UVM Interactive Debug Library: Shortening the Debug Turnaround Time

Horace Chan
Microsemi Corporation
8555 Baxter Place, Burnaby,
BC, Canada, V5A 4V7

Abstract - **Interactive debug in System Verilog (SV) simulations is not a natively supported feature, unlike other Hardware Verification Languages (HVL) such as Specman *e*. In this paper, we are presenting an interactive debug library for UVM[1] implemented in the SystemVerilog Direct Programming Interface (SV-DPI)[2]. At a basic level, this enables high-level interactive debug at simulation run time. The features of this library includes: 1) writing or reading a register through the UVM register abstraction layer, 2) creating, randomizing, initializing, and starting a UVM sequence on any sequencer, and 3) calling a user-defined SV function or task from the interactive debug prompt. We are also presenting a debug and regression methodology outlining the best practice for how to speed up debug and minimize the regression run time. The preliminary results show that using the interactive debug library significantly reduced the debug turnaround time and regression run time. Users can retest different scenario in matter of seconds instead of waiting minutes or hours for the testbench to recompile, elaborate, and rerun the simulation from time zero. The uvm_debug library is available as an open-source project in github[3].**

Keywords **– SystemVerilog, Verification, UVM, SV-DPI, interactive, debug, open source**

I. INTRODUCTION

Users of verification frameworks implemented in other programming languages, (such as Specman ***e***[4] and Cocotb/Python[5]), have been accustomed to the flexibility of interactive debug during run time. The user can run the simulation to a certain point in time, pause the simulation, activate the debug prompt, and then call any testbench functions to change the behavior of the test scenario before resuming the run. In SystemVerilog, due to the limitation of the language, once a testbench is compiled, the user has little control over the test scenario during the simulation. Currently, during simulation run time, users are limited to using a difference seed or poking/forcing signals inside the simulation environment.

Imagine the following scenario: a user has a long simulation that runs for hours. Halfway into the simulation, a testbench checker reports an error. If the user needs to change the stimulus in order to debug the problem, then they have to update the SV code, recompile the testbench, and wait a long time to find the error again. With the help of the interactive debug library, the user can try different debug scenarios without restarting the simulation. They can query the device status through register reads, try new device configurations through register writes, change the stimulus generated by the testbench by executing a new UVM sequence with a higher priority, or they can call any testbench functions to help diagnose the problem. They can try out dozens of different scenarios and pinpoint the bug in a matter of minutes instead of hours. The drastic reduction in debug turnaround time increases productivity, and the time is spent on isolating the bug, rather than waiting for the simulation to compile/run.

An earlier attempt, cli_seq_pkg [6], was made by M. Peryer to support interactive debug in UVM by introducing an command line debug interface. The biggest drawback of the cli_seq_pkg is that it requires a wrapper class for each sequence. Each wrapper class is created manually or generated by a preprocessing scripts in two phases - either solution is tedious and prone to error. Moreover, the author did not provide the source code of cli_seq_pkg and the description in the paper did not contain enough details to allow users to re-implement the package with ease.

Another attempt, RESSL [7], was made by J. McNeal and B. Morris to implement a read, evaluate, start sequence loop. Its implementation is highly intrusive to an existing testbench. RESSL requires a hacked version of UVM. All sequences supported in the read, evaluate loop must be derived from a common base class and manually

registered into a sequence registry in the testbench. These requirements make adoption in a large-scale production project very difficult unless there is a strong mandate from management to drive these changes throughout the project.

The UVM Interactive Debug Library (uvm_debug), presented in this paper was developed independently, without prior knowledge of [6][7]. The uvm_debug library supports all the functions of cli_seq_pkg and indirectly supports all features of RESSL, but offers much more. The author wanted to bring the powerful debug features and flexibility from Specman into SV-UVM so that multiple SV testbench environments can benefit from it. The uvm_debug library is available as an open-source project from github[3]. Setting up the uvm_debug library is easy and non-intrusive: simply download the code, then compile the .sv and .c file together with the testbench. The user does not have to go through a complicated testbench hook up process or any patch into the UVM source code. All it takes is calling a simple function to initialize the uvm_debug object.

