RESSL UVM Sequences to the Mat

Jeff McNeal, Bryan Morris

Verilab
Jeff McNeal (Verilab US), Bryan Morris (Verilab Canada)

www.verilab.com

ABSTRACT

Read-Evaluate-Start-Sequence-Loop (RESSL -- pronounced "wrestle") is inspired by the Read-Evaluate-Print-Loop (REPL) found in Lisp and Python. The REPL in these languages encourage a rapid, iterative and interactive development process allowing the user to easily develop and test new sequences with a minimum of overhead.

In the context of ASIC verification, RESSL enables the iterative development and debug of UVM sequences. Similar to the Lisp REPL, it includes four phases:

***Read**: A simple interpreter allowing the user to input commands via STDIN.*
***Evaluate**: The evaluator takes those commands and executes them. These commands include among others, the ability to clone, alter parameters and start sequences.*
***Start-Sequence**: The system starts the sequence (and any sub-sequences) defined.*
***Loop**: Clean up and return back to the Read.*

This paper provides details on the usage model, implementation and future work planned for the RESSL.

Table of Contents

Table of Figures

Table of Tables

Introduction

Dynamic languages such as Python [1], Ruby [2] and Lisp [3] provide ways to develop programs iteratively and interactively. Using a command line interpreter, programmers can create small snippets of functions that they can immediately test for correctness. They can then iteratively build and test these snippets until they have fully defined functions. This system is referred to in Lisp and Python as the REPL (Read-Evaluate-Print-Loop). Unfortunately, providing this kind of functionality is not possible in SystemVerilog as its compile and run model is vastly different from the dynamic languages referred to above.

However, UVM can generate new objects using a factory pattern [4], and UVM sequences can nest sub-sequences within sequences -- thus enabling dynamic creation of progressively more complicated sequences. These sequences can be started dynamically on a compatibly-typed sequencer. Combining the REPL development methodology with UVM's inherent ability to create new sequences using a factory pattern and launch sequences dynamically, we created a solution we call RESSL (Read-Evaluate-Start Sequence-Loop).

RESSL adds a simple parser, interpreter and execution loop that allows the user to interactively create new sequences (from sequences that are already declared, compiled and registered with the `uvm_factory`), build new composite sequences from other sequences, and then run these sequences -- all done while the simulation is running.

In addition to providing the means to create and run sequences interactively, we made updates to many of the UVM field macros (`uvm_field_*`) to allow the user to view a sequence's variables (which have been defined using the field macro) and update its value. While this introspection/reflection capability is quite useful in this context, it is a key limitation for those groups who are not able to patch their UVM library. However, it is possible to add this capability without modifying UVM -- it simply requires more coding than the transparent 'under-the-hood' solution of modifying the UVM field macros.

Sequences are great, but…

UVM Sequences are a powerful feature of UVM. Using the UVM sequence classes and associated macros, coupled with SystemVerilog random constraints, sequences of interesting scenarios can be created quickly and easily. More importantly, complicated sequences can easily and iteratively be created based on simpler sequences that already exist.

One limitation with this capability is that new sequences must be coded and then compiled before they can be used. Since they are compiled source code, they cannot be modified or expanded while a simulation is running.

There are two principal users of sequences: RTL design engineers and Verification engineers and they each have different use cases. RTL designers want sequences that are easy to create and modify while testing their code. They generally don't want to mess with the testbench or learn UVM in order to do simple testing. Ideally they would like to be able to quickly build and tweak stimulus, in this case a sequence, on their own without waiting for the verification engineer to modify the sequence source code or create a multitude of command line arguments to set different attributes of the sequence. On the other hand, verification engineers want it to be convenient to be able to change the attributes of the sequence under development without having to recompile the code and then launch the sequence to verify its behaviour. For both use cases, it would be ideal that once you have created an interesting sequence you should be able to save and replay it for subsequent simulations or regressions.

The system we've created, RESSL, provides the ability to dynamically create sequences from existing sequences, tweak any sequence's parameters, and then start the interactively created sequences, all while the simulation is running from a command line prompt using a simple API.

The Lisp Inspiration

One of the original languages to provide dynamic, interactive development capability was Lisp. It was one of the first languages to introduce a REPL [8]. The four phases of a Lisp REPL are:

- **Read**: accept input from the user (typically Lisp code.)
- **Evaluate**: process the input from the user, ensure it is syntactically correct and meaningful in the current context and then allow it to be executed within the Lisp system.
- **Print**: the results from the Evaluate phase are then presented, allowing the developer to immediately and interactively build their program.
- **Loop**: returns back to the Read phase enabling the iterative process.

This type of development model would be useful when writing or using a SystemVerilog testbench. However, since SystemVerilog is a compiled, static language, we cannot interactively execute new code during run time. RESSL takes its inspiration from interactive languages and goes as far as it can to provide a way to interactively create, alter, inspect, and start UVM sequences. This provides much of the benefit of a full REPL in an interactive language in SystemVerilog. As the name implies, and similar to above, it comprises four phases as shown in Figure 1 - RESSL Processing on page 6:

- **Read**: a simple interpreter that reads the command line input, tokenizes it and passes it to the Evaluate phase
- **Evaluate**: interprets the command and executes.
  This evaluation may include:
  - adding, changing the sequence registry[1] where all sequences are stored.
  - displaying, modifying or randomizing sequence variable values via introspection [5][6]
  - creating new sequences.
  - storing and loading sequences to and from files.
- **Start-Sequence**: starts the sequence specified from the sequence registry. Note that sequences can contain sub-sequences.
- **Loop**: executes any clean-up and loops back to the Read phase.

