I Spy with My VPI:
Monitoring signals by name, for the UVM register package and more

Jonathan Bromley

Verilab
Glasgow, Scotland

www.verilab.com

ABSTRACT

Verification environments commonly need to react to value-changes on arbitrary DUT signals that are not part of a standard interface protocol. The package presented here supports both value probing and value-change detection for signals identified at runtime by their hierarchy name, represented as a string. This provides a useful enhancement to the UVM Register package, allowing the same string used for backdoor register access to be used also for value-change detection. It is also an interesting case study in the use of SystemVerilog DPI and VPI in the same package. For environments entirely coded in SystemVerilog, the package is completely portable. We also discuss how it can be applied to VHDL signals in a mixed-language environment.

Table of Contents

Table of Code Examples

1 Introduction

This paper describes a software package that provides useful new functionality for SystemVerilog verification environments.

Section 2 outlines the requirements that motivated development of the package. Section 3 describes some implementation challenges and how they were overcome, while section 5 gives fuller implementation details. Section 4 presents the package from a user's point of view.

In section 6 we examine the performance impact of using the package, and present some performance measurements using the VCS™ simulator. Section 7 indicates known limitations of the package, and section 8 discusses some desirable future enhancements.

2 Motivation

Users of the ***e*** [1] and Vera [2] hardware verification languages have the useful option of specifying DUT and test harness signal names as strings. These strings can be computed at runtime – for example, they can be read from a configuration file. Just as important, string names in a verification subcomponent can be specified relative to a point in the HDL hierarchy in a way that makes verification environment reuse much more straightforward than would otherwise be the case.

A SystemVerilog programmer, by contrast, has only limited access to such luxury. It is possible to read or write the value of a signal specified by its string name by using an additional VPI application (possibly with a DPI layer to make the user interface more attractive). Such an application has been distributed as part of the Universal Verification Methodology [3]. Unfortunately, though, this application does not allow detection of value-changes on a specified signal.

The combined SystemVerilog/DPI/VPI package presented in this paper provides an alternative solution, allowing SystemVerilog code to detect value changes on signals specified by their string path name. Sections 2.1 and 2.2 outline two of the more pressing reasons why this functionality is useful.

2.1 Active Monitoring in the UVM Register Layer

Almost every verification environment must deal with control and status registers in the design under test (DUT). The verification environment must have access to values in control registers so that it can correctly model the design's intended operation. Values in status registers reflect the current state of the DUT and its surroundings; these values, too, must be observed by a verification environment so that it can manipulate the DUT in an appropriate and meaningful way.

The UVM Register Layer, described in section 5 of the UVM User's Guide [4], provides comprehensive register modelling facilities for a verification environment. Using either explicit (bus monitor based) or implicit (register sequence based) prediction, each register's representation in the model is kept in step with the values that have been written into that register. However, this mechanism is not entirely satisfactory for status registers. Explicit prediction can update the model of a status register whenever a physical read is performed to the DUT's register, but this may not be timely enough to satisfy other modelling requirements. For example, if an interrupt occurs, the verification environment needs to know about it in order to predict the expected interrupt response. It is only *after* this response occurs that the testbench will detect an interrupt and, in due course, read the DUT's interrupt status register. The UVM register model, therefore, does not get the early warning it needs to predict an interrupt correctly.

This problem can be solved in the UVM Register Layer by the use of *active monitoring*. For each status register, a few lines of custom SystemVerilog testbench code use cross-module reference (XMR) to observe the relevant status signal. Whenever the status signal changes, a UVM Register Layer method is invoked to update the register model using back-door access. To do this, though, the user must specify the status signal twice: once as a hard-coded Verilog hierarchical path to provide the XMR, and once as a string property in the register model for use by the Register Layer's back-door access machinery, which works through the SystemVerilog VPI. This duplication is not merely inconvenient. Any XMR must be hard-coded, and therefore is not configurable in the dynamically created UVM verification environment.

A very similar problem exists in other register modelling frameworks such as RGM [5].

It would be much more satisfactory if the status signal's string pathname could be used for both purposes. The package presented here makes this possible.

2.2 Probing of Individual Signals

Although the automatic update of modelled status registers is probably the most important application for the new mechanism, it is also common to find DUTs whose operation depends on external mode or status signals that are not directly reflected in registers. A good example from the author's recent experience was an audio processing device whose synchronization was controlled by one or more system-wide audio data rate signals. These signals took the form of square waves running at a few tens of kHz, and were not reflected in registers.