II. Overview of UVM Interactive Debug Library

A. *The Core Logic*

The core of the interactive debug library is remarkably simple. SV lacks any language syntax to read a user input when the simulator is running. Fortunately, SV can call C code through the SV-DPI interface, and so user input can be read in the C domain and then passed back to the SV domain.

In the SV code, the function dpi_read_line() in uvm_debug_pkg prints a prompt to the screen, waits for the user to enter a line, and then returns the inputted line as a string. In the C code, the simplest implementation is to call the C built-in getline() function from <stdio.h> to read a line from the standard input. In a case where the simulator is running in GUI mode, it is quite inconvenient to switch back and forth between the shell and the GUI to enter in debug commands. Alternatively, the C code can be compiled to read the user input through the Tcl prompt of the simulator.

SV code

```
import "DPI-C" function void dpi_read_line(string prompt, inout string line)
function string read_line(string prompt);
    dpi_read_line(prompt, sbuffer);
    read_line = sbuffer;
endfunction: read_line
```

C code

```
char buffer[1000];
void dpi_read_line(char* prompt, char** line) {
     #if defined(CADENCE)
        *line = cfcReadline(prompt);
     #elsif defined(MENTOR)
         mti_AskStdin(*line, prompt);
     #else
          printf(prompt);
          fflush(stdou);
          *line = &buffer;
          int bufsize = sizeof(buffer) / sizeof(char);
          getline(&buffer, &bufsize, stdin);
     #endif
};
```

Once the testbench can read a user input in simulation, the rest is fairly straightforward. All objects and components created in UVM can be accessed by their names, which is a text string. The debug library implements a command line parser to process the user input, query the object pointer using the full name, and then call the specified task or function using the parameters supplied by the user.

B. Starting the Debug Prompt

There are two ways to activate the interactive debug prompt. The first is to embed a function call to the uvm_debug object in the SV code. The uvm_debug object is a singleton class, so it can be used anywhere in the testbench or testcases without worrying about the need to keeping track of different instances. The user can wrap the `prompt()` function in an if-else statement to specify any triggering condition, such as triggering the debug prompt when the testbench detects a DUT error or receives some event flags. The `prompt()` function takes in a debug-level argument, which allows the user to control when to activate the debug prompt through an UVM command line argument, `+debug_level=<debug level>`. The debug prompt will only activate if the debug level in the simulation is higher than the debug level specified in the `prompt()` function, a similar idea to UVM verbosity. The second way to trigger the debug prompt is to pause the simulation and then call the debug_prompt VPI system function from the Tcl prompt.

SV code

```
import uvm_debug_pkg::*;
...
uvm_debug_util uvm_debug = uvm_debug_util::get();
uvm_debug.prompt(1);
```

Tcl commands

```
ncsim> call debug_prompt
debug prompt (help for all commands)
1000ns: debug >
```

C. Housekeeping Debug Commands

The debug prompt supports the following built-in debug commands to provide basic housekeeping functions.

Table 1 Housekeeping Debug Commands

Command	**Description**
`help [command]`	Display the help message. With no arguments, it lists all the available commands; otherwise, it displays the help message of the command specified.
`continue`	Exit the debug prompt and continue the simulation.
`pause`	Pause the simulation and switch to the simulator's Tcl Prompt.
`run <runtime>`	Run the simulator for the specified time and go back to the debug prompt
`.history [list|clear|save <file>`	List all the previous command inputted by the user; clear the history; save the history to a command file.
`repeat #`	Repeat the specified line in the command history.
`read <file>`	Read a command file.
`save_checkpoint [-path <path>] <snapshot name>`	Save a checkpoint snapshot.

D. Register Debug Commands

Before performing register operations, the user has to initialize the uvm_debug library by passing in the root uvm_reg_block to set up the relative uvm_reg hierarchical full name used by the debug commands.

