Applications of Custom UVM Report Servers

Gordon M^cGregor

Verilab, Inc.
Austin, Texas

www.verilab.com

ABSTRACT

The Universal Verification Methodology (UVM) provides a feature-rich set of reporting and message handling objects. These message handlers are being improved in UVM version 1.2 to provide an OOP-style messaging infrastructure. This paper discusses the existing handlers in UVM 1.1 and previews the implementation coming in UVM 1.2.

A common task in many verification environments is customizing the log format and this paper will show how to do this for the UVM 1.1 and UVM 1.2 base class libraries. Methods to extend both implementations to produce log files using markup text formats will be discussed. Several use cases for these structured text files are also considered, with examples using an XML based approach that can be extended to other markup formats or SQL database storage, as needed. An interactive GUI, built to manipulate this structured log format, is demonstrated that enables complex filtering and dynamic message reconstruction as well as command line filtering approaches to produce custom reports.

Table of Contents

Table of Figures

1. Introduction

The Universal Verification Methodology (UVM) [1] is the most commonly used verification methodology in the industry. When using the UVM, much of a verification engineer's time is spent running and debugging testcases, both while developing a testbench and also when running regressions and driving towards coverage closure. The UVM provides a rich set of logging objects that are used to produce text log files, with fine-grained control over the levels of verbosity and message filtering. These logs, along with waveforms, are typically the main source of information used when debugging a failing test. Powerful IDEs and GUI interactive debugging tools are available to debug SystemVerilog, but the length of simulation time often makes their use less practical than using the text logs and waveforms to find the cause of a failure. Simulations produce copious log messages, from many sources within the testbench, tracking the operation of the various subsystems. It can be difficult to quickly get a sense of the flow of the test, while still recording sufficient detail to find the root cause of an issue.

Tools are available [4, 5] that post process log files and attempt to extract additional meaning or structure from a UVM log. These tools are entirely dependent on the specific format string used within the UVM report handler to construct messages. Any change made to this format typically will break tools that parse UVM logs. These tools use regular expressions to extract structure from the messages so require knowledge about the structure of each message and expect fields to be in a known order and relationship. This tight coupling between log parsers and the message generator is not ideal and makes the current solutions less flexible than they could be.

While these post-processing tools are tied to the message format within the library, it is a frequent request within verification teams to be able to change that message format. There are several motivations for this. Often the main string of the message is obfuscated by the surrounding meta-data, describing the originating object, originating file, message severity, plus several other pieces of ancillary data about any given message. While this meta-data can be very useful in understanding the context of a given entry, it also tends to distract the eye when trying to understand the flow of a test. This is particularly true of the __FILE__ and __LINE__ macros that provide detailed information of the originating source file and line number for a given message. This can be vital, in certain circumstances, to find where a message is coming from, but the rest of the time it is almost completely redundant from the perspective of trying to understand a test. As a result, most users run their simulations with this information disabled, only enabling it when truly necessary. This requires a repeat simulation run to capture this additional data, which wastes time and compute cycles. A more useful situation would be to be able to selectively view this sort of meta-data, on demand, without having it visible the majority of the time.

2. The Problem With the Standard Message Format

As an example of the problem, Figure 1 shows a message with all the available meta-data. The actual message gets pushed far to the right of the screen, making it difficult to even view without line wrapping. In the example below, the real 'message' is not visible until column 230, or, more typically, wrapped so that a useful message would appear on every third line. This is in a simple testbench (the Ubus example that ships with the UVM source code), with a shallow hierarchy and with a not very deep file structure. More complex file structures and hierarchies push the message further to the right.

```
UVM_INFO
/home/gmcgregor/src/sv/uvm_latest/distrib/examples/integrated/ubus/sv/ubus_master_monitor
.sv(205) @ 3030: uvm_test_top.ubus_example_tb0.ubus0.masters[0].monitor
[uvm_test_top.ubus_example_tb0.ubus0.masters[0].monitor] Covergroup 'cov_trans' coverage:
32.083332
```

Figure 1 Example message with all meta-data enabled

If we then disable the occasionally useful __FILE__ and __LINE__ context information from the message (by defining UVM_REPORT_DISABLE_FILE_LINE at compilation time) we get a message similar to the one shown in Figure 2.

```
UVM_INFO @ 3030: uvm_test_top.ubus_example_tb0.ubus0.masters[0].monitor
[uvm_test_top.ubus_example_tb0.ubus0.masters[0].monitor] Covergroup 'cov_trans' coverage:
32.083332
```

Figure 2 Example message removing __FILE__ and __LINE__ info

Even with the file information removed, the remaining meta-data, particularly the object name and context ID fields, push the message far to the right hand columns of the log, or wrapped around making the log structure difficult to visually parse. This example also highlights a common problem where the message id field is provided by *get_full_name()* or *get_type_name()*. This is in general a bad idea as it usually just adds redundant information to the log. The object name is already included prior to the context ID in any given message string. Using *get_type_name()* does occasionally provide some useful information, if type overrides are being used but often this is redundant in the log. This convention of using *get_full_name()* or *get_type_name()* was shown in some of the standard UVM examples and as a result has spread in tutorials. A better approach is to define useful additional contexts for messages and use those in place of repeating the type or object naming. (e.g., NOVIF for missing virtual interfaces, COV for coverage related messages and so on). Making these simplifications would give messages such as the one shown in Figure 3. In this case the message starts in column 79, right at the point that most tools will wrap messages and still continues on to the second line.

```
UVM_INFO @ 3030: uvm_test_top.ubus_example_tb0.ubus0.masters[0].monitor [COV] Covergroup
'cov_trans' coverage: 32.083332
```