As with the interrupt example, a verification environment needs to be aware of the values of such signals in order to model DUT behaviour adequately. Unfortunately there is no conventional, natural way to get such signal values into a UVM or other class-based verification environment. It is straightforward to observe signals by XMR from the test harness and reflect those signals into the verification environment using UVM events or some similar mechanism, but this leaves the user with two major problems:

- Any such solution is sure to be "home-grown" because the UVM does not clearly prescribe how it should be done. This lack of standardization makes it much less likely that diverse verification components will be interoperable.
- Such signal probing is not configurable from the verification environment. Each XMR path name must be hard-coded and is therefore fixed at elaboration time, whereas it would be configurable in the UVM verification environment if string-based signal access could be used.

The string-based signal access package described in this paper provides a configurable, uniform approach to the probing of arbitrary DUT signals from a verification environment.

3 Rushing In Where Angels Fear To Tread

The UVM 1.1a User Guide [4] insists that active monitoring must be done using custom SystemVerilog code because

"... there is no standard value-change callback VPI or PLI functionality ..."

The author, like many engineers, dislikes being told that something is impossible, and treats it as an implied challenge. A glance at the VPI documentation in the SystemVerilog LRM [6] shows that there is a well-defined value-change callback mechanism readily available through VPI routine `vpi_register_cb`. This routine allows a VPI application to specify a user callback function, written in C, that will be called automatically by the VPI whenever a chosen signal's value changes. It seems possible to prove the UVM User Guide wrong.

Callbacks can also be set on other events, but it is value-change callbacks that concern us here.

3.1 Optimism Gives Way to Frustration

The obvious next step would be to have the user's value-change callback function invoke a SystemVerilog DPI exported subprogram, in order to notify SystemVerilog user code of the signal change. Unfortunately, this is indeed impossible. To call a DPI export subprogram from C, it is necessary for a SystemVerilog DPI scope to be set, as described in Clause 35 and Annex H of the SystemVerilog standard [6]. This can only be done by the DPI itself when a DPI `import context` subprogram is called from SystemVerilog. A VPI user callback function has no SystemVerilog scope, and has no way to set that scope.

This appears to be a dead end. It seems that the UVM User Guide may be right: there is no way to call a SystemVerilog function from a VPI value change callback.

3.2 If All Else Fails, Cheat!

Happily, there is no such thing as a dead end in engineering – just another opportunity for a workaround. In this case, the workaround depends on a variable declared in a package. Because any SystemVerilog package variable is instantiated exactly once, it has a fixed unique name in the instance hierarchy and is easy to access both from VPI and from SystemVerilog code. Our VPI value change callback no longer attempts to do the impossible by calling a SystemVerilog function. Instead, our callback function records some information in a suitable C data structure and then uses VPI value get/set routines to toggle a package variable, which we call the *notifier*. This change of the notifier's value triggers the execution of some SystemVerilog code. The ensuing chain of activity, described in section 5.3.4, allows other SystemVerilog code to detect a value-change event on the original signal of interest.

4 User's View of the Package

4.1 Basic usage

Code Example 4-1 shows a trivial example of a UVM component that uses the package to probe a chosen signal and report every value change on that signal during the simulation run. Boiler-plate UVM code, such as the component registration macros and class constructor, has been omitted from this example for clarity. Code that is especially relevant to the discussion has been highlighted.

```
class sigWatcher extends uvm_component;
  string sigName;  // Full hierarchical name of the signal
  vlab_probes_pkg::signal_probe probe;  // Signal probe object
  function start_of_simulation_phase(uvm_phase phase);
    // Create the probe object and remember it for later use
    probe = vlab_probes_pkg::signal_probe::create(sigName);
  endfunction
  task run_phase(uvm_phase phase);
    forever begin
      probe.waitForChange();
      `uvm_info("PROBE",
                $sformatf("Signal %s has changed", sigName),
                UVM_LOW)
    end
  endtask
  ... // other standard UVM code: macros, constructor, ...
```

Code Example 4-1: Creating and using a probe

To probe a chosen signal[1], we supply that signal's full hierarchical name as the argument to function `signal_probe::create()`. This function creates a new object of class `signal_probe`, sets up that object to observe the chosen signal, and returns as its result a reference to the newly created object. We save that reference in a class data member of type `signal_probe`. The user's view of a probed signal is completely encapsulated in a `signal_probe` object. Once the object has been created, user code can call methods in that object to wait for a value-change on the probed signal, inspect the signal's value, and make other enquiries about the signal's properties. In this example, the signal's string name would be provided by UVM configuration.