[1] The sequence registry is a structure to hold the sequences known to RESSL. More details are provided in the RESSL design on page 12.

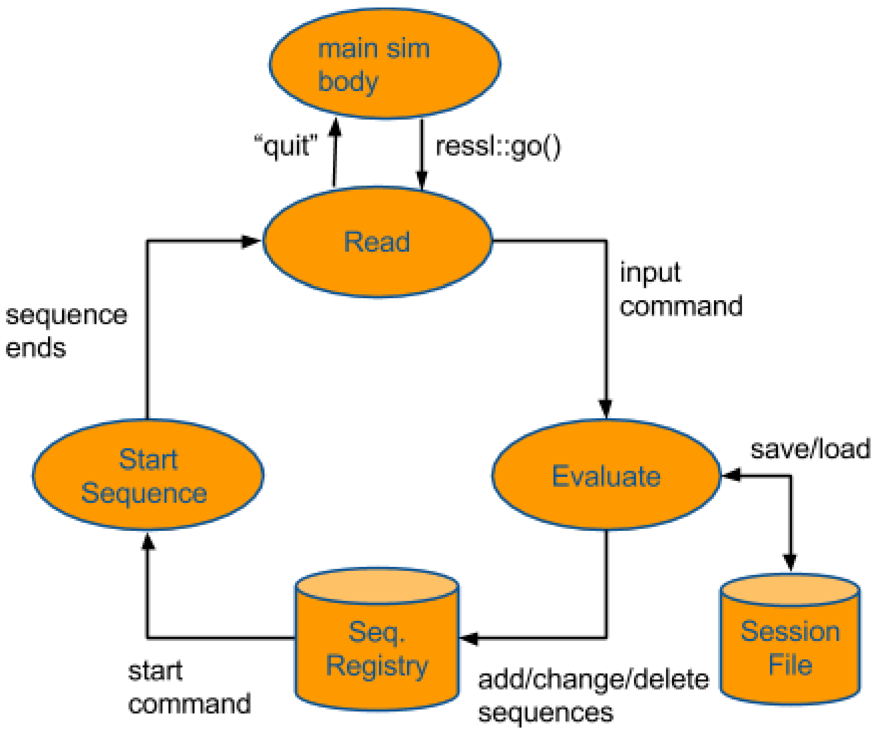


Figure 1 - RESSL Processing

RESSL Usage

General UVM guidelines recommend that sequences be divided into several levels. At the bottom level are atomic or protocol specific sequences. These sequences generally perform a single, well bounded task, such as transferring a word of data, or setting control signals. The next level up are sequences which perform a task using the lower layer sequences. These may be register writes and reads, or packet transfers. The third level of sequences use the second (and first) levels to accomplish tasks, such as transfer a block of memory, or configure the DUT.

RESSL requires that at least the lowest level of sequences is already written and working in the verification environment. At that point the user can start up RESSL and interactively build mid-level sequences out of the low level sub-sequences in an interactive manner. RESSL allows the user to string together several sub-sequences in a sequence and execute the sequence, then observe the result.

Then changes can be made to the sequence and it can be run again, all within the same simulation execution. The user can change the values of fields in the sequence to whatever is desired, allowing users to quickly try out different combinations of values without editing the source or re-compiling.

Once the user is happy with the sequences that have been built, they can be saved to a file for later use, or to be coded into the verification environment as tests. In a future simulation the user can load the sequences and use them again.

Example Usage Scenario

This scenario assumes that the user has already developed an environment with at least one UVC and several low level sequences. The user has also integrated RESSL into their environment as described in the section entitled "Interesting… but how hard is it to add to my existing environment?" on page 15. The environment includes three sequences `read_only`, `write_only`, and `deallocate_only`. Once the user's simulation encounters the `ressl::go()` task it transfers control to the RESSL command line prompt which displays to the Unix stdout:

```
SV-RESSL: Pinning sequences to the Mat since 2014....
Version 0.1
[*] >>> _
```

Let's list the known sequences defined in the registry, so we see what we have to work with.

```
[*] >>> list
read_only [1]
write_only [1]
deallocate_only [1]
```

In this example, we loaded three sequences named `read_only`, `write_only`, and `deallocate_only` in SystemVerilog as part of starting the RESSL. These sequences are now part of the sequence registry that holds all the sequences known to RESSL.

Load more sequences from an external file.

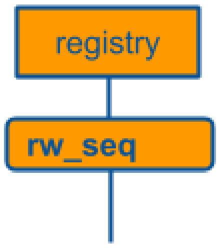
```
[*] >>> load common_sequences.ressl
loading registry from common_sequences.ressl
...read_write

[*] >>> list
read_only [1]
write_only [1]
deallocate_only [1]
read_write [1]
```

The `common_sequences.ressl` file contained a sequence named `read_write` (which is a sequence containing the `read_only` sequence followed by the `write_only` sequence). The sequence registry now has four slots, with each slot holding a handle to a sequence.

When we want to add a new sequence, or focus on an existing sequence, we select that sequence. If the name does not exist in the sequence registry a new empty slot is added that will eventually hold a new sequence.

```
[*] >>> select rw_seq
selecting sequence rw_seq (new)
[rw_seq] >>>
```

In this case, the `rw_seq` name does not refer to any other sequence in the registry so a slot is added into sequence registry. Note also that the prompt changes from `[*]` to `[rw_seq]` to reflect the currently selected sequence is now the new `rw_seq` sequence. The diagram below illustrates the new sequence in the sequence registry.