Figure 3 Example message with abbreviated IDs

The style of message shown in Figure 3 is about as simplified as messages can get, without modifying the report handler. The desire to get more readable messages is often the motivation for changing this. There are also some additional reasons. The default messages do not indicate what level verbosity was used for a given message, which is a

strange oversight. Message verbosities, (e.g., UVM_HIGH, UVM_LOW, UVM_DEBUG, etc.) are a useful indicator of the level of importance of a given message – with the UVM_LOW and UVM_NONE verbosities being typically the most important messages, giving an overview of the test flow and the higher verbosity messages providing increasing detail. The default handler only provides the severity context for a message (e.g., UVM_INFO, UVM_ERROR). Another reason to change the message format is to use different time unit formatting (either by changing the units used or adding unit suffixes (e.g., ns). One final reason is to constrain the maximum size of message fields, such as the name and context id fields, so that for example, only the final 20 characters of the name are used, which is often sufficient to provide useful context without the entire hierarchy.

```
UVM_HIGH (3030ns) masters[0].monitor [COV] Covergroup 'cov_trans' coverage: 32.083332
```

Figure 4 Example custom message

Figure 4 provides an example of how a custom message might be structured, using the verbosity in place of severity for UVM_INFO messages, while potentially still using the severity for warnings and errors. The name field is truncated to the last two hierarchical identifiers and the context ID uses 'COV' to indicate the coverage context for the message. The main message in this case now starts on column 43, which is some improvement over the original column 230 with all meta-data included, in Figure 1.

Given this example, the motivation to modify the message handler is clear. Luckily the UVM provides a simple mechanism to achieve this.

3. Replacing the Standard Message Handler in UVM 1.1

Every object in the UVM extends from the basic *uvm_report_object* class, which implements the standard reporting methods. (*uvm_report_info*, *uvm_report_warning*, *uvm_report_error, uvm_report_fatal* and other supporting methods). The implementation of these methods delegate the reporting task to a *uvm_report_handler* instance and its associated *report* method. The *report* method in the handler implements some severity level filtering, then passes the message on to the singleton global instance of a *uvm_report_server.* The actual message composition and logging occurs in this global report server.

There are also global static versions of the reporting methods that are used when messages are generated that aren't within the context of a UVM object. These static implementations use an instance of *uvm_top* to access the reporting subsystem.

For each of the reporting methods, there are equivalent macro implementations. It is recommended that you use the macro versions of the reporting methods in most cases because of some efficiencies they provide and common errors they help avoid. The *`uvm_info* macro is a replacement for *uvm_report_info*. The macros will only call the underlying method if the verbosity controls are such that the message would be printed. Often messages are constructed using expensive string operations such as, *$sformatf* and particularly for high verbosity messages, these may be being called frequently while never being displayed. The macros help to curtail the processing overhead in these cases.

Using the macros also enforces a verbosity setting of UVM_NONE for the warning, error and fatal types of messages. This avoids mistakenly disabling these messages by setting a lower verbosity threshold. Particular warnings or errors can still be controlled via the report handler methods, but the macros avoid a common pitfall with the verbosity settings.

To change the default message format, you simply have to replace the global report server with your own implementation, as shown in Figure 5. This custom report server is sub classed from *uvm_report_server* and for the basic changes, such as those described in section 1, only needs to re-implement two methods; the constructor and the *compose message* method.

```
import uvm_pkg::*;
`include "uvm_macros.svh"

class custom_report_server extends uvm_report_server;
  uvm_report_server old_report_server;
  uvm_report_global_server global_server;

  function new(string name = "custom_report_server");
    super.new();
    set_name(name);
    global_server = new();
    old_report_server = global_server.get_server();
    global_server.set_server(this);
  endfunction

virtual function string compose_message(
      uvm_severity severity,
      string name,
      string id,
      string message,
      string filename,
      int    line
      );
    uvm_severity_type sv;
    string time_str;
    string line_str;

    sv = uvm_severity_type'(severity);
    $swrite(time_str, "%0t", $realtime);

    case(1)
      (name == "" && filename == ""):
               return {sv.name(), " @ ", time_str, " [", id, "] ", message};
      (name != "" && filename == ""):
               return {sv.name(), " @ ", time_str, ": ", name, " [", id, "] ", message};
      (name == "" && filename != ""):
            begin
                $swrite(line_str, "%0d", line);
                 return {sv.name(), " ",filename, "(", line_str, ")", " @ ", time_str, "
[", id, "] ", message};
            end
      (name != "" && filename != ""):
            begin
                $swrite(line_str, "%0d", line);
                  return {sv.name(), " ", filename, "(", line_str, ")", " @ ", time_str, ":
", name, " [", id, "] ", message};
            end
    endcase
  endfunction
```

Figure 5 Extending the default report server implementation for UVM 1.1

To use the custom server, create an instance of it within your test class as shown in Figure 6,

```
`include "custom_report_server.svh"

// Base Test
class example_base_test extends uvm_test;

  `uvm_component_utils(example_base_test)
   custom_report_server report_server = new();
```

Figure 6 Instantiating a custom report server

That's all that is required. The actual message customization occurs in the *compose_message*, where you can change the existing format, in each of the *return* lines, or replace the existing code to generate a return message format of your own from scratch.

4. Changes in UVM 1.2

The reporting subsystem in UVM version 1.2 is receiving a revamp, to streamline some of the method calls and present a more object-oriented API. The default message formats should be the same between UVM 1.1 and UVM 1.2 to keep the base class library backward compatible, particularly for tools that use log files as golden references.

Instead of passing each piece of meta-data as a separate argument, *uvm_report_message* objects have been introduced, which help to simplify message passing through the reporting hierarchy. Additionally the message verbosity is now contained within this message object, making it simpler to include this within the message output, just by changing the *compose_message* function. The relevant fields within the *uvm_report_message* object are shown in Figure 7.

```
class uvm_report_message extends uvm_object;

  uvm_severity_type severity;
  string id;
  string message;
  int verbosity;
  string filename;
  string context_name;
  uvm_action action;
```