Although it could be created at any time, we chose to create the probe object during our component's `start_of_simulation_phase`. This gives an opportunity for variable `sigName` to be populated during some earlier phase, typically `build` or `connect`. UVM configuration (the resource database mechanism) can of course be used to provide the name string's value.

Our `run_phase` task calls the probe's `waitForChange()` method. This stalls its execution until the next change of value on the probed signal. The task then displays a message before looping back to wait for a future change.

No other setup is required. The package needs some internal setup operations, but they are performed automatically by the very first call to `signal_probe::create()`.

[1] The signal can be a net or variable of any SystemVerilog *integral type* – any single bit (2-state or 4-state), packed array or packed struct, integer or 2-state integer-like type. Part-selects and bit-selects of any integral net or variable can also be probed.

4.2 *Other user-visible methods of `signal_probe`*

Once a probe has been created, various operations are available through the `signal_probe` object. In addition to waiting for a value change, the probe allows you to:

- read the value of the probed signal;
- enquire about various properties of the probed signal, including its bit width and signedness;
- disable and re-enable the detection of value changes at any time;
- artificially trigger the probe's internal value-change event, releasing any threads that are waiting at `waitForChange()` task calls.

These additional features impose no overhead if they are not used. They are all straightforward and are not described further here. Further information on them can be found in the package's own documentation.

4.3 *Error handling*

Various error conditions are detected and flagged by the package.

4.3.1 Failure to create a requested probe

The only error condition that a user of the package should encounter in practice is failure to create a probe. This can happen for several reasons:

- the requested signal does not exist because its string name is incorrect;
- the string name specifies some object in the Verilog hierarchy that is not an observable net or variable of integral type (for example, real variables and unpacked array variables cannot be probed);
- inappropriate command-line options have been specified to the simulator, making the requested signal unreachable from the VPI because of the effect of optimizations.

In any of these cases, warning messages are displayed from both C and SystemVerilog parts of the package, and a `null` object handle is returned from the `create` method. User code should detect this `null` result and respond appropriately.

In an early version of the package this condition was treated as an error that caused simulation to stop. In response to beta-test user feedback, the package was modified to behave in the more tolerant manner described here.

4.3.2 Internal errors

Internal consistency checks are performed at various stages of the package's operation. These checks were originally added during development to give as much information as possible about the author's own coding errors. They have been left in the code as a safeguard against inconsistent internal data structures, as they have an insignificantly small runtime overhead.

These consistency checks protect against faulty user code that interferes with the package's internal operation, or hitherto undetected bugs in the package itself. If any such check fails it is treated as a fatal error, displaying a diagnostic message and halting the simulation.

4.4 Compilation

To use this package, you must compile the file `vlab_probes_pkg.sv` along with other SystemVerilog verification code. Since this package does not depend on any other SystemVerilog code, and is likely to be used by various parts of a testbench, it should appear as early as possible in the compile list.

The DPI and VPI code in file `vlab_probes.c` must be compiled using the chosen simulator's normal DPI flow. In the case of Synopsys's VCS simulator [7] this is rather easily achieved by including that file in the source file list of the `vcs` command line, but it is also important to include these command line options:

```
+vpi +acc+2
```

These options instruct VCS to enable VPI access, and to defeat certain optimizations that might prevent VPI routines from gaining access to DUT signals. The resulting performance overhead is discussed in section 6.

5 The Package In More Detail

5.1 SystemVerilog Package and C Source File Structure

The entire package source code is delivered as exactly two files, `vlab_probes.c` and `vlab_probes_pkg.sv`, containing about 650 lines of C code (including comments) and 300 lines of SystemVerilog.

The SystemVerilog code appears to the user as a single package `vlab_probes_pkg` (the vendor name prefix has been added to reduce risk of name collisions in SystemVerilog's completely flat package namespace). This package contains exactly one declaration, of class `signal_probe`. It imports package `vlab_probes_pkg_private`, but does not re-export it (see section 5.5).

5.2 Methodology Neutral

Although the original motivation for this package was that it should be used with the UVM Register Layer, it is clear that it may be useful in other contexts. It was therefore written as a plain SystemVerilog package and class, without using base classes or other features from any published methodology library. Because its class structure is so simple, with only one user-accessible class, it should be easy to provide wrappers for it to suit the needs of any base class library including UVM, VMM and proprietary libraries. All user-accessible methods of the class are declared `virtual` and therefore could be overridden in a derived class if necessary.

