whenever information or commands need to be exchanged between a software-interface XVC and the software, the base addresses are used to monitor requests and response activity between the XVC and the software environment. The exact communication protocol and functions are defined by the XVC and the software designer.
INDEX

Symbols
#1step 82
#define 448
#include 448
$display() 138, 139
$past 66, 288, 291
$psprintf() 139
$sformat() 139
$write() 138, 139

A
abort_on_error() 380
ABORT_SIM 137, 370
abstraction
methodology 3
acceleration
See also hardware-assisted accuracy
See also Response checking, accuracy
ACT_COMPLETED 183, 390
ACT_STARTED 390
ACTION 446, 454
action
See also XVC
action_descriptor 242
See also XVC, action descriptor
action_chan 441
activate() 179, 180, 184, 393
ACTIVATED 390
active_region 49
active_transactor
See also Proactive transactor
active_slot() 394
add_action() 441
add_to_output() 400
add_watchpoint() 377
add_xvc() 440
AFAP 399
ALAP 399
aliasing 113
ALL_SEVS 369
ALL_TYPS 369
allocate() 155, 174, 384
allocate_scenario() 423
alternative
definition 11
and property operator 49
and sequence operator 47
append_callback() 131, 132, 378, 411
apply() 235, 236, 237, 424
arbitration 55, 324
arithmetic_operations 57, 287
array
randomization 233
size of randomized 145
assert_always 430
assert_always_on_edge 430
Index

assert_arbiter 434
assert_bits 434
assert_change 430
assert_code_distance 434
assert_cycle_sequence 430
assert_data_used 434
assert_decrement 431
assert_delta 431
assert_driven 434
assert_dual_clk_fifo 435
ASSERT_END_OF_SIMULATION 84
assert_even_parity 431
assert_fifo 435
assert_fifo_index 431
assert_frame 431
ASSERT_GLOBAL_RESET 79
assert_handshake 431
assert_implication 431
assert_increment 431
ASSERT_INIT_MSG 80
assert_memory_async 435
assert_memory_sync 435
assert_multiport_fifo 436
assert_mutex 436
assert_never 431
assert_next 432
assert_next_state 436
assert_no_contention 436
assert_no_overflow 432
assert_no_transition 432
assert_no_underflow 432
assert_odd_parity 432
ASSERT_ON 78, 91
assert_one_cold 432
assert_one_hot 432
assert_proposition 432
assert_quiescent_state 433
assert_range 433
assert_reg_loaded 437
assert_req_ack_unique 437
assert_stack 437
assert_time 433
assert_transition 433
assert_unchange 433
assert_valid_id 437
assert_value 437
assert_width 433
assert_win_change 433
assert_win_unchange 433
assert_window 433
assert_zero_one_hot 434
assertion coverage 72, 262
definition 13
assertion-based verification
definition 13
assertions 43–102, 287
adopting 9
adopting for formal 10
advantages of SystemVerilog 44
applicability 33
arbitration 55
arithmetic operations 57, 287
as comments 50
as coverage points 8
ASSERT_ON 78, 91
assume 45
assumption 35
asynchronous 56
benefits 7
binding 60
bus 58, 89
category 58
coding guidelines 63–77
combinatorial logic 57
control 58, 78, 79, 81, 91, 93, 98
cover
See also Coverage property
definition 13, 44
disabling property 49
emulation
See also Assertions, formal tools
end-to-end 61
end-to-end properties 285
external 59–63
failure message 59
FIFO 55
finite-state machine 54
formal analysis 35
formal tools 281–302
implementation-specific 35
inside always block 51
internal 50–59
language 46–49
library
    See also Checker library
local variables 283
location 50, 55, 71
matching sequence 49
memory 55
message service 80, 81, 99
methodology 3
negated property 68
on interfaces 54
open-ended 63, 65
packaging 54, 60, 78, 91, 92, 94
property evaluation 49
property operators 48
qualification 100–102
reset 52, 58, 70, 79
reuse 45, 77–100
    architecture 90–99
    packaging 45
sampling design variables 82
sequence 47
sequence operators 47
stack 55
success 49
synthesizable 283
    with local variables 297–302
    without local variables 293–297
trivial 36, 56
used by testbench 61
variables 69
verification IP 45
    See also Assertions, reuse
VHDL 91
VMM_FORMAL 285
vs assumptions 93
vs scoreboard 61
vs testbench 33
X propagation 67
assume 289, 290, 291
    as random constraints 284
See also Assertions, assume
vs assert 93
asynchronous interface 111
atomic generation 231
atomic generator 122
    using 245
automation
    methodology 3
    using random stimulus 3
B
base class
    vmm_data 383
    vmm_env 365
    vmm_xactor 411
    xvc_action 442
    xvc_manager 439
    xvc_xactor 440
bcast_off() 400
bcast_on() 400
bidirectional signals
    See also Signal layer, direction of signals
black box 60
BLAST 406
block interconnect
    definition 323
boot, OS 345, 347, 348, 349
bootstrap 358
bridge 324
broadcast_mode() 399
BUILD 365
build() 128, 130, 131, 247, 331, 366
bus-functional model 169, 202
    definition 13
    HDL 206
    See also Transactor
    VMM compliance 206
bus-functional models 206–210
byte_pack() 152, 155, 386, 443
byte_size() 155, 387, 444
byte_unpack() 155, 156, 386, 444
C
callback method 198–201
    adopting 10
    injecting errors 222
Index