Figure 7 The new UVM 1.2 report message object's fields

The basic approach for replacing the global report server remains the same, with a few small changes, as shown in Figure 8. Note in particular that the *uvm_report_server* constructor now takes a name argument, so there is no need for a distinct *set_name* call in the constructor of the new report server. The main change is that the compose function has been renamed to *compose_report_message* from *compose_message* and it now takes one *uvm_report_message* argument.

```
import uvm_pkg::*;
`include "uvm_macros.svh"

class custom_report_server extends uvm_report_server;
  uvm_report_server old_report_server;
  uvm_report_global_server global_server;

  function new(string name = "custom_report_server");
    super.new(name);
    global_server = new();
    old_report_server = global_server.get_server();
    global_server.set_server(this);
  endfunction


function string convert_verbosity_to_string(int verbosity);
    uvm_verbosity l_verbosity;

    if ($cast(l_verbosity, verbosity)) begin
        convert_verbosity_to_string = l_verbosity.name();
    end else begin
        string l_str;
        l_str.itoa(verbosity);
        convert_verbosity_to_string = l_str;
    end
  endfunction


virtual function string compose_report_message(uvm_report_message report_message);

    string sev_string;
    string verbosity_string;
    string filename_line_string;
    string time_str;
    string line_str;
    string context_str;

    sev_string = report_message.severity.name();
    verbosity_string = convert_verbosity_to_string(report_message.verbosity);

    if (report_message.filename != "") begin
      line_str.itoa(report_message.line);
      filename_line_string = {report_message.filename, "(", line_str, ") "};
    end

    // Make definable in terms of units.
    $swrite(time_str, "%0t", $time);

    if (report_message.context_name != "")
      context_str = {"@@", report_message.context_name};

    compose_report_message = {sev_string, " ", filename_line_string, "@ ",
      time_str, ": ", report_message.report_handler.get_full_name(), context_str,
      " [", report_message.id, "] ", report_message.convert2string()};

  endfunction
```

Figure 8 Extending the default report server implementation for UVM 1.2

Note that the verbosity is now available in the compose function and there is a new function *convert_verbosity_to_string*. As the verbosity is an integer with some defined intermediate values (*UVM_DEBUG, UVM_HIGH*) etc., the translation to a string has to consider the potential to use those intermediate values.

One common modification to make to the message format is to replace the severity string with the verbosity level, in cases where the severity is *UVM_INFO* and leave it as the severity string for the other levels of severity. This means that warnings, errors and fatals are marked as before, but adding additional INFO verbosity information, without

increasing the overall message width. This can be done easily in the compose function in the reporting hierarchy, once the verbosity is available.

In contrast, if you are using version 1.1 or older versions of the UVM, it is a little more involved to modify the library to access verbosity information in a message composition function. The approach, shown in Figure 9, is to override the function *report* that calls the *compose_message* function, to generate the message string. At this point, replace the *compose_message* call with a similar function, which takes an additional verbosity argument. This function then generates the message that will be passed on to the *process_report* function as normal.

```
  virtual function void report(

    // … portions of the report function removed for clarity

    if(report_ok) begin
      m = custom_compose_message(severity, verbosity_level, name, id, message, filename,
line);
      process_report(severity, name, id, message, a, f, filename,
                     line, m, verbosity_level, client);
    end
  endfunction

  virtual function string custom_compose_message (
      uvm_severity severity,
      int verbosity_level,
      string name,
      string id,
      string message,
      string filename,
      int    line
      );
    // … compose message as normal
```

Figure 9 Adding verbosity display information to a custom report server for UVM 1.1

So far, we've covered how to extend and replace the existing text based message logging server. This does allow a user to change the format of each message, but it has limited usefulness. Ideally, all of the pieces of meta-data about a message would be available and the user should be able to pick and chose dynamically which pieces of the message are shown. Occasionally you will care about the __FILE__ and __LINE__ information, so it should be available without re-running a simulation. At other times, the context information or hierarchical location of a message source is critical information and it should be possible to see it. The majority of the time this information just gets in the way of seeing what is happening in a given test. A more useful approach would be to allow the message composition to be controlled dynamically by a viewer, allowing the messages to be reformatted to change the information provided.

Perhaps more importantly than this reformatting of the message is the ability to selectively filter and fold messages throughout the logfile. A very useful capability is to change the verbosity level of the log messages shown; collapsing down to a UVM_NONE or UVM_LOW level of verbosity to understand the high-level flow of a test, and then expand particular sections to higher levels of verbosity to dive into the details of what is happening in a test. Additionally, being able to filter on the other meta-data in the message can be a very powerful technique; isolating messages from a single monitor or driver, tracing a given transaction through the log and so on.

It is possible to re-extract or infer all of this meta-data from a fully expanded text log, using regular expressions and similar parsing techniques, but there is a simpler, more reliable and more powerful method available, using standard techniques from the software world.

5. A Better Approach: Markup Log Formats

Markup languages are often used to create combined human- and machine-readable documents. In particular, the Extensible Markup Language (XML) [2] has seen widespread industry adoption. This standard format defines an approach based on tags to demark data, along with associated metadata. XML is designed to be extended to implement a variety of different document formats and is well suited to the logging problem discussed in this paper. Rather than composing the message along with selected pieces of metadata at runtime, we can store each message with all of the relevant metadata in an XML formatted log file. The selection of relevant metadata and construction of the final message to display can then be deferred to a later time. This log file can then post-processed in a variety of different ways to generate log messages.

Post processing an XML format is much less error-prone than using regular expressions to parse a plaintext log file. Changes in the metadata will not break tools that understand the existing attributes. The tools easily handle minor changes in format and there are a large number of standardized approaches to manipulating XML. Filtering standards, such as EXtensible Stylesheet Language (XSLT) [3], can be used to provide a style sheet to quickly process a log to present a designer-friendly overview of the flow of a test, without diving into the details. Log checkers can more accurately search for error conditions, by looking for actual errors, without being sensitive to valid, informational messages such as '*inserting an error*'. These types of messages about error injection often require waivers or obscure spellings (e.g., e_r_r_o_r) to avoid incorrectly indicating a failing test when plain text logs are used.

Given an XML based log it is also straightforward to develop interactive viewing and filtering tools or custom analyzers as will be shown later.