5.3 How the Package Works

Rather than presenting the package code in full here, we focus on how a few key problem areas were tackled. For details of the various VPI routines that are used in the package, consult the VPI reference Clause 36 of [6], or a PLI/VPI reference book such as [8]. Section 5.4 describes some aspects of the C DPI/VPI code.

The SystemVerilog part of the package is generally straightforward. Its overall behaviour is described in section 5.3.4.

5.3.1 *One SV object per probed signal*

Each probed signal is represented by an instance (object) of SystemVerilog class `signal_probe`. This object, and a companion data structure maintained in the C code, are created when a user calls `signal_probe::create`, discussed in a later section. Once created, this object encapsulates all the user's interaction with the probed signal. A user can create as many such objects as are needed, one for each probed signal. It is also acceptable to create more than one probe on a given signal; all such probes operate independently and, for example, can be disabled or enabled individually without affecting the others (see section 4.2).

5.3.2 *DPI interaction must be package-global*

SystemVerilog's DPI does not support the exporting of SystemVerilog class methods. Consequently, for C code to invoke a method of a SystemVerilog class, it is necessary for the C code to call a package-global SystemVerilog function exported via DPI, passing an argument value indicating in which object the method should be called. The package function can then determine the target object, and call the appropriate method in it. In the package described here, the argument value is an integer index into a queue of objects.

5.3.3 *SV code waits for a value change*

If user SV code wishes to wait until a probed signal changes its value, it should call task `waitForChange` in the corresponding object. This task is very simple:

```
task signal_probe::waitForChange();
  @change;
endtask
```

Code Example 5-1: Implementation of `waitForChange`

Each object contains an event named `change` that is used to indicate value-changes on the probed signal. Of course, the interesting question is how this `change` event is fired.

5.3.4 *Outline of value-change notification flow*

When a signal probe is created as described in 5.3.1, a VPI value change callback is attached to that signal. This value-change callback is automatically invoked whenever the signal's value changes. From that point forward, the flow of activity is:

- VPI callback causes a notifier bit in the SV package to be toggled. There is precisely one notifier bit, common to all signal probe objects.
- The notifier bit's change of value is detected by the following code in the SV package:

```
forever begin
  @notifier;
  vlab_probes_processChangeList();
end
```

- Function `vlab_probes_processChangeList` is known as the *enquirer* function. It is a DPI imported C function. This enquirer function then determines which probed signal was responsible for the value change, and calls back to SystemVerilog via the DPI export function `vlab_probes_vcNotify(`*`sv_key`*`)`.

- The *`sv_key`* argument is a unique integer identifier that allows the SystemVerilog code to find the corresponding signal probe object quickly. It is in fact the index into a queue of such objects, `probes_by_key`. The SystemVerilog implementation of this function can then call a function in that object:

```
probes_by_key[sv_key].releaseWaiters();
```

- The implementation of `releaseWaiters` is trivial, merely firing the object's `change` event and so releasing any threads waiting in task `waitForChange`:

```
function void signal_probe::releaseWaiters();
  -> change;
endfunction
```

The remainder of section 5 describes some aspects of the code in more detail.

5.4 The DPI/VPI C code

5.4.1 Finding a signal by its string name

SystemVerilog's VPI provides a routine `vpi_handle_by_name` that can find any signal in the hierarchy using its string XMR pathname. The C function `vlab_probes_create` is called via DPI to do this:

```
void * vlab_probes_create(char *name, ...) {
  vpiHandle obj;
  // Locate the chosen object
  obj = vpi_handle_by_name(name, NULL);
  if (obj == NULL) {
    vpi_printf(
     "*W,VLAB_PROBES: create(\"%s\") could not locate signal",
     name);
  ...
```

Code Example 5-2: Using the VPI to locate a signal by name

Note that the VPI routine `vpi_handle_by_name` returns a result of type `vpiHandle`. This handle is stored in variable `obj` for later use.

5.4.2 A data structure for each signal

For each signal, a C `struct` is created that holds all necessary information. The critical pieces of information are shown here:

```
typedef struct t_hook_record {
   vpiHandle obj;          // reference to the monitored signal
   int sv_key;             // unique key to help SV find this
   vpiHandle cb;           // VPI value-change callback object
   ...                     // other housekeeping fields
} s_hook_record, *p_hook_record;
```

Code Example 5-3: C data structure representing a signal

The first `obj` field will be populated with the handle we obtained in Code Example 5-2. `sv_key` is a unique integer identifier that is shared between C and SystemVerilog. It helps to identify probed signals efficiently by associating each probe with an element of a SystemVerilog queue[2].