We want to add a single `read_only` sequence as a sub-sequence of `rw_seq`.

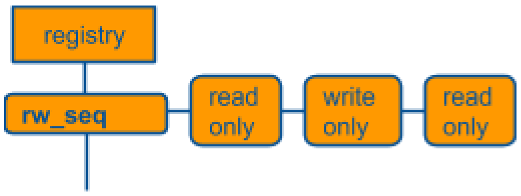
```
  [rw_seq] >>> add read_only
  rw_seq:
      [0] read_only
```

Here the `[0] read_only` indicates that the `read_only` sequence is the first sub-sequence to be run in the `rw_seq` sequence.

Now add a write sequence, and then another read sequence

```
  [rw_seq] >>> add write_only
  rw_seq:
      [0] read_only
      [1] write_only
  [rw_seq] >>> add read_only
  rw_seq:
      [0] read_only
      [1] write_only
      [2] read_only
```

At this point we have a new sequence (`rw_seq`) that has three sub-sequences in it, `read_only`,



`write_only`, and `read_only`.

Get some info about sub-sequence index 0 (the first `read_only` sequence) of this seq

```
  [rw_seq] >>> describe 0
  [0] Sequence: read_only (type:read_only_seq)
      Fields:
        field: number_of_commands = 1
        field: pre_delay = 100
```

The `read_only` sequence has two fields: `number of commands` and `pre_delay`. RESSL will print any fields defined in the sequence.

Or the fields for all the sub-sequences:

```
  [rw_seq] >>> describe
  [0] Sequence: read_only (type:read_only_seq)
      Fields:
        field: number_of_commands = 1
        field: pre_delay = 100
  [1] Sequence: write_only (type:write_only_seq)
      Fields:
        field: number_of_commands = 1
        field: pre_delay = 100
  [2] Sequence: read_only (type:read_only_seq)
      Fields:
        field: number_of_commands = 1
        field: pre_delay = 100
```

Change the number of commands for the 0th index sequence i.e., the first `read_only` sequence:

```
  [rw_seq] >>> set 0 number_of_commands 5
  [0] Sequence: read_only (type:read_only_seq)
      Fields:
        field: number_of_mcs_commands = 5
        field: pre_delay = 100
```

Now list the known sequences in the registry

```
[rw_seq] >>> list
  read_only [1]
  write_only [1]
  deallocate_only [1]
  read_write [1]
  rw_seq [3]
```

Start the sequence of sequences. We can either run it once (default) or specify a number of times to repeat. In this case, we run it once.

```
  [rw_seq] >>> start
  --- starting read_only sequence
  Fields:
     field: number_of_mcs_commands = 5
     field: pre_delay = 100
  ... normal log for { read x 5 } sequence ...
  --- end of read_only sequence
  --- start of write_only sequence
  Fields:
      ## whatever the attributes are
  ...normal log for write seq...
  --- end of write_only sequence
  --- starting read_only sequence
  Fields:
     field: number_of_mcs_commands = 1
     field: pre_delay = 100
  ... normal log for read sequence ...
```

```
--- end of read_only sequence

[rw_seq] >>> start 3
--- starting read_only sequence
... a bunch more stuff here ...
--- end of read_only sequence
```

We changed our mind, we don't need that second `read_sequence`, let's discard it:

```
[rw_seq] >>> describe
[0] Sequence: read_only (type:read_only_seq)
    Fields:
      field: number_of_commands = 1
      field: pre_delay = 100
[1] Sequence: write_only (type:write_only_seq)
    Fields:
      field: number_of_commands = 1
      field: pre_delay = 100
[2] Sequence: read_only (type:read_only_seq)
    Fields:
      field: number_of_commands = 1
      field: pre_delay = 100

[rw_seq] >>> delete 2
Chucking [2] read_only from rw_seq. Ya sure (y/n)? y
[rw_seq] rw_seq:
   [0] read_only
   [1] write_only
```

Now, save the entire sequence registry to a file.

```
[rw_seq] >>> store common_sequence.ressl
...read_only
...write_only
...deallocate_only
...read_write [1]
...rw_seq
saved registry to common_sequence.ressl
```

If all else fails: there's always a bit of...

```
[rw_seq] >>> help
--- RESSL Help ---
     add <sequence name>
     create <seq_type> <seq_name>
     describe <sequence name>
     help [verb]
     list
     randomize <seq_index>
     select <seq_path>
     set <seq_index> <field> <value>
```

RESSL API

The current set of commands available in RESSL are [2]:

Table 1 - Current RESSL Commands

Verb	Description
`load <registry_file>`	Load sequence registry from file.
`store <registry_file>`	Store sequence registry to a file.
`select <seq_name>`	Start recording a new sequence (or open an existing sequence); which is added to the sequence registry
`List`	List all the sequence names in the sequence registry
`create <seq_type> <seq_name>`	Using the uvm_factory to create a uvm_sequence_base of type <seq_type> and assigning it the name <seq_name>
`add <seq_name> <repeat>`	Adds the sub-sequence to the current sequence.
`copy <seq_name> <copy_name>`	Copies the sequence to a new name: any sub-sequences are also copied.
`delete [<seq_name>] [<seq_index>]`	Delete the sub-sequence indexed by <seq_index> from the sequence named <seq_name>. If no <seq_index> is supplied, it deletes the entire <seq_name> from the sequence registry.
`move <src_index> <dst_index>`	Allows you to move a sub-sequence up or down to a new index position in the sequence.
`describe <seq_name> [<index>]`	Displays the attributes for named sequence's sub-sequence at the index specified. The attributes for all sub-sequences will be provided if the <index> is not supplied.
`set <index> "attribute" "value"`	Sets attributes of one of the sub-sequences in the sequence.
`start [seq_name] [index]`	Starts the sequence. With no arguments it starts the currently selected sequence; Supplying only the <seq_name> starts that sequence. Supplying both arguments starts the sub-sequence indicated by <index> of the <seq_name> sequence.
`shuffle <seq_name> [<shuffled_seq_name>]`	In addition to starting the named sequence, it first randomizes the order of the sub-sequences defined to create a different sequence. By default, this is a transient shuffle where the original order defined remains--unless the user supplies the optional <shuffled_seq_name> in which case the shuffled sequence is copied to the new name.
`help [cmd]+`	Displays help message. With no arguments it displays all the known commands; otherwise it displays the help message for the <cmd> specified.
`randomize <seq_index>`	Randomize the sub-sequence at <seq_index>
`attach <seqr> <seq>`	FUTURE: Attach the specified sequencer <seqr> to the <seq>. This enables RESSL to access multiple sequences

[2] The command set and underlying design are in a state of flux as we continue to develop RESSL. Any commands listed in RED are for future consideration and possible implementation.

	and sequencers within the same simulation.

Typical Usage Scenarios

We envision RESSL being useful to a variety of people in a variety of ways.

- Simple sequences for RTL debug
- Debugging sequences, drivers, monitors.
- Developing and identifying interesting set of sequences
- Quick way to develop a sequence library.

RESSL Design

The design is conceptually split into four parts:

- *Interpreter*: The 'R' and 'E' of **RE**SSL. A simple command execution loop that accepts input from the command line, tokenizes the input, and then creates and executes the appropriate command.
- *Sequence Loop*: The "SSL" of RE**SSL**. Starts the sequences identified and loops back to the interpreter.
- *Introspection/Reflection*: The silent "I" in RESSL. A necessary part of the design is a crude introspection capability to enable dynamic setting of a sequence's attributes via a command line.
- *Storage*: The ability to record, save and re-load libraries of previously developed sequences.

From the UML class diagram below there are currently two main classes that implement the RESSL functionality. The `ressl` class derives from `uvm_object` and provides the following functionality:

- **a sequence registry**: a simple database of all the defined `uvm_sequence_base` objects. Sequences are added into the registry using the `add_to_sequence_registry()` function or using the `add` command. The current implementation uses a simple associative array of handles to `uvm_sequence` objects keyed by the sequence name.
- **an interpreter** that accepts input from a simple command line display, tokenizes the input and then passes processing onto the associated `ressl_cmd` that implements the requested command.