SV code

```
uvm_debug.reg_util.set_top(my_top_reg_block);
```

The debug prompt supports the following built-in uvm_reg-related debug commands.

Table 2 Register Debug Commands

Command	Description
`wr_add <addr> <value>`	Write a register by address.
`rd_addr <addr>`	Read a register by address.
`wr_reg <reg> <value>`	Write a register by the uvm_reg full name.
`rd_reg <reg>`	Read a register by the uvm_reg full name.
`wr_regfld <field> <value>`	Write a register field by the uvm_reg_field full name.
`rd_regfld <field>`	Read a register field by the uvm_reg_field full name.

E. Sequence Debug Commands

The debug prompt supports the built-in uvm_sequence-related debug commands listed in the following table. The debug prompt keeps track of all sequences and sequence items created by the user in an associative array. The user can always refer to the created sequence or sequence item using the name specified during its creation. The sequence debug commands operate on the UVM sequence base class - there is no dynamic typing checking. If the sequence or sequence item does not match the required data type of the sequencer, the simulator may have a fatal error. The sequencer path used by the debug prompt is the uvm_component full hierarchical name of the sequencer relative to uvm_top_test. A sequence or a sequence item can be run in blocking mode or in a new thread. In blocking mode, the simulation time advances until the sequence or sequence item is done then returns to the debug prompt. When running in a new thread, the sequence or sequence item is runs in the background inside a fork..join_none block; it will return to the debug prompt after the command is dispatched.

Table 3 Sequence Debug Commands

Command	Description
`seq_list`	List the sequences created in the debug prompt.
`seq_create <seq_type> <seq_name>`	Create a new sequence.
`seq_rand <seq_name>`	Randomize the sequence created in the debug prompt.
`seq_set_fields <seq_name> [<field>=<value>...]`	Set values of the fields in sequence created in the debug prompt.
`seq_start [-priority <priority>] [-new_thread <0|1>] <seq_name> <seqr_path>`	Start the sequence in a sequencer.
`seq_kill <seq_name>`	Kill a running sequence.
`seq_item_list`	List the sequence items created in the debug prompt.
`seq_item_create <seq_item_type> <seq_item_name>`	Create a new sequence item.
`seq_item_rand <seq_item_name>`	Randomize the sequence item created in the debug prompt.
`seq_item_set_fields <seq_item_name> [<field>=<value>...]`	Set values of the fields in the sequence item created in the debug prompt.
`seqr_stop_sequences <seqr_path>`	Stop all the sequences in the sequencer.
`seqr_execute_item [-new_thread <0|1>] <seqr_path> <seq_item_name>`	Execute a sequence item in the sequencer.

F. User Defined Custom Debug Commands

In addition to the built-in debug commands that come with the uvm_debug library, users also have the ability to create custom debug commands by creating a debug command callback function, in which the user can call any function or task in the testbench. The user does not have to register the custom debug command explicitly; the only setup required is to create the debug command call back object. The built-in constructor of the debug command call back class automatically registers the debug command with the uvm_debug singleton object. The command registration to the debug prompt is handled in the constructor of the debug command callback base class. The following code snippet shows how to define a sample custom debug command.

SV code

```
class custom_debug_command extends uvm_debug_command_cb;
    function new(string name = "custom_debug_command");
        super.new(name);
        // ---------------------------------------------------------------
        command =       "cmd_name";
        usage =         "<arg1> <arg2>";
        description =   "description of the command showed in help";
        // ---------------------------------------------------------------
    endfunction

    task parse_args(string args[$]);
        // parse the arguments and call the testbench function/task
        ...
        // set the return value (string)
        uvm_debug_util.rv = ...
    endtask
endclass: custom_debug_command
```

The input arguments from the debug prompt are passed into the callback object as a list of strings. To help the user parse the argument list before calling the testbench function/task, the uvm_debug library comes with a built-in argument parser. The argument parser supports two argument formats, the tcl dash option format (`-option value`) and the SV command line plus argument format (`+option=value`). It also supports converting bin, oct, or hex values specified in SV 'b/'o/'h notation from string to integer value. The following table lists the argument parser text processing functions.