Finally, `cb` is a VPI handle to the value-change callback object. We discuss the creation of this callback in section 5.4.3 below.

In the final implementation of the package, additional housekeeping fields have been added to this struct in order to make various common operations more efficient. In particular there are:

- pointer fields that allow these structs to be assembled into linked lists;
- a self-reference pointer field, for internal consistency checking;
- fields containing static properties of the probed signal (bit width, signedness), populated when the probe is created, to avoid repeated VPI calls to find out this information.

5.4.3 Getting a value-change callback on a probed signal

Having obtained and saved a VPI handle to the probed signal, we must place a value-change callback on it by calling the `vpi_register_cb` routine. This routine's single argument points to an `s_cb_data` struct that describes details of the callback. Before calling `vpi_register_cb` we must create this struct and populate its fields with various pieces of information:

- a copy of the probed signal's `vpiHandle`, so that the VPI knows which signal should get the new value-change callback;
- an integer constant `cbValueChange` specifying what kind of callback is to be set up;
- a pointer to our C function `vc_callback` that we want to be called when the value-change occurs;
- a `user_data` field that allows our callback function to identify which signal caused the callback.

In the code fragment below, `hook` is a pointer to the newly constructed `s_hook_record` struct, which already contains the signal's handle in its `obj` field.

[2] The `sv_key` field is not strictly necessary. It would be more elegant to use a C pointer to the hook record itself to provide this unique identifier, exposing it to SystemVerilog as a `chandle` variable. The mechanism described here, based on integer keys, is slightly more efficient (albeit at the expense of some additional housekeeping in the data structures) and avoids reliance on simulator support for associative arrays indexed by `chandle`.

```
...
s_cb_data cb_data;
cb_data.reason = cbValueChange;
cb_data.cb_rtn = &vc_callback;
cb_data.obj = hook->obj;
cb_data.user_data = (PLI_BYTE8*)hook;
hook->cb = vpi_register_cb(&cb_data);
...
```

Code Example 5-4: Creating a value change callback on a signal

The callback is now in place, and every value change on the probed signal will cause a call to our function `vc_callback`.

5.4.4 Multiple Simultaneous Value Change Events

This approach requires some special care because of the possibility of multiple value-changes in the same SystemVerilog timeslot. If we toggle our notifier variable on every value-change callback, there can be a problem if more than one signal changes at the same moment of simulation time. In particular, if there is an even number of changes at a given time, the notifier will toggle back to its original value and there is a possibility that the event control `@`*`notifier`* might not be released at all.

Fortunately, it is straightforward to deal with this by establishing a request-acknowledge protocol in which toggling the notifier acts as a request from C to SystemVerilog, and acknowledgement is provided when SystemVerilog calls the enquirer function in C. The first value-change callback in a timeslot causes the notifier to toggle. Subsequent value-change callbacks do not toggle the notifier, but instead are merely logged in a data structure. This behaviour continues until SystemVerilog responds to the notifier toggle and calls the enquirer function, which reads and then resets all the data structures so that the whole process can begin again on some future value-change. In this way the package correctly handles any number of simultaneous value-change events.

5.4.5 Handling the value change callback

The implementation of `vc_callback` is surprisingly simple. It makes use of a static variable of the package, `changeList`, which points to a linked list of structs representing signals that have had a value-change but have not yet been serviced by the SystemVerilog enquirer function. If this list is non-empty, we know that the notifier signal has already been toggled and SystemVerilog will in due course call the enquirer, so there is no need to toggle the notifier again. But if the list is empty, we need to toggle the notifier. In either case, we add our signal to the list.

```
static PLI_INT32 vc_callback(p_cb_data cb_data) {
  p_hook_record hook = chandle_to_hook(cb_data->user_data);
  // Is this the first value change?  If so, we must notify it.
  int require_notification = (changeList == NULL);
  // Put this object on the changelist, if it isn't already.
  changeList_pushIfNeeded(hook);
  if (require_notification) {
    // Toggle the notifier bit.
    int ok = toggle_notifier();
    return ok;
  } else {
    return 1;
  }
}
```

Code Example 5-5: The value change callback

5.4.6 Locating and toggling the notifier

In addition to probing the user's signals, our VPI code must also interact with the package's notifier signal. During initialization of the package we obtain a reference to the notifier signal using `vpi_handle_by_name`, but in this case the signal's string name is a well-known constant because the notifier signal is a variable of the package. The resulting object handle is stored in a static `vpiHandle` variable, `notifier`, in the C code. Package initialization code is not discussed in detail here.