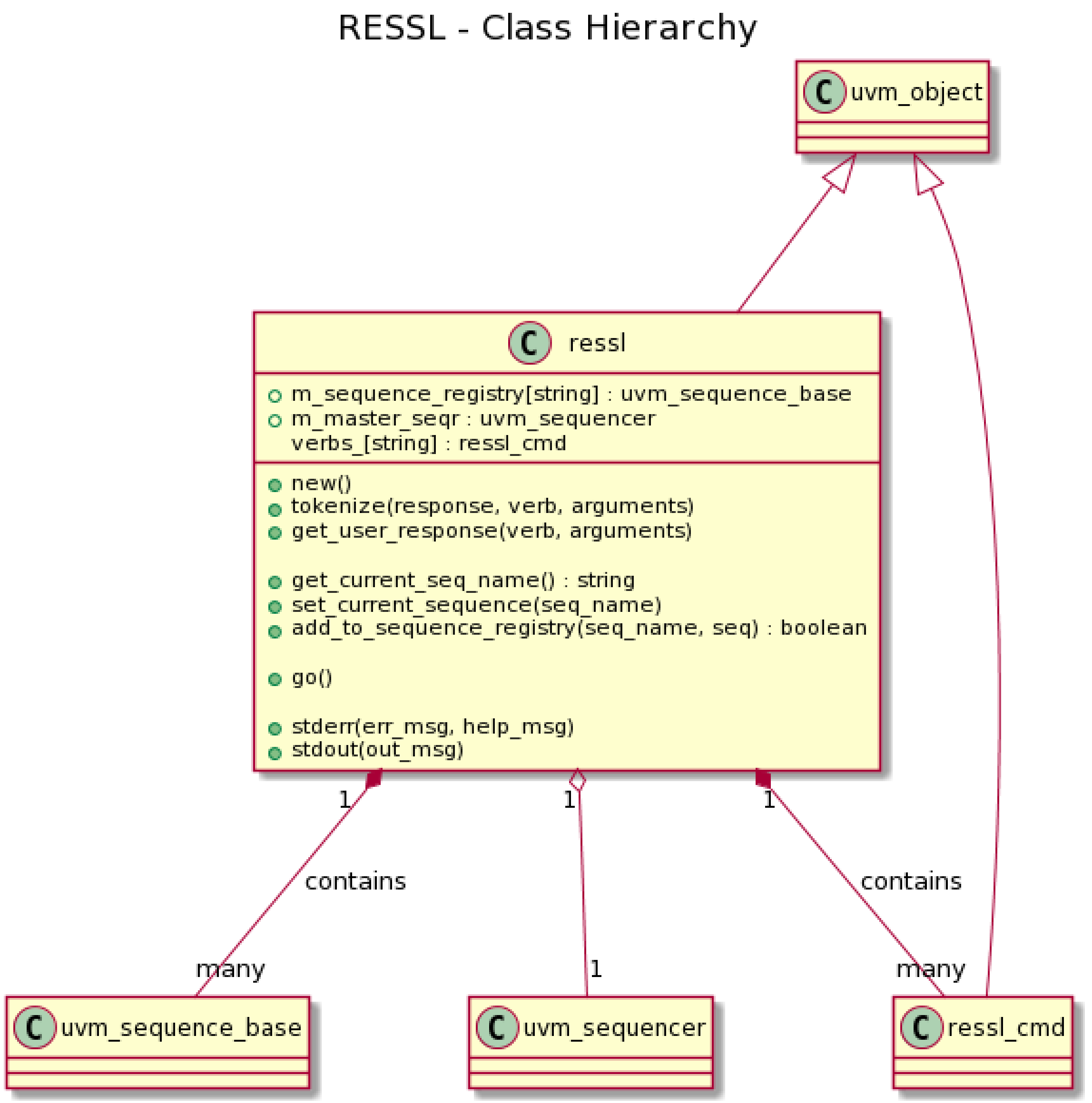

RESSL - Class Hierarchy
uvm_object
ressl
m_sequence_registry[string] : uvm_sequence_base
m_master_seqr : uvm_sequencer
verbs_[string] : ressl_cmd
new()
tokenize(response, verb, arguments)
get_user_response(verb, arguments)
get_current_seq_name() : string
set_current_sequence(seq_name)
add_to_sequence_registry(seq_name, seq) : boolean
go()
stderr(err_msg, help_msg)
stdout(out_msg)
1
contains
many
1
1
1
contains
many
uvm_sequence_base
uvm_sequencer
ressl_cmd

Figure 2 - RESSL Class (UML Class Diagram)

Interpreter

The interpreter component is split into three parts:

Get User Input

The `ressl::go()` function implements a loop that accepts any input (one per input line) until the `quit` command is received. Each line is passed to the tokenizer for processing.

Tokenizer

implemented in the `ressl::tokenize()` function, it uses a simple command line parsing that splits the input into a “verb” followed by a set of “nouns”. Where the verb is assumed to be first word in the command, and the nouns is every word that remains on the line. The `tokenize()` function accepts the user’s response and outputs the verb as a string, and the nouns as a queue of strings (can be zero). Each valid verb has an entry in the `ressl::verbs_` associative array of handles to `ressl_cmd` objects. The queue of strings (nouns) is passed to the `ressl_cmd::execute()` command for processing. For example, the `help` command has an expected syntax of `help <cmd>`, so if the user enters `help add`, `ressl` passes the `add` string in a queue to the `help_cmd` object (derived from `ressl_cmd`) for processing.

Verbs

Each verb in the RESSL system corresponds to a class derived from `ressl_cmd`. `ressl_cmd` implements the “strategy”[7] or policy pattern. The `ressl_cmd` is an abstract class that has two `pure virtual` functions:

- `execute`: performs the main processing for the command
- `help`: displays help information specific to the command.

For example, the `add` command’s `execute()` function accepts a single argument `seq_name` which indicates which sequence you want to add as a sub-sequence to the currently selected sequence. The `add` command does the the following processing:

- searches the sequence registry for the `seq_name` provided and returns the handle to the associated `uvm_sequence_base`.
- using the `uvm_object::clone()` function, creates a new copy of the `uvm_sequence_base`.
- adds this cloned `uvm_sequence_base` object to the sequence registry as a sub-sequence of the currently selected sequence.

As can be seen from the UML class diagram below, there is a one-to-one correspondence between a verb in the RESSL API and a class that derives from `ressl_cmd`.

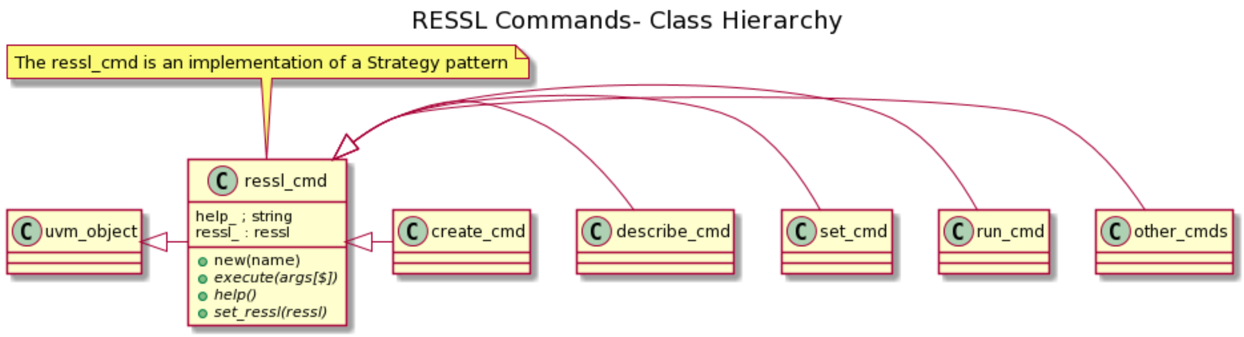


Figure 3 - RESSL Command (UML Classs Diagram)

Start Sequence

Once a new sequence has been created and added to the sequence registry it can be started using the normal `uvm_sequence::start()` function. The `ressl` object simply uses the currently selected sequence, finds it in its sequence registry and then executes a SystemVerilog `foreach` loop on each sub-sequence defined for the sequence. The `uvm_sequencer` associated with any sequence defined in the `ressl` sequence registry must have been previously set. In the body of the `foreach` loop, this `uvm_sequencer` is passed to the `uvm_sequence::start()` task.