invoking 200
random error injection 225
registration order 256
self-checking integration 253
vmm_atomic_gen_callbacks 418
vmm_log_callbacks 381
vmm_scenario_gen_callbacks 426
vmm_xactor_callbacks 415
XVC action execution 314
callback registration 131, 132
callbacks 314
callbacks[] 443
category 79, 80, 82, 98, 99
  See also Assertions, category
CFG_DUT 365
cfg_dut() 127, 128, 129, 132, 366
channel
  consumer 175, 177
direction 175
draining to self-checking 254
producer 175, 177
  See also Transaction-level interface
checker 204
  definition 13
XVC 308
checker library 53, 77, 302, 429–437
  assert_always 57, 58, 430
  assert_always_on_edge 58, 430
  assert_arbiter 434
  assert_bits 54, 57, 434
  assert_change 54, 57, 58, 430
  assert_code_distance 54, 57, 434
  assert_cycle_sequence 54, 430
  assert_data_used 434
  assert_decrement 57, 431
  assert_delta 57, 431
  assert_driven 54, 58, 434
  assert_dual_clk_fifo 55, 435
  assert_even_parity 54, 431
  assert_fifo 55, 435
  assert_fifo_index 55, 431
  assert_frame 54, 58, 431
  assert_handshake 54, 431
  assert_hold_value 54, 57, 435
  assert_implication 58, 431
  assert_increment 57, 431
  assert_memory_async 55, 435
  assert_memory_sync 55, 435
  assert_multiport_fifo 55, 436
  assert_mutex 57, 436
  assert_never 57, 58, 431
  assert_next 54, 57, 58, 432
  assert_next_state 54, 436
  assert_no_contention 54, 57, 58, 436
  assert_no_overflow 57, 432
  assert_no_transition 432
  assert_no_underflow 57, 432
  assert_odd_parity 54, 432
  assert_one_cold 54, 57, 432
  assert_one_hot 54, 57, 432
  assert_proposition 57, 432
  assert_quiescent 84
  assert_quiescent_state 433
  assert_range 57, 433
  assert_reg_loaded 57, 437
  assert_reg_resp 57
  assert_req_ack_unique 54, 437
  assert_req_loaded 54
  assert_req_requires 54, 58
  assert_stack 55, 83, 437
  assert_time 54, 57, 433
  assert_transition 54, 433
  assert_unchange 54, 58, 433
  assert_unchanged 57
  assert_valid_id 54, 437
  assert_value 54, 57, 437
  assert_width 57, 433
  assert_win_change 54, 57, 48, 433
  assert_win_unchange 54, 57, 48, 433
  assert_window 54, 57, 433
  assert_zero_one_hot 54, 57, 434
  customizing 53
checksum 147
  circular references, avoiding 248
class of service 184, 186
class property
  definition 13
  payload 148
CLEANUP 366
cleanup() 128, 367
clock cycles 112, 170
clock domains 112, 115
clock generation 114
code coverage 262
definition 13
limitations 262
coding guidelines
assertions 63–77
command layer 116–118
services 117
transaction 116
transactor 169, 202
COMMAND_TYP 136, 369
compare() 152, 155, 386
complete() 180, 181, 184, 185, 394
COMPLETED 395
completion channel 187
completion model 176–195
blocking 177–181
completion information
See also Transaction descriptor, status information
definition 176
nonblocking 182–189
passive response 189–192
reactive response 192–195
request/response 122
compliance test suite 74, 90
components
See also Verification component
configurable RTL 24
configuration 24
DUT 129, 132
generator 122
how long to run 133
management 29
random 245
randomizing 129
test 129
transactor 168
XVC 307
configuration descriptor 120, 129, 247
transactor 168
configuration-dependent generation 239
configure() 406
connect() 395
constrained-random
definition 13
constrained-random stimulus 212
constrained-random tests 279
constraint
declarative, advantage of 228
using stream identifier 219
variable visibility 228
constraint_mode() 228
CONTINUE 137, 370
continue_msg() 380
copy() 155, 368, 385, 405
copy_data() 385
corner cases 50, 74
constraints 158
random stimulus 308
COUNT_ERROR 137, 370
COVER_ON 79, 91
coverage
assertion
See also Assertion coverage
code
See also Code coverage
cross
See also Cross coverage
definition 13
FSM
See also FSM coverage
functional
See also Functional coverage
coverage grading 266
coverage group 29, 266, 268–276
bins 269
data coverage 83
delay coverage 84
documentation 276
packaging 273
sampled using property 75
sampled using sequence 76
vs coverage property 72
weight 275
Coverage metric
XVC 308
coverage metric 260–261
<table>
<thead>
<tr>
<th>Term</th>
<th>Page References</th>
</tr>
</thead>
<tbody>
<tr>
<td>across sampling domains</td>
<td>273</td>
</tr>
<tr>
<td>data collection</td>
<td>260, 266</td>
</tr>
<tr>
<td>data sampling</td>
<td>268, 273, 275</td>
</tr>
<tr>
<td>DUT state</td>
<td>271</td>
</tr>
<tr>
<td>feedback</td>
<td>277–279</td>
</tr>
<tr>
<td>from regression</td>
<td>261</td>
</tr>
<tr>
<td>goal</td>
<td>260</td>
</tr>
<tr>
<td>relevance</td>
<td>261</td>
</tr>
<tr>
<td>scoreboard</td>
<td>271</td>
</tr>
<tr>
<td>stimulus</td>
<td>269, 270</td>
</tr>
<tr>
<td>transactions</td>
<td>272</td>
</tr>
<tr>
<td>transactor</td>
<td>272</td>
</tr>
<tr>
<td>weight</td>
<td>274, 275</td>
</tr>
<tr>
<td>coverage model</td>
<td>261–266</td>
</tr>
<tr>
<td>block interconnect</td>
<td>324</td>
</tr>
<tr>
<td>completeness</td>
<td>264</td>
</tr>
<tr>
<td>coverage space</td>
<td>262</td>
</tr>
<tr>
<td>data vs information</td>
<td>265</td>
</tr>
<tr>
<td>effort</td>
<td>259</td>
</tr>
<tr>
<td>implementation</td>
<td>264</td>
</tr>
<tr>
<td>in verification environment</td>
<td>266</td>
</tr>
<tr>
<td>verification requirements</td>
<td>21</td>
</tr>
<tr>
<td>vs verification plan</td>
<td>263</td>
</tr>
<tr>
<td>coverage object</td>
<td>273</td>
</tr>
<tr>
<td>coverage point</td>
<td>273</td>
</tr>
<tr>
<td>data sampling</td>
<td>268</td>
</tr>
<tr>
<td>implementation specific</td>
<td>267</td>
</tr>
<tr>
<td>weight</td>
<td>274</td>
</tr>
<tr>
<td>coverage property</td>
<td>8, 22, 29, 45, 72–77, 266, 276–277</td>
</tr>
<tr>
<td>category</td>
<td>99</td>
</tr>
<tr>
<td>control</td>
<td>79, 85, 91</td>
</tr>
<tr>
<td>COVER_ON</td>
<td>79, 91</td>
</tr>
<tr>
<td>definition</td>
<td>13</td>
</tr>
<tr>
<td>reset</td>
<td>74</td>
</tr>
<tr>
<td>vacuous success</td>
<td>73</td>
</tr>
<tr>
<td>vs coverage group</td>
<td>72</td>
</tr>
<tr>
<td>coverage-driven verification</td>
<td>259–279</td>
</tr>
<tr>
<td>COVFILE</td>
<td>450</td>
</tr>
<tr>
<td>CPU integration verification</td>
<td>343–364</td>
</tr>
<tr>
<td>CRC</td>
<td>147</td>
</tr>
<tr>
<td>create_watchpoint()</td>
<td>376</td>
</tr>
<tr>
<td>cross coverage</td>
<td>268</td>
</tr>
<tr>
<td>definition</td>
<td>14</td>
</tr>
<tr>
<td>CYCLE_TYP</td>
<td>136, 369</td>
</tr>
</tbody>
</table>