To toggle the notifier signal we must use the VPI first to get the notifier's current value, and then to write back the logical negation of that value. This is a straightforward VPI value get/set operation. A local `s_vpi_value` struct holds the VPI value representation for both getting and setting. We can safely treat the notifier as a `vpiScalarVal` (single-bit logic value) because we know that it has been set up as a single `bit` by the SystemVerilog package.

```
static PLI_INT32 toggle_notifier() {
  s_vpi_value value_s;
  value_s.format = vpiScalarVal;
  vpi_get_value(notifier, &value_s);
  value_s.value.scalar = (value_s.value.scalar==vpi1)? vpi0: vpi1;
  vpi_put_value(notifier, &value_s, NULL, vpiNoDelay);
  return 1;
}
```

Code Example 5-6: Toggling the notifier

5.5 *Draco dormiens nunquam titillandus*[3]

There is an intimate and carefully managed relationship between the SystemVerilog and C (DPI/VPI) components of the finished package. It would be disastrous for user code to interfere with that relationship directly. The package should be used only through its published application programming interface (API). However, in SystemVerilog it is impossible to hide the contents of a package from users' view. It is therefore very important that the package's user API should be kept entirely separate from its internal implementation. This was achieved by bundling almost all the internal SV code, including all the DPI export and import function prototypes, in a package named `vlab_probes_pkg_private`. Although a perverse user could access this private package, it is intentionally undocumented and its name was chosen to discourage casual meddling. There is no need for users to import it into their code. User code should import only the top-level package `vlab_probes_pkg`, which contains the user API class `signal_probe` and nothing else.

6 Performance Considerations and Measurements

6.1 *Portability and Standards Compliance*

The package has been tested on recent versions of several major SystemVerilog simulation tools, and works correctly in all of them with no source code changes. It is believed to be fully compliant with the SystemVerilog 2009 standard [6].

In mixed-language environments it would be desirable to probe not only SystemVerilog signals, but also signals in VHDL design blocks. As described in section 8.1, this is sure to be tool-specific because the necessary VPI extensions are not standardized.

6.2 *Why Performance is a Concern*

The most important criteria for design of this package were usability and performance. Its development was prompted by a colleague's need for value change detection in a UVM Register Layer model with many thousands of registers. In such a large environment, poor performance would be an unacceptable price to pay for the convenience of string-based probing.

Inevitably, the use of VPI value change detection carries some overhead. The rather complicated control flow described in section 3.2 presents a risk of even worse performance penalty than might be anticipated. To mitigate these concerns, great care was taken to ensure that not only is the implementation as efficient as reasonably possible, but also that it scales well with increasing problem size. The next few sections examine some of the runtime costs in more detail.

6.3 *Overhead Caused by Defeating Optimizations*

When a signal specified by its hierarchical name is passed as an argument to a SystemVerilog task such as `$monitor(top.middle.lower.signal)`, the simulator knows that this signal will be accessed through VPI. It therefore defeats optimizations that might cause the signal to disappear from the simulator's internal data structures. All other parts of the simulation can be optimized as usual.

[3] The Latin motto of Hogwarts, fictitious wizardry college of Harry Potter fame [9].
A sleeping dragon should never be tickled.

By contrast, when signals are specified by their string name, the simulator has no way of knowing which signals will be accessed during the run. For such string-based access to work, the user must specify command line options to make *all* signals accessible from VPI routines. As previously mentioned, in VCS this is achieved by using command-line switches `+vpi +acc+2`. Inevitably this imposes a performance penalty on the whole simulation. Our measurements suggest that the performance penalty is typically around 10% to 15%. However, this penalty is not specific to the package described here. If *any* string-based signal access is needed (for example, for the UVM Register Layer's existing back door HDL access mechanism) then this overhead is already present, and the package described here imposes no additional cost.

6.4 Run-Time Cost per Value Change Event

Each value-change event on any probed signal gives rise to a VPI callback and various internal operations in the package. This carries a cost that is directly proportional to the number of events, even if there is no user thread waiting on those events. For this reason it is better to avoid probing signals such as fast clocks that will change very frequently during simulation, as this would give rise to very large numbers of events with a run-time penalty exacted for each of those events.

To put this into perspective, tests on a modest Linux workstation showed that each probed value change event was imposing a runtime overhead of one or two microseconds. For a simulation with 1000 probes each of which has 10000 value changes during the simulation run, that would lengthen the simulation runtime by ten or twenty seconds. This seems reasonable, and is unlikely to represent a major slowdown in a large simulation. Nevertheless, it must be borne in mind that this is the dominant performance cost associated with the package and it could represent a severe penalty if large numbers of rapidly changing signals were probed.