Table 4 Command Line Parse Helper Functions

Function	Description
`extract_options(ref string args[$], ref string options[string])`	Extract the dash options and plus options from the argument list and store them in an associate array.
`has_option_flag(string args[$], string flag)`	Check whether an option flag or option with no value, exists in the argument list.
`extract_keyvals(string args[$], ref string vals[string]`	Extract key-value pairs, <key>=<value>, in the argument list.
`int str_to_int(string s)`	Convert a string to integer, (supports SV bin/dec/oct/hec notation).
`qint str_to_qint(string s);`	Convert a string to a list of integer. The integer list is separated by a comma. It also supports integer range in the format of <min>..<max>.
`string get_option_string(string options[string], string key, string default_value="")`	Get the value of the specified option and return it in a string. If the option is not found, return the default value.
`int get_option_int(string options[string], string key, int default_value=0)`	Get the value of the specified option and return it in an integer. If the option is not found, return the default value.
`qint get_option_int_list(string options[string], string key, qint default_value={});`	Get the value of the specified option and return it in an integer list. If the option is not found, return the default value.

H. Tcl Script Integration

The debug prompt is good for trying out different test scenarios interactively. Loading a command files saves time on typing if the same set of debug commands are used many times. However, the debug prompt does not have any built-in programming elements - it can only execute one command after another sequentially. In other words, the debug prompt is not a Turing complete language on its own. Fortunately, we already have a fully-featured programming environment in the simulator in the form of the Tcl prompt. The uvm_debug library creates the Tcl wrapper function that calls the debug prompt, passes in and executes one debug command, and then returns to the

Tcl prompt with the return value of the debug command. The uvm_debug library together with the simulator's Tcl prompt transforms UVM into a scripting language.

Tcl prompt

```
ncsim> set read_value [debug_prompt rd_addr 0]
ncsim> echo $read_value
```

Please note that the Tcl script integration is still highly experimental. The author has only tried running the debug commands inside simple if-else branches and control loops. The Tcl prompt is essentially running everything under a single thread. It would be interesting to see how well the uvm_debug library works if there are multiple threads calling multiple debug commands simultaneously. However, Tcl was never intended to be a serious multi-thread safe programming language, so why don't we simply implement those complex scenarios in native SV code?

I. Examples Testbench

The uvm_debug library comes with an example testbench. We chose the `uvm_example/integrated/codec` example that comes with every UVM distribution to demonstrate how easy it is to set up the uvm_debug library. We selected this testbench as our example because it is simple and well understood by the verification community, yet it provides a skeleton of a full feature testbench. It has an APB register interface and a simple serial VIP sequence.

The integration is very straightforward. Other than the extra information messages added to the examples, to highlight which debug commands are currently running, it only takes three lines of code. The users can run the `demo.sh` in the uvm_debug library to see the debug prompt in action. The example also has the following simple debug command file to demonstrate the most commonly used features.

debug.cmd

```
# register access
rd_addr 0
wr_reg regmodel.IntMask 0
wr_regfld regmodel.TxStatus.TxEn 0

# stop running sequences
seqr_stop_sequences env.vip.sqr

# send a sequence item
seq_item_create vip_tr tr
seq_item_rand tr
seq_item_set_fields tr chr='hff
seqr_execute_item env.vip.sqr tr

# start a new sequence
seq_create vip_idle_esc_seq idle_seq
seq_rand idle_seq
seq_start -priority 200 idle_seq env.vip.sqr
```

III. DEBUG AND REGRESSION METHODOLOGY

Once we have established the ability to change the behavior of a compiled simulation in SV, we can adopt the proven debug methodology outlined in our previous paper [8] to speed up the debug turnaround time and minimize the regression run time. Figure 1 illustrates the three most commonly used models of the uvm_debug library.