**D**
- data descriptor
  - See also Transaction descriptor
- data model
  - See also Transaction descriptor
- data protection
  - definition 14
- data_id 383
- debug messages 139
- DEBUG_SEV 137, 139, 369
- DEBUG_TYP 135, 139, 369
- DEBUGGER 137, 370
- debugger 346, 348
- DEFAULT_HANDLING 370
- DEFAULT_SEV 369
- DEFAULT_TYP 369
- define_scenario() 234, 422
- definitions
  - alternative 11
  - recommendation 11
  - rule 11
  - See also Terminology
  - suggestion 11
- design
  - configuration 24
  - interface-based
    - See also Interface-based design
- design for test
  - definition 14
- design for verification 6–8
  - architecture 6
  - assertions 7
  - definition 6, 14
- directed random
  - definition 14
- directed scenario 235
- directed stimulus 30, 219–221
  - bypassing generators 219
  - error injection 222
- directed testbench
  - definition 14
- directed tests 122, 328
  - vs formal tools 279
  - vs functional coverage 263, 264
  - vs random coverage 264, 279
disable iff property operator 49, 70, 71, 74
disable_types() 373
disabling property 49
discriminant class property
definition 14
display() 384
distribution 158
DONE 417, 420
driver 116, 121
   proactive 116
   reactive 116
DUMP_STACK 137, 370
DUT configuration 132
E
   embedded generator 195
   embedded software 343
   embedded stimulus 226–227
   EMIT 456
   EMPTY 390
   empty_level() 389
   emulation
      See also hardware-assisted
   enable_types() 373
   end of test
      See also vmm_env, wait_for_end()
   end_msg() 373
   end_test 134, 367
   ENDED 179, 180, 181, 183, 184, 185, 187, 384
   end-of-test 363
   watchdog timer 322
   XVC manager 318
   end-of-test condition 133, 134
   environment
      See also Verification environment
   error injection 27, 221–226
      callback registration order 256
      directed 222
      in state-dependent generators 244
      preventing 224
      random 223
      self-checking 255–256
      via callback method 222
   XVC 308
   error messages 138
   ERROR_SEV 136, 137, 138, 369
   EVENT 452
   examples
      source code xiv
      exception descriptor 221, 223, 255
      context information 224
      randomizing 224
      randomizing in callback method 225
   exec_chan 310, 313, 442
   EXECUTE 384, 446, 457
   execute() 313, 314, 443
   execution model
      See also Completion model
   expected response
      See also Response checking
   expression descriptor
      randomizing 224
      randomizing in callback method 225
   external constraint block 229
   external constraints 160
F
   factory pattern 129, 155, 174, 216, 224, 229
      adopting 10
      naming convention 218
   FAILURE_TYP 135, 138, 369
   fatal messages 138
   FATAL_SEV 136, 137, 138, 369
   FCS 147
   FIFO 55
   fill_scenario() 424
   finite-state machine 54
   FIRM_RST 413
   first_match sequence operator 66
   flow() 391
   flush() 391
   for_each 396
   for_each_offset() 396
   formal
      assertions
         See also Assertions, formal tools
   formal engines 282
   formal tools
      applicability 281
      arithmetic operations 287
      assertions
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>validation 285</td>
</tr>
<tr>
<td>assumptions 284</td>
</tr>
<tr>
<td>auxiliary variables 291</td>
</tr>
<tr>
<td>bug hunting 283, 284</td>
</tr>
<tr>
<td>engines 282</td>
</tr>
<tr>
<td>proofs 282</td>
</tr>
<tr>
<td>random stimulus 284</td>
</tr>
<tr>
<td>reachability analysis 283</td>
</tr>
<tr>
<td>reset 287</td>
</tr>
<tr>
<td>vs simulation 279</td>
</tr>
<tr>
<td>formal verification</td>
</tr>
<tr>
<td>definition 14</td>
</tr>
<tr>
<td>to qualify assertions 101</td>
</tr>
<tr>
<td>format_msg() 379</td>
</tr>
<tr>
<td>FSM</td>
</tr>
<tr>
<td>defined using constraints 243</td>
</tr>
<tr>
<td>FSM coverage 262</td>
</tr>
<tr>
<td>definition 14</td>
</tr>
<tr>
<td>FULL 390</td>
</tr>
<tr>
<td>full_level() 389</td>
</tr>
<tr>
<td>functional coverage 263–266</td>
</tr>
<tr>
<td>analysis 265</td>
</tr>
<tr>
<td>definition 14</td>
</tr>
<tr>
<td>implementation 266–277</td>
</tr>
<tr>
<td>in directed tests 264</td>
</tr>
<tr>
<td>limitations 264</td>
</tr>
<tr>
<td>model</td>
</tr>
<tr>
<td>See also Coverage model</td>
</tr>
<tr>
<td>point</td>
</tr>
<tr>
<td>definition 14</td>
</tr>
<tr>
<td>sampling 28</td>
</tr>
<tr>
<td>vs directed tests 263</td>
</tr>
<tr>
<td>vs testcases 263</td>
</tr>
<tr>
<td>functional layer 118–122</td>
</tr>
<tr>
<td>configuration 120</td>
</tr>
<tr>
<td>sub layer 166</td>
</tr>
<tr>
<td>sub layers 119, 120, 171</td>
</tr>
<tr>
<td>transaction 119</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>GEN_CFG 365</td>
</tr>
<tr>
<td>gen_cfg() 366</td>
</tr>
<tr>
<td>GENERATED 417, 420</td>
</tr>
<tr>
<td>generation</td>
</tr>
<tr>
<td>atomic 231</td>
</tr>
<tr>
<td>configuration-dependent 239</td>
</tr>
<tr>
<td>multiple steps 242</td>
</tr>
<tr>
<td>multi-stream 236–238</td>
</tr>
<tr>
<td>peer-to-peer protocols 240</td>
</tr>
<tr>
<td>scenario 232–236</td>
</tr>
<tr>
<td>state-dependent 238–244</td>
</tr>
<tr>
<td>actions 241</td>
</tr>
<tr>
<td>transitions 243</td>
</tr>
<tr>
<td>strategy, selection 244–246</td>
</tr>
<tr>
<td>generator 122, 204</td>
</tr>
<tr>
<td>atomic 122</td>
</tr>
<tr>
<td>connecting 215</td>
</tr>
<tr>
<td>constrainability 30</td>
</tr>
<tr>
<td>constructor 214</td>
</tr>
<tr>
<td>control 227–246</td>
</tr>
<tr>
<td>controllability 30</td>
</tr>
<tr>
<td>definition 14</td>
</tr>
<tr>
<td>directed interface 220</td>
</tr>
<tr>
<td>directed tests 122</td>
</tr>
<tr>
<td>embedded 195</td>
</tr>
<tr>
<td>naming convention 218</td>
</tr>
<tr>
<td>output channel 213</td>
</tr>
<tr>
<td>replacing 245, 246</td>
</tr>
<tr>
<td>requirements 26, 30</td>
</tr>
<tr>
<td>scenario 122</td>
</tr>
<tr>
<td>See also Stimulus</td>
</tr>
<tr>
<td>See also Transactor</td>
</tr>
<tr>
<td>stopping 219</td>
</tr>
<tr>
<td>stream_id 218</td>
</tr>
<tr>
<td>transactor 213</td>
</tr>
<tr>
<td>get() 175, 179, 186, 393</td>
</tr>
<tr>
<td>get_instance() 368, 411</td>
</tr>
<tr>
<td>get_message_count() 376</td>
</tr>
<tr>
<td>get_name() 368, 411, 442</td>
</tr>
<tr>
<td>get_notification() 408</td>
</tr>
<tr>
<td>get_object() 403</td>
</tr>
<tr>
<td>get verbosity() 374</td>
</tr>
<tr>
<td>GOT 390</td>
</tr>
<tr>
<td>guidelines</td>
</tr>
<tr>
<td>basic 12</td>
</tr>
</tbody>
</table>

| H |
| HARD_RST 413 |
| hardware-assisted verification 336–342, 349 |
synthesizable transactor 336
vs simulation 336, 349
HW/SW verification 343–364

I
IEEE xvi
if/else property operator 49
IGNORE 370
INACTIVE 395
indicate() 408
inheritance 149
inject() 417, 421
inject_obj() 421
input signals
See also Signal layer, direction of
signals
interface-based design 4–5
verification requirements 5
INTERNAL_TYP 136, 369
INTERRUPT 446, 455
interrupt_chan 442
intersect sequence operator 47, 64, 65
is_above() 368
is_configured() 406
is_full() 389
is_locked() 392
is_on() 407
is_valid() 155, 384
is_waited_for() 407
items[] 423

L
layers 104
command 116–118
functional 118–122
scenario 122–123
signal 107–116
test 123–124
length 422
level() 389
line coverage 262
list() 369
load() 387
lock() 391
LOCKED 391
LOG 449
log 365, 389, 398, 411, 421, 439
log_start() 375
log_stop() 375
loop-back 326

M
main() 164, 165, 414
MAPEVENT 453
mapping signals 113
max_byte_size() 387, 444
memory 55
message
example 381, 382
example using macro 382
filters 135
handling 137
promotion example 382
severity 136
source, definition of 134
type 135
message service 134–139
assertions 59
debug 139
errors 138
fatal 138
filter 135
format
See also vmm_log_format
handling 137
in assertions 80, 81, 99
issuing messages 144
See also vmm_log
severities 369
severity 136
simulation handling 370
source 134
trace 139
type 135
verbose 139
warnings 138
methodology
adoption 8–10
directed testcases 2
history 2
objectives 1
model checking 281, 282–292
   See also Formal tools
modify() 374
modport 110, 111, 169
module
top-level 112, 114, 124, 206
monitor 117
definition 15
embedded 32–33
passive 117, 118
reactive 117, 118
See also Transactor
multi-stream generation 236–238
multi-stream scenario 238