Introspection/Reflection

An important functionality for RESSL is the ability to display and set field values for any sequence in the sequence registry. RESSL implements this functionality by extending the `uvm_field` macros to add a "set" and "get" capability. New functionality created in the `` `uvm_field_* `` macros automagically adds the ability to view and change any field defined with these macros from within RESSL.

Storage And Retrieval

RESSL can record the sequences and sub-sequences being developed, which can then be saved to and reloaded from an external file. This enables creating and defining the sequence; provides a "replay" functionality to restore the sequence registry to the same state when the registry was "stored" to an external file. The storage file could be shared with other users, to facilitate development or to aid in debugging.

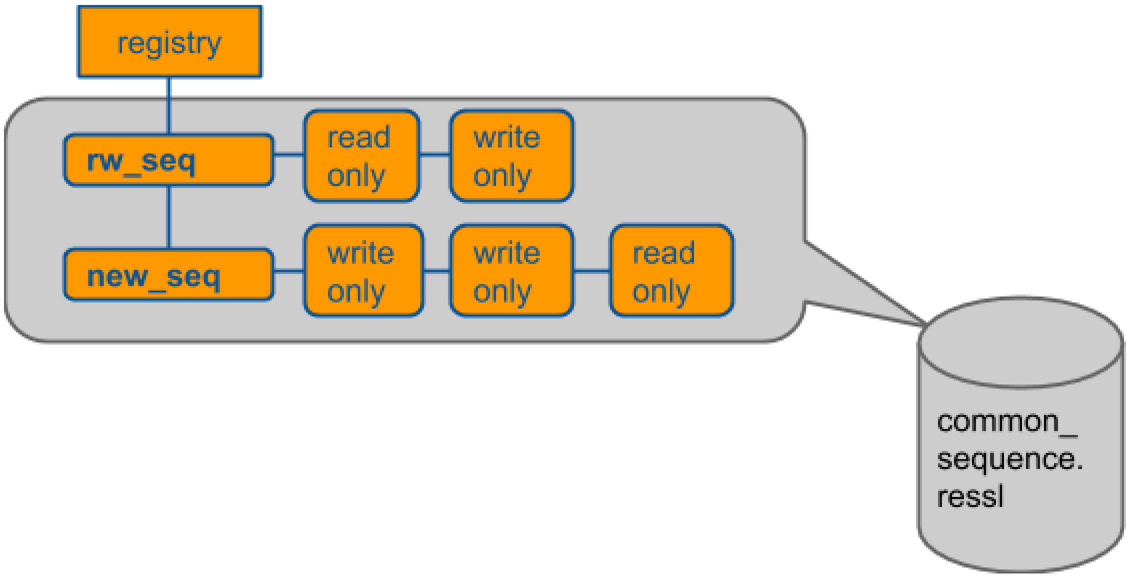


Figure 4 - RESSL External File

Once useful sequences are identified using RESSL, it is best to then code a `uvm_sequence` and add it into your verification environment (and to the instance of the RESSL). To be clear, RESSL helps identify useful scenarios by enabling a rapid development cycle that encourages exploration. At some point, the good sequences should be coded as true UVM sequences and selected during regressions.

Interesting… but how hard is it to add to my existing environment?

To "RESSL-ize" your environment there are three key requirements, but the first two are already part of the normal development process:

- Create a set of UVM sequences that you can optionally seed the sequence registry using the `ressl::add_to_sequence_registry()` function. Generally, these are sequences of 'atomic' traffic i.e., sends a stream of transactions of a certain type. These are generally going to be the sequences that are developed as a part of normal verification, so do not represent extra work.

- Create a UVM sequencer that can be used to send these sequences. You can optionally attach these sequencers to the sequence using the uvm_sequence_base. This sequencer will also usually be developed as part of normal verification work.
- Add the call to the `ressl::go()` task at the appropriate time in the `run_phase` of the simulation.

To demonstrate how straightforward it is to add RESSL to your environment, we'll list the steps we did to add RESL to the `ubus` example found in the UVM 1.2 library. Adding RESSL required changes to the base `ubus_example_base_test` and the `derived test_2m_4s` (two masters, two slaves) class.

Step 1. Add the svlib

RESSL has been incorporated into svlib [9]. svlib is a open-source utility library for SystemVerilog that provides many useful functions including file and string manipulation, regular expression matching, etc. svlib uses the same open-source license as UVM and so RESSL explicitly uses the same licensing and will be freely available.

In turn, RESSL uses svlib's string manipulation capabilities in the current implementation of its command tokenizer.

In order to access RESSL, we include the `svlib_macros.svh` (macro definitions used in svlib), and `import svlib_pkg::*`.