One criticism of XML is that it is somewhat heavyweight, with tags surrounding every feature in the file. Lighter weight formats like the JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML) that use structure and indentation were also considered but have been rejected in favor of XML, because of the improved reliability of the XML format. If stray messages get into the log, from other parts of the simulation (e.g., an IP using *$display* statements or other non-UVM messaging formats) these can be more easily recognized by an XML parser and captured as alternative message sources. In the lighterweight markup languages, these sort of unstructured messages injected in the log would cause parse errors that would be difficult to recover from in an elegant way.

For this application, we define an XML format, without actually implementing a formal schema. The structure is kept very simple. All log documents start with an XML header,

displayed by overriding the *report_header* method in the *uvm_report_server*. This method allows you to print a custom header prior to logging messages and is perfect for constructing an XML log file. Most of the header is standard boilerplate. Note that we provide a style sheet (*uvm.xsl*) which provides a mechanism to specify a standard way to render the log file in a web browser. The final tag of the header is the <*log*> tag, that is being defined here as the top-level container for all of the log messages to follow.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="uvm.xsl"?>
<log>
```

Figure 10 XML log header format

Similarly the log format is closed using a final tag, inserted in the *report_footer* function in the extension of *uvm_report_server*. This function is the pair of *report_header* and provides a useful entry point to close the log file and insert a final closing tag.

```
</log>
```

Figure 11 XML log footer format

Between the opening <*log*> and closing <*/log*> tags, the custom report server inserts tagged elements that represent each message output by the simulation. A message starts with an opening <*msg*> tag that has several attributes associated with it. These attributes are all of the meta-data that is defined for an UVM message. The main element of a <*msg*> is the actual text of the message string. Anything else detected in a log is considered a message coming from a non-UVM source so it can be easily extracted and captured.

```
<msg verbosity="100"  severity="UVM_INFO"
file="/home/gmcgregor/src/sv/uvm_latest/distrib/examples/integrated/ubus/sv/ubus_master_m
onitor.sv"  line="205"  id="uvm_test_top.ubus_example_tb0.ubus0.masters[0].monitor"
time="2580" >Covergroup cov_trans coverage: 32.083333</msg>
```

Figure 12 XML message format

6. Generating XML in SystemVerilog

The XML packaging for the message data is implemented in a very similar way to modifying the normal message format. The main change is in the final compose message function, which takes all the pieces of metadata and generates an appropriate XML message element. We define two generator functions, *xle()* and *xla()* that are used to compose the message, shown in Figure 13. *xla()* creates an attribute pair from a name string and a value. *xle()* similarly creates the overall element tag and combines the attributes and message string.

```
  // Function: xla
  // XML Attribute
  // Generate an XML attribute ( tag = "data" )
  function string xla(string tag, string data);
    xla="";
    if (data != "") begin
      xla = {" ", tag, "=\"", sanitize(data), "\" "};
    end
  endfunction

    // Function: xle
    // XML Element
    // Generate an XML element ( <tag attributes>data</tag> )
  function string xle(string tag, string data, string attributes="");
    xle = "";
    if (data != "") begin
      xle = {"<", tag, attributes, ">", sanitize(data), "</", tag, ">\n"};
    end
  endfunction
```

Figure 13 Generator functions for XML attributes and elements

Using these two generator functions, the final message is constructed as shown in Figure 14, bundling all of the metadata as attributes, attached to the message element.

```
    compose_xml_message = xle("msg", message,
                              {xla("verbosity", verbosity_str),
                               xla("severity", severity_string),
                               xla("file", filename),
                               xla("line", line_str),
                               xla("id", id),
                               xla("time", time_str),
                               xla("context", name) });
```

Figure 14 Generating an XML element in SystemVerilog

One final thing that has to be considered is handling special case characters within XML data. There are several reserved characters in XML that have special meaning, that need to be translated to predefined strings. Those are listed in the *replacements* associative array and the function *sanitize()* is used to switch any of these reserved characters within the output messages or metadata.

```
string replacements[string] = '{ "<" : "&lt;",
                                  "&" : "&amp;",
                                  ">" : "&gt;",
                                  "'" : "&apos;",
                                  "\"": "&quot;"
                                };

  // Function: sanitize
  //
  // Given an unencoded input string, replaces illegal characters for XML data format
  function string sanitize(string data);

    for(int i = data.len()-1; i >= 0; i--) begin
      if (replacements.exists(data[i])) begin
          data = {data.substr(0,i-1), replacements[data[i]], data.substr(i+1, data.len()-
1)};
      end
    end
    return data;
  endfunction : sanitize
```

Figure 15 Handling special case characters for XML formating

7. Custom Log Reconstruction

It is straightforward to write scripts to reconstruct a log file from an XML log generated in the style described in Section 6. Figure 16 demonstrates how this can be done in the Python scripting language. Python has standard libraries that can be used for XML parsing and it is easy to write a filter that can parse and output a customizable view of any given log.

```
from xml.etree.ElementTree import parse

data=parse('log.xml')
log=data.getroot()