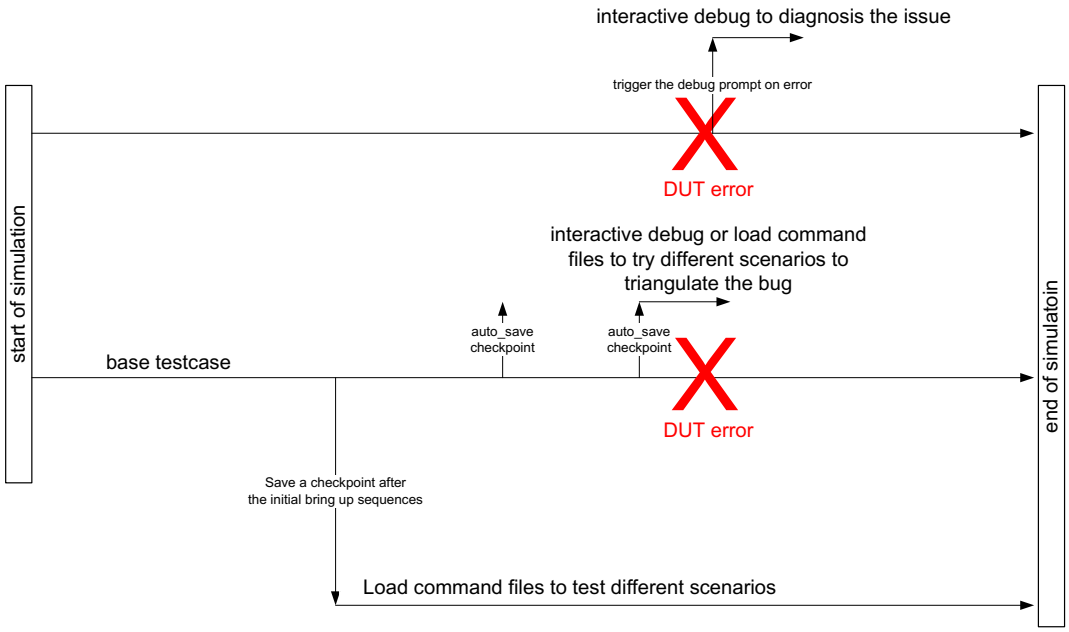

Figure 1 Interactive Debug Methodology Examples

The first use model is similar to the traditional debug flow of setting a break point in the testbench in the simulator or setting a breakpoint in the software in the post-silicon debug environment in the lab. Once an error is detected in the testbench, it will pause the simulation and trigger the debug prompt. The user can interactively read or write registers and/or inject simple stimulus to diagnosis the problem. What used to be done by manually peeking and poking RTL signals is now replaced by executing higher level debug commands, thus saving the verification engineer from the tedious work of manipulating RTL signals in Tcl.

Sometimes, using waveforms from existing simulation and merely tweaking the design after the point of failure may not provide enough information to root cause the problem. It may require the engineer to try different corner case scenarios to triangulate the bug. If the error occurs several hours into the simulation, re-running the simulation from time zero to reproduce the error could consume an engineer's entire day. A typical failure takes several iterations to isolate and correctly identify the bug, so re-running the simulation in every iteration would be unacceptable and extremely inefficient. We need to replicate the failure condition and try different stimuli in a much shorter time.

The second use model combines the uvm_debug library with the snapshot save/restore functionality of the simulator. In a simulation run, the testbench will save a checkpoint snapshot periodically, say every 30 minutes. The goal is that whenever a failure occurrs, the user can quickly restore the last saved checkpoint, knowing that it would take at most, 30 minutes to replicate the failing scenario. After rewinding the simulation to the saved checkpoint, the user can use the uvm_debug library to try various "what if" scenarios interactively by starting new sequences to modify existing stimuli. The user can also load a command file and run the simulation in batch mode, to test different scenarios in parallel instead of typing in the debug commands manually in the interactive prompt.