Part of the per-event overhead is attributable to the cost of toggling the notifier variable, and the subsequent DPI call from SystemVerilog to the enquirer function, as described in section 3.2. If more than one probed signal has a value change at the same moment of simulation time, this price is paid only once for all those value changes. This will slightly reduce the total cost in simulations where many probed signals change together.

6.5 Run-Time Cost per Waiting Thread

When a thread calls a probe's `waitForChange` method, it is internally stalled at a Verilog `@` event control inside the method, waiting for an event to be fired. The event in question is a member of the probe class. Consequently, it is reasonable to anticipate that the cost of stopping and restarting the thread at this event control would be identical to the cost of waiting at any other Verilog event control, and therefore is no worse in this package than for any other approach. Our measurements confirmed this to be the case for VCS.

6.6 Simulator Memory Cost per Probed Signal

Each created signal probe causes the simulator to allocate one SystemVerilog object (the `signal_probe` object) with a memory footprint of a few tens or hundreds of words, plus the memory required for a copy of the signal's string name. It also causes the allocation of one entry on a SystemVerilog queue, holding a reference to the object.

In the C code, each probed signal causes a `struct` of about 50 bytes to be allocated in the C code, and also creates two VPI objects – one referencing the signal itself, and another for the value-change callback.

The resulting memory cost, perhaps one or two kilobytes per probed signal, is believed to be acceptable even for the largest problems with many thousands of probes.

6.7 Run-Time Cost per Probed Signal

The time required to perform a `create` operation on a probe appears to be insignificant in the context of a typical simulation, with many thousands of probes created in only a few seconds of simulation runtime.

If we assume that the time needed to access an element of a SystemVerilog queue is independent of the size of the queue, then we can also show that increasing the number of probed signals has *no impact whatsoever* on the cost of any individual operation in the package. This constant-time property has been achieved by careful design of the package's data structures both in C and in SystemVerilog.

Naturally, in a simulation with a large number of probed signals it is likely that there will be a correspondingly large number of value change events. This will inevitably increase the runtime cost. However, the runtime cost of each value change event is *not* affected by the total number of probes in use.

6.8 Dynamically Enabling and Disabling Value Change Detection

When a probe has been created, it can be disabled or enabled at any time using the probe object's method `setVcEnable`. This can be used to improve performance if a given probe is not required during part of the simulation, because a probe imposes no runtime cost per value-change event when it is disabled. However, this feature should be used with caution because the disable and enable operations carry a non-trivial cost. Detailed measurements have not yet been performed on this issue, but it seems likely that the cost of a disable/enable operation is comparable with the cost of around 10 to 100 value-change events.

6.9 Miscellaneous Optimizations

Considerable care was taken to minimize the number of cross-language function calls between C and SystemVerilog. Although the DPI's call interface is reasonably efficient, it can require some format conversion of function arguments and therefore the number of calls should be minimized. This consideration has led to the duplication of some information in C and SystemVerilog data structures so that each side can, as far as possible, find the information it needs without needing to make additional DPI calls across the language boundary. The duplication is transparent to users, who see only the public API of the `signal_probe` class. Internal assertion checks are used to protect against possible corruption of these data structures, which might occur if a careless or malicious user were to abuse the private package or its DPI import functions.

6.10 Experimental Results with VCS™

A test was set up in which a configurable number of instances of a signal was created, with that signal toggled at varying intervals by ordinary Verilog code. The simulation was run long enough to allow about 2000 such events to occur on each signal. Three different test configurations allowed comparison of runtime with:

- *setup_1*: no detection of the events;
- *setup_2*: detection of the value changes using a regular Verilog cross-module reference:

```
always @top.middle.low.signal
  counter++;  // chosen to consume as little time as possible
```

- *setup_3:* detection of the value changes using the signal_probe package, as described in this paper:

```
always begin
  probe.waitForChange();
  counter++;
end
```

The difference in execution time between configurations 2 and 1 provides a reference to standard Verilog code by estimating the time cost of detecting the events in a conventional way. The cost of using the `signal_probe` package was gauged by computing the ratio

(*setup_3* – *setup_1*) / (*setup_2* – *setup_1*)

This ratio was found to vary between 2 and 10, depending on a variety of factors. It is difficult to measure such values accurately, so the experiment was run many times with different numbers of signal instances (lowest 100, highest 2000). The spread of measured ratios was within +/- 10%, leading to reasonably good confidence in the measurements.