N
naming convention 162
negated property 68
new() 366, 368, 383, 388, 401, 405, 411,
   419, 440, 441, 442
new_output() 400
new_source() 402
NORMAL_SEV 136, 138, 369
not property operator 48
NOTE_TYP 135, 138, 139, 369
notify 167, 365, 384, 389, 412, 440, 441

O
observability 32
offline checking 40–41
   applicability 40
   limitations 40
ON_OFF 406
ONE_SHOT 406
Open Verification Library 44, 53, 84,
   429–434
operating system 345, 349
or property operator 49
or sequence operator 47
out_chan 416, 419
output signals
   See also Signal layer, direction of
   signals
OVL 44, 53, 84, 429–434

P
packed data 155
parity 147
parse() 312, 441, 443
pass_or_fail 380
passive response 189–192
passive transactor 110, 117, 118, 119, 122,
   166, 167, 174, 176, 189, 270
   definition 15
peek() 179, 393
PEEKED 390
peer-to-peer protocols 240
PENDING 395
performance, verifying 329
peripheral test block 327, 336, 339–342,
   346
clock domains 339, 340
examples 337
external interface 337
structure 339
XVC 338
physical interface
   implementation 169
physical layer
   See also Signal layer
   physical-level interface 169–170
pipelined transaction execution 185
planning 17–41
   contributors 18
   coverage sampling 28
   directed tests 18
   environment requirements 22–28
   error detection 19
   error injection 27
   for SystemVerilog 18
   partitioning 23
   process 18–31
   response checking 31–41
   See also Response checking
   See also Verification plan
   trivial tests 26
   verification implementation 29–31
   verification requirements 18–22
   coverage model 21
   identifying 20
ranking 20
playback() 397
polymorphism 229
post_inst_gen() 418
post_scenario_gen() 426
pre_abort() 381
pre_debug() 381
pre_scenario_randomize() 426
pre_stop() 381
prepend_callback() 131, 132, 256, 378, 411
preponed region 49
proactive transactor 110, 116, 119, 166, 167, 176
definition 15
productivity 2–4
promotion example 382
property
class
definition 13
coverage
definition 13
definition 15
success 49
property checker See also Model checking
property operators 48
protocol checker 204
protocol layers 166
protocol response 195
PROTOCOL_RST 413
PROTOCOL_TYP 136, 369
psdisplay() 138, 139, 144, 155, 384
PUT 390
put() 173, 175, 176, 178, 179, 183, 186, 221, 392
rand_mode() 228
random environment
See also Verification environment
random generator
See also Generator
random response 194, 195
random scenario 235
random seed 29, 263
random stimulus 212, 213–219, 227–246, 324
as background noise 219
automation 3
constrained 212
constraint_mode() 228
error injection 223
external constraint block 229
rand_mode() 228
random tests
constrained 279
vs directed tests 264, 279
vs formal tools 279
randomize with 146
randomize() 229, 239
return value 218
randomized_obj 416
randomized_sched 404
randsequence 235
shortcomings 232
reachability analysis 283
reactive channel 192
reactive region 109, 117, 121, 123, 162
reactive response 192–195
default response 194, 195
random 194
random response 195
response delay 193
reactive transactor 110, 116, 117, 118, 119, 122, 166, 167, 174, 176, 177, 192, 195
definition 15
Recommendation
definition 11
reconfigure() 168, 389
record() 396
redefine_scenario() 422
reference model 39–40

Q
quality of service 38

R
race conditions
between DUT and testbench 109, 117, 121, 123, 162
initialization 114, 124
random environment
See also Verification environment
random generator
See also Generator
random response 194, 195
random scenario 235
random seed 29, 263
random stimulus 212, 213–219, 227–246, 324
as background noise 219
automation 3
constrained 212
constraint_mode() 228
error injection 223
external constraint block 229
rand_mode() 228
random tests
constrained 279
vs directed tests 264, 279
vs formal tools 279
randomize with 146
randomize() 229, 239
return value 218
randomized_obj 416
randomized_sched 404
randsequence 235
shortcomings 232
reachability analysis 283
reactive channel 192
reactive region 109, 117, 121, 123, 162
reactive response 192–195
default response 194, 195
random 194
random response 195
response delay 193
reactive transactor 110, 116, 117, 118, 119, 122, 166, 167, 174, 176, 177, 192, 195
definition 15
Recommendation
definition 11
reconfigure() 168, 389
record() 396
redefine_scenario() 422
reference model 39–40
Index