for msg in log:
  if eval(msg.attrib['verbosity']) < 101:
    print msg.attrib['verbosity'], msg.attrib['severity'], msg.text
```

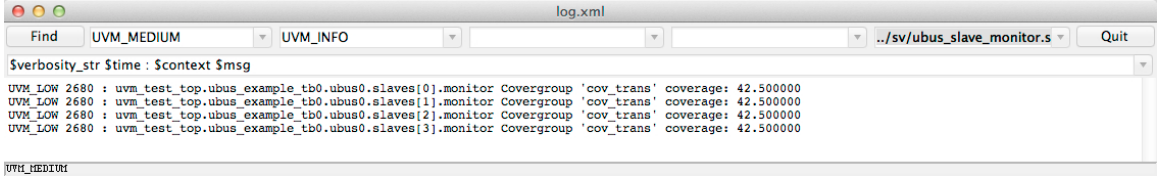
Figure 16 Python example of log file processing

This sort of scripted filtering can be useful for a variety of applications, producing a standard text log that summarizes a test, for use in continuous integration tools. It can be useful to provide a quick overview of what the flow was, while hiding much of the detail. The script in Figure 16 is also a good basis for a log file scanner, checking for particular known error conditions or other expected messages (e.g., an “END OF SIMULATION” message that might be required to indicate a successful test completion)

8. GUI to Interactively Filter Messages

The static scripted parsing of log files discussed in the previous section is useful for many applications. However, interactive manipulation of the log file is particularly powerful when trying to debug a given failure. As a proof of concept a simple GUI viewer was created. This GUI demonstrates the simplicity of the parser when using an XML log and also the powerful features that can be enabled.

Dynamically reformatting the messages, as shown in Figure 17, is beneficial to the end user. In the upper screenshot, the messages are formatted to show verbosity, timestamp, context string and the text message itself. In the lower snapshot, the format string has been changed to include the originating file and line, and the context string has been removed. This change can be done using a dropdown box that contains a variety of predefined formats, or a new format string can be typed into the same dropdown box. This allows the messages to be tuned to suit a particular debugging task, instantly. It also allows a user to see all of the metadata they are interested in, while easily hiding additional information that is cluttering the log. Enhancements could provide all of the relevant metadata for a log as a tooltip when the mouse cursor is hovered over a message. As all of the metadata is available, it is relatively easy to link directly to the originating point of the message in the source code.



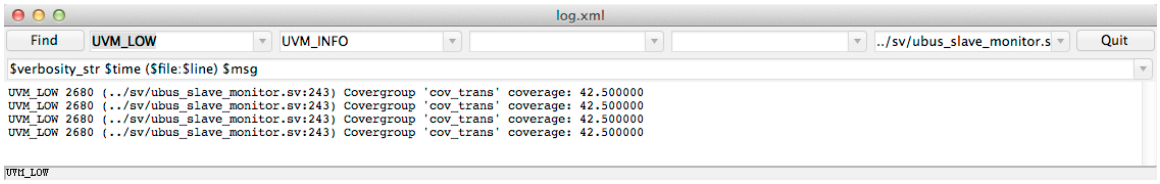
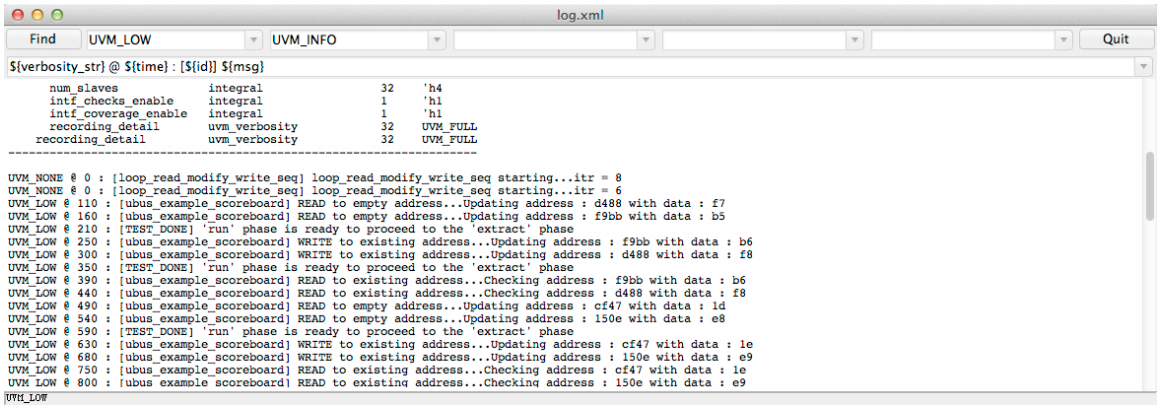


Figure 17 Example of dynamically reformatting a message

With the additional metadata present in the file, it is also possible to provide filtering tools to collapse and expand the visible log messages, based on all of the metadata fields. Figure 18 demonstrates filtering based on the verbosity of the message string, with the upper screenshot showing messages at UVM_LOW or lower levels and the lower screenshot showing the same portion of the log, with UVM_HIGH and lower levels of verbosity enabled. Similarly, Figure 19 demonstrates filtering the message based on the ID context of the log.



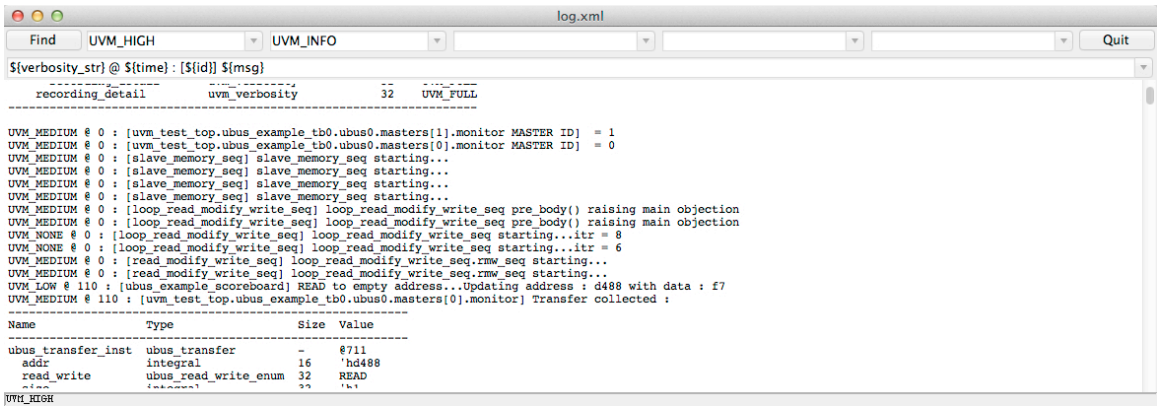


Figure 18 Example of filtering on verbosity

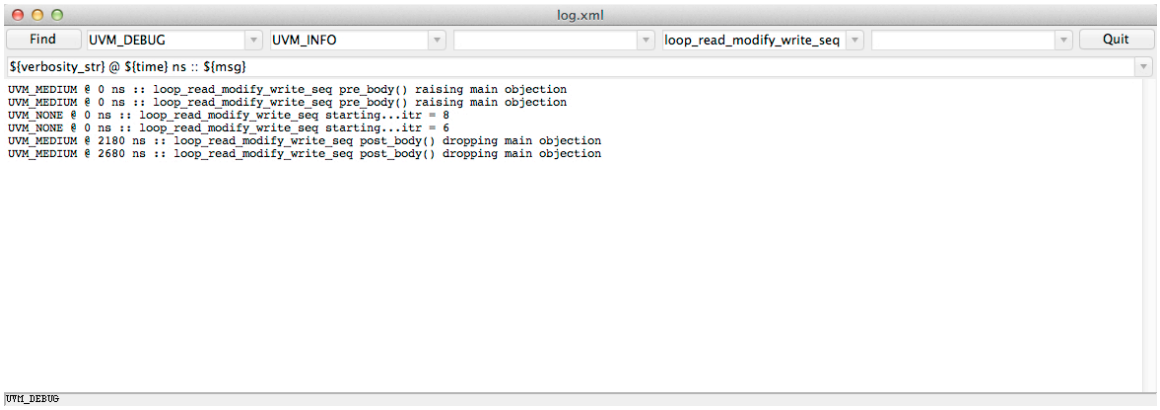


Figure 19 Filtering on a message id

While this sort of filtering is possible in existing tools (with the exception of verbosity) it is much simpler to provide a feature rich and flexible way of doing this using a markup based log format. It is also simpler to implement useful features such as syntax highlighting and hierarchical folding when the metadata is already separated.

9. Managing Large Log Files

Long simulation runs and high verbosity settings will produce extremely large log files. I do not advocate always running at the UVM_DEBUG level of verbosity, but have found it very useful to capture at a level one or two higher than is often used, then selectively view the details using log viewers as described in this paper. Markup and XML in particular actually make these larger log files potentially easier to manage. As individual messages are demarked in the log with tags and parsed as units in the various XML tools, it is simpler to step through larger log files and just show a view on the contents. Tools do not have to read in the entire log but can operate on sliding windows to help reduce the overhead of extremely large log files from long simulation runs. XML has two main parsing styles, XML Document Object Model (DOM), which uses tree model and loads the entire log and Simple API for XML (SAX) which parses the log using an event based, callback model to only process one element at a time. SAX could potentially be used to reduce the memory footprint of processing a large log. The examples provided in this paper are all DOM based parsers.

It is also possible to use similar techniques as described in this paper to store the data in an SQL or SQLite database rather than to XML. In that case, each message is tagged with a unique id number and it is even simpler to manage iterating through the messages. SQL has similar advantages of storing all of the metadata with the message and allowing later reconstruction of the desired messages. An example of how to log to SQL is included in the sample code made available along with this paper.

10. Conclusions and Future Work

The logging format and viewer have been implemented and have proven to be useful and effective in debugging UVM simulations. Very little modification is required to an existing testbench to implement this XML logging and it has a low overhead on the simulation time, compared to normal logging. The viewer tools have also been useful in

debugging and analyzing log reports more efficiently than with the existing plaintext files.

The current implementation stores messages to a separate log containing the XML messages, and leaves the normal unprocessed stdout log with UVM messages and any messages from other parts of the simulation (DPI C-code, third-party IP that uses $display messages and so on). Ideally, the stdout message log should contain the XML markers so that non-UVM messages can be incorporated into any viewer tools. The parser would then need to handle the exceptions caused by having these messages in the file, as they would not be properly formatted XML. Any of these strings that cause exceptions could be captured and stored in a non-UVM category that could be displayed inline or filtered, appropriately. The current implementation of the viewer does not implement this feature and should be added.
Also, the viewer should be able to support folding of messages (e.g., by verbosity) and it is planned to add this feature. Similarly, syntax highlighting, or color-coding of sources, highlighting numbers and similar pieces of interesting data will be implemented. Optimizations to support larger log files and more efficiently parse the data need to be implemented. XML parsers do provide mechanisms to handle large data sets efficiently, but the current implementation of the viewer loads the entire log file into memory and parses it. This could be more efficiently implemented.

11. Source Code

The source code for the XML logging report server along with examples using other markup formats such as JSON, HTML and YAML are available online [6]. There is also an SQLite logging implementation available.

12. References

1. UVM Accellera standard, *http://www.accellera.org/downloads/standards/uvm*
2. XML standard, *http://www.w3.org/TR/REC-xml/*
3. XSLT standard, *http://www.w3.org/TR/xslt20/*
4. DVE SmartLog, *http://www.synopsys.com*
5. DVT Eclipse Smart Log, *http://www.dvteclipse.com/documentation/sv/UVM_Smart_Log.html*
6. UVM Logging source code, https://bitbucket.org/verilab/uvm_structured_logs

13. Unified UVM_1_1 and UVM_1_2 XML Server