Although this performance degradation appears burdensome, it should be noted that it applies only to the actual detection of events. Adding even modest amounts of other activity to the code triggered by each event caused this ratio to fall dramatically. The penalty associated with the package is believed to be acceptably small.

Finally, the cost of creating each probe object must be considered. In practice this cost was found to be acceptably small, with many thousands of probes created in one second of simulation runtime.

7 Limitations

7.1 Methodology-specific Messaging

Certain warning messages are displayed directly from the VPI code using `vpi_printf()`, because the necessary diagnostic information is available there and the overhead of getting that information into SystemVerilog would be unreasonably large. Unfortunately, such messages will inevitably bypass any methodology-specific messaging mechanism such as the UVM report handler. This is not a disastrous limitation, because information about the existence of the warning is also available in SystemVerilog. Consequently, class `signal_probe` could be extended using methodology-specific messaging to draw attention to the original diagnostic from VPI code.

7.2 Simulator Checkpoint Save and Restore

For any VPI or DPI package, there is a serious problem if the simulator attempts to restore a saved checkpoint. The simulator's saved state does not include the state of persistent data structures within the C application. To restart successfully from a saved checkpoint, it would be necessary for the signal probe package to save its own internal state to some file, and automatically restore that state when the simulator loads the corresponding checkpoint file.

This problem is common to many VPI/DPI applications, including the signal probe package described here. Simulator restart to time zero is correctly handled, but checkpoint save/restore is not currently supported. This is a challenging problem for which there is no obvious solution.

8 Further Work

8.1 Mixed-Language Designs

All major SystemVerilog simulation tools also support mixed-language VHDL/SystemVerilog simulation. The author believes that VCS can also support VPI named signal probing across the language barrier, and aims to extend the package in the near future so that mixed-language probing works with the same user-facing API.

8.2 Integration with Published Register Modelling Packages

As noted in section 5.2, the package is completely methodology-neutral. However, there are some interesting opportunities to integrate it more tightly into the UVM Register Layer and perhaps other similar packages, improving automation and reducing the integration burden for users.

9 Availability

The source code can be downloaded from `www.verilab.com` without restriction – simply pick *Other Downloads* from the *Resources* menu on the home page. The author will always try to respond to queries and bug reports relating to it, but has to earn his living and therefore cannot guarantee a response. Equally, Verilab and the author will be glad to hear of suggestions for improving the package.

10 Conclusions

A package of SystemVerilog and DPI/VPI C code has been developed that allows users to detect value changes on simulated signals identified by their string name in the Verilog hierarchy. The package is easy to use and has acceptable performance overhead. It provides some hitherto unavailable functionality that is valuable for active monitoring in UVM register models, and for various other applications in class-based SystemVerilog verification environments. The package is freely available and has already been deployed in a production verification environment.

11 Acknowledgements

The author wishes to thank his colleagues at Verilab for their support and suggestions, and their reviews of drafts of this paper. Special thanks go to Mark Litterick and his project team for providing clear requirements and valuable feedback for early versions of the package.

Thanks also to my former colleague Dr David Long of Doulos who acted as technical reviewer, Doug Fisher of Synopsys for his encouragement, and Adiel Khan of Synopsys who carried out many of the performance measurements and made valuable suggestions for performance optimization with VCS.

12 References

[1] IEEE, Standard 1647 for the Functional Verification Language e, Piscataway, NJ: IEEE, 2008.

[2] F. Haque, K. Khan and J. Michelson, The Art of Verification with Vera, Fremont, CA: Verification Central, 2001.

[3] Accellera Systems Initiative, “UVM (Universal Verification Methodology),” [Online]. Available: http://www.accellera.org/downloads/standards/uvm.

[4] Accellera Systems Initiative, “Universal Verification Methodology (UVM) 1.1 User's Guide,” Accellera, 2011.

[5] Cadence Design Systems, Inc, “Cadence UVM_RGM2.5 Release,” [Online]. Available: http://www.uvmworld.org/contributions-details.php?id=101.

[6] IEEE, Standard 1800-2009 for SystemVerilog Hardware Design and Verification Language, Piscataway, NJ: IEEE, 2009.

[7] Synopsys, Inc, *Chronologic VCS (TM) simulator,* Synopsys, Inc, 1991-2012.

[8] S. Sutherland, The Verilog PLI Handbook, 2nd ed., Norwell, MA: Springer, 2002.

[9] J. K. Rowling, Harry Potter and the Philosopher's Stone, London: Bloomsbury, 2001.