applicability 40
  C 39, 40
  vs scoreboard 40
register tests 346, 354, 358
registers, verifying 327, 328
remove() 179, 180, 184, 394
remove_watchpoint() 377
remove_xvc() 440
repeat_thresh 423
repeated 423
REPORT 366
report() 367, 377
REPORT_TYP 136, 369
request/response 122
reset 287
reset() 408, 409
RESET_DUT 365
reset_dut() 116, 127, 128, 366
reset_xactor() 164, 165, 169, 311, 399, 402, 412
response channel 192
response checking 246–256
  accuracy 36–38
    cycle-by-cycle 36
    losses 38
    ordering 36, 37
    timing 37
    transaction 37
  assertions 33–36
  computations 34
  correctness, inferring 31
  formal analysis 35
  in directed testcase 31
  in random testcase 31
  internal signals 35
  offline
    See also Offline checking
  ordering 34
  physical-level 34
  planning 31–41
  reference model
    See also Reference model
scoreboard
  See also Scoreboard
  See also Self-checking
  transformation 34
  types of failures 25
    white box 35
response descriptor 189
response model
  definition 176
    See also Completion model
response request descriptors 194
restore_rng_state() 415
reusable assertions
  See also Assertions, reuse
  reuse
    methodology 4
RTL configuration 24
RTL model vs transaction-level model
  See also transaction-level model
Rule
  definition 11
run() 127, 128, 366
S
sample() 268
save() 387
save_rng_state() 414
SCENARIO 446, 451
scenario
  definition 15, 122
  system-level 307
scenario descriptor 232, 233
  directed 235
  multi-stream 238
  random scenario 235
scenario generation 232–236
scenario descriptor 232
  vs atomic generation 232
scenario generator 122
  using 245
scenario layer 122–123
  generator 122
scenario_id 234, 235, 383, 422
scenario_kind 422
scenario_name() 422
scenario_set[$] 420
schedule() 403
schedule_off() 402
schedule_on() 402
<table>
<thead>
<tr>
<th>Scoreboard</th>
<th>38–39, 249–252</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>250, 251</td>
</tr>
<tr>
<td>Applicability</td>
<td>39</td>
</tr>
<tr>
<td>Checking ordering</td>
<td>249, 250</td>
</tr>
<tr>
<td>Data structure</td>
<td>249</td>
</tr>
<tr>
<td>Definition</td>
<td>39</td>
</tr>
<tr>
<td>Hashing function</td>
<td>252</td>
</tr>
<tr>
<td>Integration callback</td>
<td>131</td>
</tr>
<tr>
<td>Lost transactions</td>
<td>250</td>
</tr>
<tr>
<td>See also Self-checking</td>
<td></td>
</tr>
<tr>
<td>Transaction tagging</td>
<td>252</td>
</tr>
<tr>
<td>Vs assertions</td>
<td>61</td>
</tr>
<tr>
<td>Vs reference model</td>
<td>40</td>
</tr>
<tr>
<td>Select scenario</td>
<td>420</td>
</tr>
<tr>
<td>Self-checking</td>
<td></td>
</tr>
<tr>
<td>Avoiding circular references</td>
<td>248</td>
</tr>
<tr>
<td>Callback extensions</td>
<td>253</td>
</tr>
<tr>
<td>Callback registration order</td>
<td>256</td>
</tr>
<tr>
<td>Channel consumer</td>
<td>254</td>
</tr>
<tr>
<td>Configuration</td>
<td>247</td>
</tr>
<tr>
<td>Error injection</td>
<td>255–256</td>
</tr>
<tr>
<td>Exception descriptor</td>
<td>255</td>
</tr>
<tr>
<td>In verification environment</td>
<td>247</td>
</tr>
<tr>
<td>Interface</td>
<td>248, 249</td>
</tr>
<tr>
<td>Notification status</td>
<td>254</td>
</tr>
<tr>
<td>Packaging</td>
<td>246</td>
</tr>
<tr>
<td>See also Reference model</td>
<td></td>
</tr>
<tr>
<td>See also Response checking</td>
<td></td>
</tr>
<tr>
<td>See also Scoreboard</td>
<td></td>
</tr>
<tr>
<td>Transactor integration</td>
<td>253–255</td>
</tr>
<tr>
<td>Self-checking testbench</td>
<td>246–256</td>
</tr>
<tr>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td>Matching</td>
<td>49</td>
</tr>
<tr>
<td>See also Scenario generation</td>
<td></td>
</tr>
<tr>
<td>Sequence operators</td>
<td>46</td>
</tr>
<tr>
<td>Set_format()</td>
<td>371</td>
</tr>
<tr>
<td>Set_log()</td>
<td>383</td>
</tr>
<tr>
<td>Set_notification()</td>
<td>408</td>
</tr>
<tr>
<td>Set_sev_image()</td>
<td>371</td>
</tr>
<tr>
<td>Set_typ_image()</td>
<td>371</td>
</tr>
<tr>
<td>Set_verbosity()</td>
<td>374</td>
</tr>
<tr>
<td>Signal aliasing</td>
<td>113</td>
</tr>
<tr>
<td>Signal layer</td>
<td>106, 107–116</td>
</tr>
<tr>
<td>Asynchronous signals</td>
<td>111</td>
</tr>
<tr>
<td>Clock generation</td>
<td>114</td>
</tr>
<tr>
<td>Direction of signals</td>
<td>111, 112</td>
</tr>
<tr>
<td>Packaging</td>
<td>108</td>
</tr>
<tr>
<td>Signal declarations</td>
<td>108</td>
</tr>
<tr>
<td>Signal mapping</td>
<td>113</td>
</tr>
<tr>
<td>Synchronous signals</td>
<td>109, 112</td>
</tr>
<tr>
<td>Simulation</td>
<td></td>
</tr>
<tr>
<td>Definition</td>
<td>15</td>
</tr>
<tr>
<td>Ending</td>
<td></td>
</tr>
<tr>
<td>See also vmm_env, wait_for_end()</td>
<td></td>
</tr>
<tr>
<td>Simulation control</td>
<td>124–139</td>
</tr>
<tr>
<td>DUT configuration</td>
<td>132</td>
</tr>
<tr>
<td>Executing the test</td>
<td>133, 134</td>
</tr>
<tr>
<td>Instantiation</td>
<td>130</td>
</tr>
<tr>
<td>See also vmm_env</td>
<td></td>
</tr>
<tr>
<td>Starting environment</td>
<td>133</td>
</tr>
<tr>
<td>Test configuration</td>
<td>129</td>
</tr>
<tr>
<td>Sink</td>
<td>392</td>
</tr>
<tr>
<td>Sink()</td>
<td>391</td>
</tr>
<tr>
<td>Size()</td>
<td>389</td>
</tr>
<tr>
<td>Sneak()</td>
<td>176, 188, 190, 193, 392</td>
</tr>
<tr>
<td>SOFT_RST</td>
<td>413</td>
</tr>
<tr>
<td>Software tests</td>
<td></td>
</tr>
<tr>
<td>Bootstrap</td>
<td>363–364</td>
</tr>
<tr>
<td>Compilation</td>
<td>359–361</td>
</tr>
<tr>
<td>Configuration</td>
<td>359</td>
</tr>
<tr>
<td>Directory structure</td>
<td>360</td>
</tr>
<tr>
<td>Execution order</td>
<td>362</td>
</tr>
<tr>
<td>Portability</td>
<td>359</td>
</tr>
<tr>
<td>Running</td>
<td>361–363</td>
</tr>
<tr>
<td>See also Test action</td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>349–354</td>
</tr>
<tr>
<td>Software-accessible registers</td>
<td>327</td>
</tr>
<tr>
<td>Source code xiv</td>
<td></td>
</tr>
<tr>
<td>Split transaction execution</td>
<td>185</td>
</tr>
<tr>
<td>Split()</td>
<td>440</td>
</tr>
<tr>
<td>Stack</td>
<td>55</td>
</tr>
<tr>
<td>START</td>
<td>365</td>
</tr>
<tr>
<td>Start()</td>
<td>128, 133, 180, 184, 367, 394</td>
</tr>
<tr>
<td>Start_msg()</td>
<td>371</td>
</tr>
<tr>
<td>Start_xactor()</td>
<td>164, 165, 203, 311, 398, 402, 412</td>
</tr>
<tr>
<td>STARTED</td>
<td>179, 180, 184, 187, 191, 384, 395</td>
</tr>
<tr>
<td>State coverage</td>
<td>271</td>
</tr>
<tr>
<td>State-dependent generation</td>
<td>238–244</td>
</tr>
</tbody>
</table>
Index

injecting errors 244
status descriptor 189
status() 394, 408
stimulus 211–227
constrained-random 212
coverage 269, 270
definition 15
directed 219–221
bypassing generators 219
See also Directed stimulus
XVC 308
directed vs random 212, 219, 228
embedded 226–227
error injection 221–226
random 212, 213–219, 227–246
XVC 308
requirements 26
synchronizing streams 237
STOP 366
stop() 128, 367
stop_after_n_insts 416, 419
stop_after_n_scenarios 419
STOP_PROMPT 137, 370
stop_xactor() 165, 311, 399, 402, 412
STOPONERROR 446, 450
STOPONEVENT 446, 451
stream identification 167
stream_id 167, 218, 383, 411, 421
structural coverage
definition 15
See also Code coverage
sub layers 119, 120, 166
suggestion
definition 11
super.apply() 236
super.main() 165
SV_A_CHECKER_FORMAL 81, 98
svIO_BYTE_READ() 356
svIO_BYTE_WRITE() 356
svIO_WORD_READ() 356
svIO_WORD_WRITE() 356
svSYS_ACTION_RUN() 361
svSYS_ACTION_SHEET_RUN() 362
svSYS_ActionSheetItem 362
svSYS_AllActionSheetsRun() 362
svSYS_ASSERT() 357
svSYS_CACHE_BLOCK_END() 357
svSYS_CACHE_BLOCK_START() 35
7
svSYS_Element 353
svSYS_GET_SYS_DATA() 356
svSYS_HARDCODED() 356
svSYS_Peripherals 362
svSYS_Printf() 358
svSYS_SeqTest() 355
svSYS_Sequence 362
svSYS_SystemData 352
svSYS_SystemElement 352
svSYS_ThrowException() 357
svTEST_LEVEL() 356
svTEST_NAME() 355
testFailed 358
synchronous interface 109, 112
SYNTHESIS 80, 289
synthesizable assertions
See also Assertions, synthesizable
SysData 356
system
definition 15, 305
system descriptor 351, 353, 354, 358
system descriptor 352
system-level scenarios 307
system-level verification 305–342
objective 305
vs block-level 305
T
tagged union 150, 152
TCP 240
te() 395
tee_mode() 395
temporal expression 34
formal analysis 35
terminated() 408
terminology 13–16
test
definition 16
See also Testcase
test action 354, 354–364
complexity level 356
naming convention 355
Verification Methodology Manual for SystemVerilog