```
//------------------------------------------------------------------------
//   Copyright 2012 Verilab Inc.
//   Gordon McGregor (gordon.mcgregor@verilab.com)
//
//   All Rights Reserved Worldwide
//
//   Licensed under the Apache License, Version 2.0 (the
//   "License"); you may not use this file except in
//   compliance with the License.  You may obtain a copy of
//   the License at
//
//       http://www.apache.org/licenses/LICENSE-2.0
//
//   Unless required by applicable law or agreed to in
//   writing, software distributed under the License is
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
//   CONDITIONS OF ANY KIND, either express or implied.  See
//   the License for the specific language governing
//   permissions and limitations under the License.
//------------------------------------------------------------------------

import uvm_pkg::*;
`include "uvm_macros.svh"

// Class: xml_report_server
//
// Replacement for uvm_report_server to log messages to an XML format
// that can be more easily reused and manipulated by external tools/ viewers
//
// Basic XML schema
// <log>
//   <msg verbosity="val" severity="string" file="filename" line="val" id="string"
time="val" context="string(optional)">text</msg>
//   ...
//   <msg verbosity="val" severity="string" file="filename" line="val" id="string"
time="val" context="string(optional)">text</msg>
//   <msg verbosity="val" severity="string" file="filename" line="val" id="string"
time="val" context="string(optional)">text</msg>
//   ...
// </log>
//
class xml_report_server extends uvm_report_server;

  uvm_report_server old_report_server;
  uvm_report_global_server global_server;