Step 2. Declare and instantiate the RESSL object

Add the declaration for the `ressl` instance as part of the `ubus_example_base_test`:

```
ressl ubus_ressl;
```

Create the `ressl` instance using the UVM factory:

```
ubus_ressl = ressl::type_id::create("ubus_ressl");
```

Step 3. Disable default sequence

The `ubus` example creates a default sequence for the masters called `loop_read_modify_write_seq` and assigns it as a default sequence for the master's uvm_sequencer. We need to "turn off" the default sequence because it interferes with the `ressl` instance's use of the master sequencer. The disabling of this default sequence is by NOT allowing the default sequence to be defined in the `uvm_config_db`. In this case, we simply comment this out, because at some point we'll likely stop using RESSL and we'd like to add this default back in (yes, there are better ways of doing this):

```
lrmw_seq = loop_read_modify_write_seq::type_id::create();
//uvm_config_db#(uvm_sequence_base)::set(this,
//
"ubus_example_tb0.ubus0.masters[1].sequencer.main_phase",
//                "default_sequence",
    //              lrmw_seq);
```

Step 4. Seed RESSL's sequence registry with the lrmw_seq

This is an optional step as this sequence can be created using RESSL's `create` command. Nevertheless, this step populates the RESSL sequence registry with a useful sequence, and gives the user something to see when doing a `list` command at startup.

```
    ubus_ressl.add_to_sequence_registry("lrmw_seq", lrmw_seq);
```

Step 5. Start RESSL

In the run phase of the test, we start the RESSL interpreter and pass control to it using the `ressl::go()` task:

```
    ubus_ressl.go();
```

Now we're ready to start the simulation. The `go` task will block, jump out to the RESSL command line, and the user can begin working interactively as illustrated previously.

UVM Library Modifications

In order to give RESSL users the ability to inspect and modify sequence field values, we have modified the standard UVM library. This section outlines what was added to the library, and which classes or macros are affected by these changes.

Modifying uvm_object Class

Specifically, `uvm_field_{int,string,enum}` macros, and the `uvm_object` class have been altered from the UVM 1.2 library (added `get_field_members` and `set_field_value` functions)

Modifying UVM Field Macros

The `uvm_field_*` macros already enable the auto-magic creation of many useful functions for each field defined including copy, compare, pack, unpack, print, and record. As discussed, in order to tweak individual elements of a `uvm_sequence` we needed to add the concept of introspection and reflection to RESSL. Briefly, introspection "provides the ability for a program to examine the type or properties of an object at runtime" [5]; while reflection is "the ability of a computer program to examine (see type introspection) and modify the structure and behavior of the program at runtime" [6].

For the purposes of RESSL, we need to be able to retrieve the value of any field in a sequence and display it as a string and vice-versa, accept a string and convert it to the correct type to set the corresponding field value in the sequence. For this we add conversion from and to a string as part of the UVM field macros.

Since the UVM library does not provide this capability natively, we modified the `uvm_field_*` macros. This section provides an overview of the changes to the UVM library required to add this capability.

Two new constants are added into the `uvm_object_globals` to handle the new conversion for each field:

```
parameter UVM_SETFROMSTR    = UVM_START_FUNCS+4;
parameter UVM_GETSTR        = UVM_START_FUNCS+5;
```

The `UVM_SETFROMSTR` enables the creation of method to convert *from* a string into the appropriate type defined by the field macro e.g., `int`, `string`, `real`, etc. For `int` variables, it takes into account if a *radix* has been supplied for the field and assumes the incoming string is in the same radix e.g., if the user sees the variable is displayed as `UVM_HEX`, the RESSL `set` command (to set the field value) is assumed to be supplied with a hex string representation e.g., `set x 1CAFE`[3].

Similarly, `UVM_GETSTR` converts the current value of the data member into a string representation. For int variables, it takes into account if a *radix* has been supplied and provides the correct conversion.

At this time, the introspection/reflection for RESSL are limited to `int`, `string`, `enum`, and `real` values. We did not add this capability to the `uvm_field_object`, and any of the array field macros.

As each field is created via the macros, it creates a new *scope* for that variable. This is contained in a composite object called `__m_uvm_status_container`. This scoping provides the ability to query which fields have been defined for the class i.e., which scopes have been defined.
So the new `UVM_SETFROMSTR` functionalty searches the `__m_uvm_status_container` for the field being queried, and if found converts the string to the appropriate type value e.g.,for `int` fields:

```
field = string.atobin() for UVM_BIN
field = string.atoi()   for UVM_DEC
field = string.atohex() for UVM_HEX
```

This is done directly in the `uvm_field_int` macro found in `uvm_object_defines.svh` (macro’s source).

The astute reader (which included our Verilab colleague Jonathan Bromley) will notice that the use of the `atoi<int>` functions in the conversion from `string` to `integer` has the limitation that these functions return a 32-bit `integer` value. This means that the setting of any field will be limited to 32-bit values. This limitation may be removed by the time you read this paper.

Similarly, the `UVM_GETSTR` returns the string representation e.g., for `int` fields:

```
string = $sformatf("'b%0b",field)     for UVM_BIN
string = $sformatf("'d%0d",field)     for UVM_DEC
```

[3] Future update of RESSL will include the use of string radix “\x###” (hex), “\d##” (dec), etc. e.g., `set x \x1CAFE` to specify the radix of the new values.

```
    string = $sformatf("'h%0x",field)    for UVM_HEX
```

Similar macro additions are then made for `uvm_field_string`, `uvm_field_enum` and `uvm_field_real`.

As discussed, the array based field macros and object macros do not provide this introspection/reflection capability.

Limitations and Future Work

This section outlines both the current limitations of RESSL, and our plans going forward to mitigate these limitations and add some new functionality.

The key limitation of our RESSL system -- that can potentially prevent some teams from using RESSL to its fullest -- is that in order to get the introspection/reflection capabilities we modified the UVM 1.2 library source code (as discussed above, the `uvm_object` and the `uvm_field_*` macros were altered). The introspection/reflection capabilities are optional, as they are not required to create and execute sequences. However, this ability to see the current values for fields in sequences and be able to update them on the fly is a key feature of the system.