The third use model takes the batch mode debug application and applies it to normal regression runs. We observed that in our project, most of the testcases spent a significant amount of time configuring the device and waiting for the device to stabilize before we could start injecting interesting traffic patterns to stress different corner cases. If we have to test ten different traffic patterns, all using the identical bring up sequence, in the past we would have had to run the simulation ten times from time zero. With the help of the uvm_debug library and command files, we can save a checkpoint after the device is stabilized, then load different command files to augment the base testcase and change the traffic patterns. For example, the initialize sequence in our project consumes at least 25% of the simulation time on average, and so using the command files would speed up our regression by 25%.

V. Future Development

We implemented and tested the uvm_debug library primarily on the Cadence IES simulator. The majority of the code is simulator-neutral and is ready to deploy on other simulators, although we have never tried it. However, there is a small portion of code that relies on function calls to simulator-specific Tcl commands. Those features - such as triggering the debug prompt from the tcl prompt, a tcl procedure to fetch the return value of the debug command, saving a checkpoint snapshot of the simulation and setting the values of the UVM fields inside a sequence created by the debug prompt - do not work on other simulators in the current release. We have plans to support the big three simulators in the future. Groundwork for Mentor Modelsim integration is already in place in existing code - the SV domain can communicate with the Tcl domain in Modelsim through the SV-DPI C code. The remaining work is to replace the ncsim Tcl commands with modelsim Tcl commands. However, the author could not access the Tcl prompt of the Synopsys VCS simulator. It would be much appreciated if VCS experts can share their knowledge of Tcl integration and contribute to this project.

New developments to UVM are currently on hold due to Accelera working with IEEE to rectify the UVM 1.2 as the IEEE 1800.2 standard. When the UVM is open to accepting new development again, the author would like to contribute the uvm_debug library to Accelera and make it part of the UVM distribution. It would greatly benefit the verification community if the UVM came with a built-in interactive command line parser and a set of standard SV-Tcl interfaces. The current implementation of uvm_debug library is focused on being non-intrusive to existing testbench code, thus it limits the data structure accessible by the library only to the string named objects under the uvm_root tree hierarchy. If the uvm_debug library is adopted by Accelera as part of the UVM, it will allow a tighter integration with the UVM base classes with introspection support, and opens up many new possibilities, such as debug commands to control the run phase domains, fine grain sequence threads management, and dynamic function overloading. Ideally, the user could call any function in a uvm_component or uvm_object from the debug prompt.

VI. CONCLUSIONS

The UVM Interactive Debug Library is easy to set up. It is non-intrusive - no changed to the testbench is required other than loading the package file and calling the initialize function. With the help of the uvm_debug library, our debug turnaround time for a typical simulation failure has been reduced by 90%. The total simulation time for the regression run was reduced by 25%. Interactive debug enhances existing verification methodology, results in higher productivity of the verification engineers, lowers the cost of simulator license usage, and shortens the project schedule.

Acknowledgment

The author would like to thank his employer, Microsemi Corporation, for their support of the uvm_debug library development and the production of this paper. He would also like to thank the open-source project cluelib[9] for providing some of the text string processing code used in the library.

References

[1] Accellera Systems Initiative, “Universal Verification Methodology (UVM) 1.2 Class Reference,” 2014.
[2] IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language, IEEE Std. 1800-2012, 2013
[3] UVM Interactive Debug Library, Available: https://github.com/uvmdebug/uvm_debug
[4] IEEE Standard for the Functional Verification Language e, IEEE Std. 1647-2011, 2011.
[5] Cocotb Documentation, PotentialVentura, 2016, Available: http://cocotb.readthedocs.io
[6] M. Peryer, “Command Line Debug Using UVM Sequences”, DVCON2011
[7] J.McNeal and B. Morris, “RESSL UVM Sequences to the Mat”, SNUG2015
[8] H. Chan and B. Vandergriend, “Can You Even Debug a 200M+ Gate Design?”, DVCON2013
[9] K. Shimizu, “Sharing Generic Class Libraries in SystemVerilog Makes Coding Fun Again”, DVCON2014