  // characters that are invalid XML that have to be encoded
  string replacements[string] = '{ "<" : "&lt;",
                                   "&" : "&amp;",
                                   ">" : "&gt;",
                                   "'" : "&apos;",
                                   "\"": "&quot;"
                                 };
  integer logfile_handle;

  //Function: new
  // constructor
  function new(string name = "xml_report_server", log_filename = "log.xml");
`ifdef UVM_VERSION_1_2
    super.new(name);
`else
    super.new();
    set_name(name);
`endif

    global_server = new();
    install_server();
    logfile_handle = $fopen(log_filename, "w");
    report_header(logfile_handle);
```

```
  endfunction

  // Function: install_server
  // replace the global server with this server
  function void install_server;
    old_report_server = global_server.get_server();
    global_server.set_server(this);
  endfunction

  // Function: enable_xml_logging
  // Configure all components to use UVM_LOG actions to trigger XML capture
  // has to be called after components have been instantiated (end of elaboration, run
etc)
  function void enable_xml_logging(uvm_component base=null);
    uvm_root top;

    if (base == null) begin
      top = uvm_root::get();
      base = top;
    end

    base.set_report_default_file_hier(logfile_handle);
    base.set_report_severity_action_hier(UVM_INFO, UVM_DISPLAY | UVM_LOG);
    base.set_report_severity_action_hier(UVM_WARNING, UVM_DISPLAY | UVM_LOG);
    base.set_report_severity_action_hier(UVM_ERROR, UVM_DISPLAY | UVM_LOG | UVM_COUNT);
    base.set_report_severity_action_hier(UVM_FATAL, UVM_DISPLAY | UVM_LOG | UVM_EXIT);
    base.get_report_handler().dump_state();
  endfunction

  // Function: convert_verbosity_to_string
  // Helper function to convert verbosity value to appropriate string, based on
uvm_verbosity enum if an equivalent level
  function string convert_verbosity_to_string(int verbosity);
    uvm_verbosity l_verbosity;

    if ($cast(l_verbosity, verbosity)) begin
        convert_verbosity_to_string = l_verbosity.name();
    end else begin
        string l_str;
        l_str.itoa(verbosity);
        convert_verbosity_to_string = l_str;
    end
  endfunction

  // Function: report_header
  // Output standard XML header to log file
  function void report_header(UVM_FILE file = 0);
    f_display(file, "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<?xml-stylesheet
type=\"text/xsl\" href=\"uvm.xsl\"?><log>\n");
  endfunction

  // Function: report_footer
  // Output XML closing tags to log file
  function void report_footer(UVM_FILE file = 0);
    f_display(file, "</log>");
  endfunction

`ifdef UVM_VERSION_1_2
  // Function: report_summarize
  // tidy up logging and restore global report server
  function void report_summarize(UVM_FILE file = 0);
`else
  // Function: summarize
  // tidy up logging and restore global report server
  function void summarize(UVM_FILE file = 0);
`endif
    report_footer(logfile_handle);
    global_server.set_server(old_report_server);
    $fclose(logfile_handle);
`ifdef UVM_VERSION_1_2
    old_report_server.report_summarize(file);
`else
    old_report_server.summarize(file);
```

```
`endif
  endfunction

`ifdef UVM_VERSION_1_2
  // Function: execute_report_message
  //
  // Processes the message's actions.
  virtual function void execute_report_message(uvm_report_message report_message);
    if(uvm_action_type'(report_message.action) == UVM_NO_ACTION)
      return;

    // Update counts
    incr_severity_count(report_message.severity);
    incr_id_count(report_message.id);

    // Process UVM_RM_RECORD action (send to recorder)
    if(report_message.action & UVM_RM_RECORD) begin
      report_message.record_message(uvm_default_recorder);
    end

    // Process UVM_DISPLAY and UVM_LOG action (send to logger)
    if((report_message.action & UVM_DISPLAY) || (report_message.action & UVM_LOG)) begin
      // DISPLAY action
      if(report_message.action & UVM_DISPLAY) begin
        $display("%s", compose_report_message(report_message));
      end
      // LOG action
      // if log is set we need to send to the file but not resend to the
      // display. So, we need to mask off stdout for an mcd or we need
      // to ignore the stdout file handle for a file handle.
      if(report_message.action & UVM_LOG) begin
        if( (report_message.file == 0) ||
          (report_message.file != 32'h8000_0001) ) begin //ignore stdout handle
          UVM_FILE tmp_file = report_message.file;
          if((report_message.file & 32'h8000_0000) == 0) begin //is an mcd so mask off
stdout
            tmp_file = report_message.file & 32'hffff_fffe;
          end
        f_display(tmp_file, compose_log_report_message(report_message));
        end
      end
    end

    // Process the UVM_COUNT action
    if(report_message.action & UVM_COUNT) begin
      if(get_max_quit_count() != 0) begin
        incr_quit_count();
        // If quit count is reached, add the UVM_EXIT action.
        if(is_quit_count_reached()) begin
          report_message.action |= UVM_EXIT;
        end
      end
    end

    // Process the UVM_EXIT action
    if(report_message.action & UVM_EXIT) begin
      uvm_root l_root = uvm_root::get();
      l_root.die();
    end

    // Process the UVM_STOP action
    if (report_message.action & UVM_STOP)
      $stop;

  endfunction
`else
  // Function: process_report
  //
  // Processes the message's actions.
  virtual function void process_report(
    uvm_severity severity,
    string name,
    string id,
```

```
    string message,
    uvm_action action,
    UVM_FILE file,
    string filename,
    int line,
    string composed_message,
    int verbosity_level,
    uvm_report_object client
    );
    // update counts
    incr_severity_count(severity);
    incr_id_count(id);

    if(action & UVM_DISPLAY)
      $display("%s",composed_message);