Otherwise, we provide a patch for the two files that are changed to add introspection/reflection. The intent is to donate this into the UVM library for subsequent releases.

We do not have a way to add new constraints to a sequence. Any existing constraints will be respected when the sequences are randomized, but we do not support a method similar to the `` `do_with() `` macro. We have some thoughts on how we can provide a crude constraint mechanism, but that is not available at this time. This limitation is mitigated by the fact that the user can inspect a sequence after it has been randomized and then change values for random variables using the `set` command. Of course this means that constraints involving variables changed by the `set` command may not be satisfied.

In a practical sense, this means that the user can either set non-random variables, then randomize the sequence and then start it, or randomize the sequence, then modify variables, then start it, but not both. We do not have hooks into `pre_` or `post_randomize()` functions so any `set` commands will take place after randomization.

We do not provide a way to add new fields to sequences.

A *current* limitation is that each RESSL instance has access to exactly one `uvm_sequencer`. All sequences in the sequence registry must be compatible with this sequencer. We are currently testing an update to allow RESSL to hold a sequencer registry, and then the user can attach any sequence in the sequence registry to any compatible sequencer in the sequencer registry. This capability might be present in RESSL by the time we present.

Future features/improvements includes the following:

- redesigning the sequence registry from an associative array to a “tree” structure to allow the creation of a hierarchy of sequences. While the current system does allow nested sequences, the underlying structure to hold the sequences is an associative array -- providing only one level of hierarchy. Converting to a tree structure is both a cleaner architecture that matches the abstraction of nested sequences better, and provides opportunities for more interesting commands e.g., copying an entire branch. This will requiresome additional commands or changes to existing commands to select, traverse and modify the hierarchy of sequences.

- adding some mechnanisms to allow constraints (even crude constraints) to be applied during randomization (a topic for a future paper, I'm sure ;-). This could include a new verb e.g., "validate" that validates that the randomization succeeded.
- creating a "recipe" to add introspection to the sequences without altering the UVM code. This alternative requires a minimal amount of extra work while creating the sequences to 'publicize' the tweakable attributes.
- replacing the existing crude user command line with the VCS (Tcl) comand line: by either extending the existing VCS command line with the RESSL functionality, or providing a conduit *from* RESSL to the command line. This couples the existing power of the VCS command line with the powerful capabilities of RESSL. For example, from the command line the user can force Verilog nodes during a simulation, run a sequence, evaluate the results and could interleave these activities during debug.

Conclusions

In this paper we describe a new technique inspired by dynamic languages such as Lisp and Python to create an iterative, interactive development environment to create and start `uvm_sequences` on the fly called RESSL (Read-Evaluate-StartSequence-Loop). We describe the underlying design of RESSL and demonstrate that adding this environment requires little additional coding. One of RESSL's key assets is that we have added an introspection/reflection layer to many of the UVM field macros. This allows the user to dynamically inspect and change a variable in the sequence that has been defined by a `uvm_field_*` macro. However, this introspection/reflection capability is also a key limitation for many users since it requires you to use a modified UVM library.

We also describe some typical user scenarios, existing limitations of the system, and identify some potential future work.

Using the RESSL system enables an interactive development environment that encourages the easy and iterative development of a sequence library without having to recompile the code. Using RESSL provides a platform where RTL designers are given a verification environment where they can drive simple sequences, easily change sequence variables, and create more complex nested sequences to test their design. For verification engineers it allows them to interactively explore and define interesting sequences in an iterative way, and create libraries of interesting scenarios.

RESSL will continue to evolve and develop as we learn more about using such an interactive, dynamic way to develop sequences. We've identified some future improvements such as allowing sequencers to be created and attached to sequences, permitting constraints to be applied, etc. RESSL is freely available with the same open-source licensing as UVM and we'd welcome comments, ideas, and patches for inclusion into future revisions.

All in all, using a system like RESSL gives you the ability to tag-team in a cage-match, launch UVM sequences off the turnbuckle and wrestle the DUT into a submission hold with no hope for re-match to help you become known as "The Bug *Crusher*". Stay tuned for more "Smack-Down" action as we continue developing RESSL.

Acknowledgements

We would like to thank our colleagues at Verilab for their careful and positive review comments. With special thanks to Jonathan Bromley for his encouragement of this idea, and the generous gift of his time for the questions that popped up while writing this paper.

References

[1] Python Language Website: https://www.python.org/

[2] Ruby Language Website: https://www.ruby-lang.org/en/

[3] Wikipedia entry for Lisp: http://en.wikipedia.org/wiki/Lisp_%28programming_language%29

[4] Wikipedia entry describing the software Factory Pattern: http://en.wikipedia.org/wiki/Factory_method_pattern

[5] Wikipedia entry describing type introspection: http://en.wikipedia.org/wiki/Type_introspection

[6] Wikipedia entry describing Reflection: http://en.wikipedia.org/wiki/Reflection_(computer_programming)

[7] Wikipedia entry described Strategy software pattern: http://en.wikipedia.org/wiki/Strategy_pattern

[8] Lisp I Programmer's Manual (1960), References to REPL on page 2, http://history.siam.org/sup/Fox_1960_LISP.pdf

[9] svlib - a programmer's utility library for SystemVerilog written by Jonathan Bromley, http://www.verilab.com/resources/svlib/