portability 354
See also Software tests
test action sheet 354, 362
   item 354
test configuration 133
test layer 123–124
   implementation 123
testbench
   ad-hoc 201–206
definition 16
   vs assertions 33
testcase
   definition 16
directed
   generator, bypassing 122
   implementation 107, 119, 123
   portability 28
See also Simulation control
   trivial 26
testing
   definition 16
text() 371
throughout sequence operator 47, 74
timestamp() 408
timing interface 195–198
   connecting 197
TIMING_TYP 135, 369
toggle coverage 262
top-level module 112, 114, 124, 206
trace 439, 440
trace messages 139
TRACE_SEV 137, 139, 369
transaction
   command layer 116
cov
   definition 16
describing exceptions 221
   functional layer 119
   interface
   See also vmm_channel
   serial numbers 252
   vs transactors 161
transaction descriptor 140–160, 171, 189
   adding information to 174
   advantages 141
asynchronous information 196
basic constraints 157
composition 151, 153
constraints 157–160
constructor 154
context 146
controlling randomization 30
corner cases 158
data members 143–154
data protection 147, 148, 157
discriminant 145, 152, 153, 156, 159
error prevention constraints 160
executing 176
external constraints 160
implementation 146
incomplete 190
inheritance 149
local properties 146
message service 144
methods 154–157
packaging 143
public properties 146
random distribution 158
random properties 145, 146
state-dependent 239
status information 176, 181, 185, 189
virtual methods 154
   vs procedures 141
TRANSACTION_TYP 136, 369
transaction-level interface 171–195
   bidirectional 176
   channel 172
   class of service 184
   connecting 173
   in models 334–335
   naming convention 172
   procedural 171, 205
See also channel
See also vmm_channel
transaction-level model 117, 349
   and SystemVerilog 334
   packaging 333
   using transactors 334
   verifying 332–335
   vs RTL model 330, 333
transactor 161–170
  active
    See also Proactive transactor
callback methods
  See also Callback method
class of service 186
command level 169, 202
configuration 168
configuration descriptor 168
connecting two 173, 197
constructor 164, 168, 169, 173, 197
coverage 272
definition 16, 161
driver 116
DUT independence 253
embedded
  See also Generator, embedded
  See also monitor.embedded
extending
  See also Callback method
generator 123, 213
implementation 163, 164
in transaction-level model 334
memory mapped 345
message service 167
naming convention 162, 202
notifications 167
out-of-order
  See also Completion model, non-blocking
packaging 162, 163, 202
passive 110, 117, 118, 119, 122, 166,
  167, 174, 176, 189, 270
  See also Passive transactor
physical-level interface 169–170
pipelined 185
proactive 110, 116, 119, 166, 167, 176
  See also Proactive transactor
reactive 110, 116, 117, 118, 119, 122,
  166, 167, 174, 176, 177, 192, 195
  See also Reactive transactor
reconfigure() 168
replacing design blocks 117, 226
See also Transaction-level interface
See also Verification component
self-checking integration 253–255
split transactions 185
stream identification 167
stream_id 218
synthesizable 336
threads 164
timing interface
  See also Timing interface
transaction descriptor 141
vs actual CPU 344
vs CPU or DSP 319
vs transactions 161
vs XVC 306
transfer function 38
trivial tests 26
unlock() 391
UNLOCKED 391
unmodify() 375
unput() 393
unregister_callback() 378, 412
using 423
validation
  definition 16
variables in assertions 69
verbose messages 139
VERBOSE_SEV 137, 139, 369
VERBOSITY 446, 448
verification
  definition 16
design for
  See also Design for verification
verification component 4–5
configuration 4
definition 16
testable 306–316
  See also XVC
 identifying 23
reuse 321
See also Transactor
synthesizable 336
verification environment 104
architecture 104–124
basic software integration 345, 345–346
block integration 320, 326–328
block interconnect 320, 323–326
bottom-up implementation 120
configuration 331
coverage points 266
definition 16
implementation 106, 107
instantiation 123
layers 104
portability 330, 332
See also vmm_env
self-checking 247
signal layer 106
software 343–349
See also Software tests
stimulus
See also Stimulus
system functional 320, 328
system validation 321, 329–331
system, full 345, 346–349
system-level 319–331
transaction-level model 332
using assertions 61
verification IP 86–90
documentation 99–100
See also Assertions, verification IP
verification plan
definition 17
virtual method 126
virtual modport 169
vmm_atomic_gen 231, 415–418
DONE 417
GENERATED 417
inj ect() 417
out_chan 416
randomized_obj 416
stop_after_n_insts 416
vmm_atomic_gen_callbacks 418
vmm_atomic_gen() 213
vmm_atomic_gen_callbacks 418
post_inst_gen() 418
vmm_atomic_scenario 424
vmm_broadcast 397–401
add_to_output() 400
AFAP 399
ALAP 399
bcast_off() 400
bcast_on() 400
broadcast_mode() 399
log 398
new_output() 400
reset_xactor() 399
start_xactor() 398
stop_xactor() 399
vmm_callback() 200, 415
vmm_channel 142, 172, 335, 387–397
ACT_COMPLETED 183, 390
ACT_STARTED 390
activate() 179, 180, 184, 393
ACTIVATED 390
active_slot() 394
adopting 9
complete() 180, 181, 184, 185, 394
COMPLETED 395
connect() 395
crossing language boundaries 335
EMPTY 390
empty_level() 389
flow() 391
flush() 391
for_each() 396
for_each_offset() 396
FULL 390
full_level() 389
get() 175, 179, 186, 393
GOT 390
INACTIVE 395
is_full() 389
is_locked() 392
level() 389
log() 391
LOCKED 391
log 389
new() 388
notify 389
peek() 179, 393
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PEEKED 390</td>
<td></td>
</tr>
<tr>
<td>PENDING 395</td>
<td></td>
</tr>
<tr>
<td>playback() 397</td>
<td></td>
</tr>
<tr>
<td>PUT 390</td>
<td></td>
</tr>
<tr>
<td>put() 173, 175, 176, 178, 183, 186, 221, 392</td>
<td></td>
</tr>
<tr>
<td>reconfigure() 389</td>
<td></td>
</tr>
<tr>
<td>record() 396</td>
<td></td>
</tr>
<tr>
<td>remove() 179, 180, 184, 394</td>
<td></td>
</tr>
<tr>
<td>SINK 392</td>
<td></td>
</tr>
<tr>
<td>sink() 391</td>
<td></td>
</tr>
<tr>
<td>size() 389</td>
<td></td>
</tr>
<tr>
<td>sneak() 176, 188, 193, 392</td>
<td></td>
</tr>
<tr>
<td>SOURCE 392</td>
<td></td>
</tr>
<tr>
<td>start() 180, 184, 394</td>
<td></td>
</tr>
<tr>
<td>STARTED 395</td>
<td></td>
</tr>
<tr>
<td>status() 394</td>
<td></td>
</tr>
<tr>
<td>tee() 395</td>
<td></td>
</tr>
<tr>
<td>tee_mode() 395</td>
<td></td>
</tr>
<tr>
<td>unlock() 391</td>
<td></td>
</tr>
<tr>
<td>UNLOCKED 391</td>
<td></td>
</tr>
<tr>
<td>unput() 393</td>
<td></td>
</tr>
<tr>
<td>vmm_command() 372</td>
<td></td>
</tr>
<tr>
<td>vmm_cycle() 373</td>
<td></td>
</tr>
<tr>
<td>vmm_data 142, 155, 181, 213, 231, 236, 238–387</td>
<td></td>
</tr>
<tr>
<td>adopting 9</td>
<td></td>
</tr>
<tr>
<td>allocate() 155, 174, 384</td>
<td></td>
</tr>
<tr>
<td>byte_pack() 152, 155, 386</td>
<td></td>
</tr>
<tr>
<td>byte_size() 155, 387</td>
<td></td>
</tr>
<tr>
<td>byte_unpack() 155, 156, 386</td>
<td></td>
</tr>
<tr>
<td>compare() 152, 155, 386</td>
<td></td>
</tr>
<tr>
<td>copy() 155, 385</td>
<td></td>
</tr>
<tr>
<td>copy_data() 385</td>
<td></td>
</tr>
<tr>
<td>coverage 272</td>
<td></td>
</tr>
<tr>
<td>data_id 383</td>
<td></td>
</tr>
<tr>
<td>display() 384</td>
<td></td>
</tr>
<tr>
<td>ENDED 179, 180, 181, 183, 184, 185, 187, 384</td>
<td></td>
</tr>
<tr>
<td>EXECUTE 384</td>
<td></td>
</tr>
<tr>
<td>is_valid() 155, 384</td>
<td></td>
</tr>
<tr>
<td>load() 387</td>
<td></td>
</tr>
<tr>
<td>log 144</td>
<td></td>
</tr>
<tr>
<td>max_byte_size() 387</td>
<td></td>
</tr>
<tr>
<td>methods 155</td>
<td></td>
</tr>
<tr>
<td>new() 383</td>
<td></td>
</tr>
<tr>
<td>notify 384</td>
<td></td>
</tr>
<tr>
<td>psdisplay() 139, 144, 155, 384</td>
<td></td>
</tr>
<tr>
<td>save() 387</td>
<td></td>
</tr>
<tr>
<td>scenario_id 383</td>
<td></td>
</tr>
<tr>
<td>set_log() 383</td>
<td></td>
</tr>
<tr>
<td>sneak() 190</td>
<td></td>
</tr>
<tr>
<td>STARTED 179, 180, 184, 187, 191, 384</td>
<td></td>
</tr>
<tr>
<td>stream_id 167, 218, 383</td>
<td></td>
</tr>
<tr>
<td>vmm_debug() 139, 372</td>
<td></td>
</tr>
<tr>
<td>vmm_env 365–376</td>
<td></td>
</tr>
<tr>
<td>vmm_cycle() 373</td>
<td></td>
</tr>
<tr>
<td>vmm_command() 372</td>
<td></td>
</tr>
<tr>
<td>vmm_cycle() 373</td>
<td></td>
</tr>
<tr>
<td>vmm_data 142, 155, 181, 213, 231, 236, 238–387</td>
<td></td>
</tr>
<tr>
<td>vmm_cycle() 373</td>
<td></td>
</tr>
<tr>
<td>vmm_data 142, 155, 181, 213, 231, 236, 238–387</td>
<td></td>
</tr>
<tr>
<td>adopting 9</td>
<td></td>
</tr>
<tr>
<td>allocate() 155, 174, 384</td>
<td></td>
</tr>
<tr>
<td>byte_pack() 152, 155, 386</td>
<td></td>
</tr>
<tr>
<td>byte_size() 155, 387</td>
<td></td>
</tr>
<tr>
<td>byte_unpack() 155, 156, 386</td>
<td></td>
</tr>
<tr>
<td>compare() 152, 155, 386</td>
<td></td>
</tr>
<tr>
<td>copy() 155, 385</td>
<td></td>
</tr>
<tr>
<td>copy_data() 385</td>
<td></td>
</tr>
<tr>
<td>coverage 272</td>
<td></td>
</tr>
<tr>
<td>data_id 383</td>
<td></td>
</tr>
<tr>
<td>display() 384</td>
<td></td>
</tr>
<tr>
<td>ENDED 179, 180, 181, 183, 184, 185, 187, 384</td>
<td></td>
</tr>
<tr>
<td>EXECUTE 384</td>
<td></td>
</tr>
<tr>
<td>is_valid() 155, 384</td>
<td></td>
</tr>
<tr>
<td>load() 387</td>
<td></td>
</tr>
<tr>
<td>log 144</td>
<td></td>
</tr>
<tr>
<td>max_byte_size() 387</td>
<td></td>
</tr>
<tr>
<td>methods 155</td>
<td></td>
</tr>
<tr>
<td>new() 383</td>
<td></td>
</tr>
<tr>
<td>notify 384</td>
<td></td>
</tr>
<tr>
<td>psdisplay() 139, 144, 155, 384</td>
<td></td>
</tr>
<tr>
<td>save() 387</td>
<td></td>
</tr>
<tr>
<td>scenario_id 383</td>
<td></td>
</tr>
<tr>
<td>set_log() 383</td>
<td></td>
</tr>
<tr>
<td>sneak() 190</td>
<td></td>
</tr>
<tr>
<td>vmm_cycle() 373</td>
<td></td>
</tr>
<tr>
<td>vmm_data 142, 155, 181, 213, 231, 236, 238–387</td>
<td></td>
</tr>
<tr>
<td>vmm_cycle() 373</td>
<td></td>
</tr>
<tr>
<td>vmm_data 142, 155, 181, 213, 231, 236, 238–387</td>
<td></td>
</tr>
</tbody>
</table>

498 Verification Methodology Manual for SystemVerilog
CONTINUE 137, 370
copy() 368
COUNT_ERROR 137, 370
create_watchpoint() 376
CYCLE_TYP 369
DEBUG_SEV 137, 139, 369
DEBUG_TYP 135, 139, 369
DEBUGGER 137, 370
DEFAULT.SEVER 369
DEFAULT_TYP 369
disable_types() 373
DUMP_STACK 137, 370
enable_types() 373
dend_msg() 373
ERROR.SEVER 136, 137, 138, 369
FAILURE_TYP 135, 138, 369
FATAL.SEVER 136, 137, 138, 369
get_instance() 368
get_message_count() 376
get_name() 368
get_verbosity() 374
IGNORE 370
INTERNAL_TYP 369
is_above() 368
list() 369
log_start() 375
log_stop() 375
modify() 374
new() 368
NORMAL.SEVER 136, 138, 369
NOTE_TYP 135, 138, 139, 369
prepend_callback() 378
PROTOCOL_TYP 369
remove_watchpoint() 377
report() 377
REPORT_TYP 136, 369
See also Message service
set_format() 371
set.SEVER_image() 371
set_TYP_image() 371
set_verbosity() 374
severities 369
simulation handling 370
start_msg() 371
STOP_PROMPT 137, 370
text() 371
TIMING_TYP 135, 369
TRACE.SEVER 137, 139, 369
TRANSACTION_TYP 136, 369
unmodify() 375
unregister_callback() 378
VERBOSE.SEVER 137, 139, 369
vmm_command() 372
vmm_cycle() 373
vmm_debug() 139, 372
vmm_error() 138, 372
vmm_fatal() 138, 372
vmm_log_callbacks 381
vmm_log_format 379
vmm_log_msg 378
vmm_note() 139, 372
vmm_protocol() 373
vmm_report() 372
vmm_trace() 139, 372
vmm_transaction() 373
vmm_verbose() 139, 372
vmm_warning() 138, 372
wait_for_msg() 377
wait_for_watchpoint() 377
WARNING.SEVER 138, 369
XHANDLING_TYP 135, 369
vmm_log_callbacks 381
pre_abort() 381
pre_debug() 381
pre_stop() 381
vmm_log_format 379
abort_on_error() 380
continue_msg() 380
format_msg() 379
pass_or_fail() 380
vmm_log_msg 378
vmm_note() 139, 372
vmm_notification 409
reset() 409
vmm_notify 196, 254, 405–410
BLAST 406
configure() 406
copy() 405
get_notification() 408
indicate() 408
Index

is_configured() 406
is_on() 407
is_waited_for() 407
new() 405
ON_OFF 406
ONE_SHOT 406
reset() 408
set_notification() 408
status() 408
terminated() 408
timestamp() 408
vmm_notification 409
wait_for() 407
wait_for_off() 407
vmm_protocol() 373
vmm_report() 372
vmm_scenario 234, 421
allocate_scenario() 423
apply() 235, 236, 424
define_scenario() 234, 422
fill_scenario() 424
items[] 423
length 422
log 421
redefine_scenario() 422
repeat_thresh 423
repeated 423
scenario_id 234, 235, 422
scenario_kind 422
scenario_name() 422
stream_id 421
using 423
vmm_atomic_scenario 424
vmm_scenario_election 425
vmm_scenario_gen 234, 236, ??–427
apply() 237
DONE 420
GENERATED 420
inject() 421
inject_obj() 421
new() 419
out_chan 419
scenario_set[$] 420
select_scenario 420
stop_after_n_insts 419
stop_after_n_scenarios 419
vmm_scenario 421
vmm_scenario_election 425
vmm_scenario_gen_callbacks 426
vmm_scenario_gen() 213
post_scenario_gen() 426
pre_scenario_randomize() 426
vmm_scheduler 401–405
generate() 403
new() 401
new_source() 402
randomized_sched 404
reset_xactor() 402
schedule() 403
schedule_off() 402
schedule_on() 402
start_xactor() 402
stop_xactor() 402
vmm_scheduler_election 404
vmm_trace() 139, 372
vmm_transaction() 373
vmm_verbose() 139, 372
vmm_warning() 138, 372
vmm_xactor 164, 199, 200, 411–415
adopting 10
append_callback() 131, 132, 411
constructor 164
FIRM_RST 413
get_instance() 411
get_name() 411
HARD_RST 413
log 167, 411
main() 164, 165, 414
new() 411
notify 167, 412
prepend_callback() 131, 132, 256, 411
PROTOCOL_RST 413
reset_xactor() 164, 165, 169, 412
restore_rng_state() 415
save_rng_state() 414
SOFTWARE_RST 413
start_xactor() 164, 165, 203, 412
stop_xactor() 165, 412
stream_id 167, 411
super.main() 165
threads 164
unregister_callback() 412
vmm_callback() 415
vmm_xactor_callbacks 415
wait_if_stopped() 413
wait_if_stopped_or_empty() 413
XACTOR_BUSY 412
XACTOR_IDLE 412
XACTOR_RESET 412
XACTOR_STARTED 412
xactor_status() 415
XACTOR_STOPPED 412
vmm_xactor_callbacks 199, 415
vmm_xvc_manager 444–??
#define 448
#include 448
ACTION 446, 454
comments 447
COVFILE 450
EMIT 456
EVENT 452
EXECUTE 446, 457
INTERRUPT 446, 455
LOG 449
MAPEVENT 453
notifications 444–445
SCENARIO 446, 451
See also XVC manager
STOPONERROR 446, 450
STOPONEVENT 446, 451
test scenario 317
test scenario completion 318, 319
test scenario syntax ??–457
test specification 317
VERBOSITY 446, 448
WAIT 456
XVCTRACE 450
vmmd_log
DEFAULT_HANDLING 370
vmmdata
psdisplay() 138

W
WAIT 456
wait_for() 407
WAIT_FOR_END 365
wait_for_end 367
wait_for_end() 128, 133, 134
wait_for_msg() 377
wait_for_off() 407
wait_for_watchpoint() 377
wait_if_interrupted() 315, 316, 442
wait_if_stopped() 413
wait_if_stopped_or_empty() 413
warning messages 138
WARNING_SEV 136, 138, 369
when inheritance 150, 152
white-box verification 35
within sequence operator 47, 64

X
XACTOR_BUSY 412
XACTOR_IDLE 412
XACTOR_RESET 412
XACTOR_STARTED 412
xactor_status() 415
XACTOR_STOPPED 412
xactors[] 310, 311, 314, 442
XHANDLING_TYP 135, 369
XVC 306–316, 348
action 309
callback methods 314
definition 311
execution 313
interrupt 315
See also xvc_action
action descriptor 310, 311, 312
action library 306, 307
architecture 306–309
commands 312
configuration 307, 308
coordinating 316
coverage metrics 308
directed random 308
directed stimulus 308
driver layer 307, 310
error injection 308
generator layer 307, 309, 310
implementation 309–311
implementing
  action 311–316
interrupt actions 315
layers 307
out-of-order actions 315
See also xvc_xactor
stimulus 308
vs transactor 306
XVC manager 307, 316–319, 323, 325, 348
  instance 316
  predefined 317–319
  See also vmm_xvc_manager
See also xvc_manager
xvc_action 310, 312, 316, 442–444
  byte_pack() 443
  byte_size() 444
  byte_unpack() 444
  callbacks 314
  callbacks[] 443
  execute() 313, 443
  execution() 314
  get_name() 442
  max_byte_size() 444
  new() 442
  parse() 312, 443
  See also XVC, action
xvc_manager 316, 439–440
  add_xvc() 440
  log 439
  new() 440
  notify 440
  remove_xvc() 440
  See also XVC manager
  split() 440
  trace 439
  xvcQ[] 440
xvc_xactor 310, 312, 316, 440–442
  action-chan 441
  add_action() 441
  exec-chan 310, 313, 442
  interrupt-chan 442
  new() 441
  notify 441
  parse() 441
reset_xactor() 311
See also XVC
start_xactor() 311
stop_xactor() 311
trace 440
wait_if_interrupted() 315, 316, 442
xactors[] 310, 311, 314, 442
xvcQ[] 440
XVCTRACE 450
ABOUT THE AUTHORS

Janick Bergeron is a Scientist at Synopsys, Inc. He is the author of the best-selling book Writing Testbenches: Functional Verification of HDL Models and the moderator of the Verification Guild. He holds a Masters degree in Electrical Engineering from the University of Waterloo, a Bachelor of Engineering from the Université du Québec à Chicoutimi, and a MBA degree from the University of Oregon.

Eduard Cerny, Ph.D. (McGill University), is a Principal Engineer, R&D, in the Verification Group at Synopsys, Inc. He joined Synopsys in 2001 after 25 years in academia, as Professor of Computer Science at the Université de Montréal. His interests have been in design, verification and test of hardware, and he is author of many articles in these areas.

Alan Hunter, BEng(Hons), MSc, is the Design Verification Methodology Programme manager at ARM Ltd. and leads the design verification methodology work for ARM worldwide. This work covers all areas from CPU design verification through systems and system component design verification. His main areas of interest include optimizing design verification efficiency and quality, formal methods, and determinism in the design verification flow.

Andrew Nightingale, BEng(Hons), MBCS CITP, is a consultant engineer at ARM Ltd and has led the SoC Verification group in ARM’s Cambridge and Sheffield design centers for several years. The group covers ARM PrimeXSys platforms and PrimeCell development, including advanced AXI- and AHB-based system backplane components such as bus interconnects and high-performance memory controllers.