    // if log is set we need to send to the file but not resend to the
    // display. So, we need to mask off stdout for an mcd or we need
    // to ignore the stdout file handle for a file handle.
    if(action & UVM_LOG)
      if( (file == 0) || (file != 32'h8000_0001) ) //ignore stdout handle
      begin
        UVM_FILE tmp_file = file;
        if( (file&32'h8000_0000) == 0) //is an mcd so mask off stdout
        begin
          tmp_file = file & 32'hffff_fffe;
        end
        composed_message = compose_xml_message(severity, verbosity_level, name, id,
message, filename, line);
        f_display(tmp_file, composed_message);
      end

    if(action & UVM_EXIT) client.die();

    if(action & UVM_COUNT) begin
      if(get_max_quit_count() != 0) begin
        incr_quit_count();
        if(is_quit_count_reached()) begin
          client.die();
        end
      end
    end

    if (action & UVM_STOP) $stop;

  endfunction
`endif

  // Function: sanitize
  //
  // Given an unencoded input string, replaces illegal characters for XML data format
  function string sanitize(string data);

    for(int i = data.len()-1; i >= 0; i--) begin
      if (replacements.exists(data[i])) begin
          data = {data.substr(0,i-1), replacements[data[i]], data.substr(i+1, data.len()-
1)};
      end
    end
    return data;
  endfunction : sanitize

  // Function: xla
  // XML Attribute
  // Generate an XML attribute ( tag = "data" )
  function string xla(string tag, string data);
    xla="";
    if (data != "") begin
      xla = {" ", tag, "=\"", sanitize(data), "\" "};
    end
  endfunction

    // Function: xle
```

```
    // XML Element
    // Generate an XML element ( <tag attributes>data</tag> )
  function string xle(string tag, string data, string attributes="");
    xle = "";
    if (data != "") begin
      xle = {"<", tag, attributes, ">", sanitize(data), "</", tag, ">\n"};
    end
  endfunction

`ifdef UVM_VERSION_1_2
  // Function: compose_log_report_message
  // Generate the XML encapsulated report message, for logging
  function string compose_log_report_message(uvm_report_message report_message);
    string severity_string;
    string verbosity_str;
    string time_str;
    string line_str;
    string context_str;

    severity_string = report_message.severity.name();
    void'(verbosity_str.itoa(report_message.verbosity));
    void'(line_str.itoa(report_message.line));
// Make definable in terms of units.
    $swrite(time_str, "%0t", $time);

    compose_log_report_message =  xle("msg", report_message.convert2string(),
                                  {xla("verbosity", verbosity_str),
                                   xla("severity", severity_string),
                                   xla("file", report_message.filename),
                                   xla("line", line_str),
                                   xla("id", report_message.id),
                                   xla("time", time_str),
                                   xla("context", report_message.context_name) });

  endfunction
`else
  // Function: compose_xml_message
  // Generate the XML encapsulated report message, for logging
  virtual function string compose_xml_message(
    uvm_severity severity,
    int verbosity,
    string name,
    string id,
    string message,
    string filename,
    int    line
    );
    uvm_severity_type sv;
    string severity_string;
    string time_str;
    string line_str;
    string verbosity_str;

    sv = uvm_severity_type'(severity);
    severity_string = sv.name();
    $swrite(time_str, "%0t", $time);
    void'(line_str.itoa(line));
    void'(verbosity_str.itoa(verbosity));

    compose_xml_message = xle("msg", message,
                              {xla("verbosity", verbosity_str),
                               xla("severity", severity_string),
                               xla("file", filename),
                               xla("line", line_str),
                               xla("id", id),
                               xla("time", time_str),
                               xla("context", name) });
  endfunction
`endif
endclass
```

14. Deployment of XML report server

```
`include "xml_report_server.svh"

class example_base_test extends uvm_test;

  `uvm_component_utils(example_base_test)
   xml_report_server xml_server = new();
```

15. UVM XML Style sheet for log parsing

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
  <h2>UVM Log</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Severity</th>
        <th>Verbosity</th>
        <th>Time</th>
        <th>Text</th>
      </tr>
      <xsl:for-each select="log/msg[@verbosity &lt; 401]">
      <tr>
        <td><xsl:value-of select="@severity"/></td>
        <td><xsl:value-of select="@verbosity"/></td>
        <td><xsl:value-of select="@time"/></td>
        <td><span><xsl:attribute name="title"><xsl:value-of
select="@file"/></xsl:attribute> <pre><xsl:value-of select="."/></pre></span></td>
      </tr>
      </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

16. Python XML Log viewer

```
#!/usr/bin/env python3
from tkinter import *
from tkinter.ttk import *
from tkinter.simpledialog import askstring
from tkinter.filedialog   import asksaveasfilename

from tkinter.messagebox import askokcancel

from xml.etree.ElementTree import parse

import string

DEFAULT_FORMATS = ("${severity}::${verbosity_str} ${file}(${line}) @ ${time} : ${context}
[${id}] ${msg}",
                   "${verbosity_str} @ ${time} : ${context} [${id}] ${msg}",
                   "${time} : ${context} ${msg}")

uvm_verbosities = {"UVM_NONE":0, "UVM_LOW":100, "UVM_MEDIUM":200, "UVM_HIGH":300,
"UVM_FULL":400, "UVM_DEBUG":500}
uvm_verbosity_strings = {uvm_verbosities[v]:v for v in uvm_verbosities}

class ScrolledText(Frame):
    def __init__(self, parent=None, text='', file=None):
        Frame.__init__(self, parent)
        self.pack(expand=YES, fill=BOTH)
        self.makewidgets()
        self.settext()
    def makewidgets(self):
        self.sbar = Scrollbar(self)

        text = Text(self, relief=SUNKEN)
        self.sbar.config(command=text.yview)
        text.config(yscrollcommand=self.sbar.set)
        self.status = StringVar()
        self.status_widget = Label(self, textvar = self.status, relief=SUNKEN)
        self.status.set('status bar')
        self.status_widget.config(font=('courier', 10, 'normal'))
        self.status_widget.pack(side=BOTTOM, fill=X)
        self.sbar.pack(side=RIGHT, fill=Y)
        text.pack(side=LEFT, expand=YES, fill=BOTH)
        self.text = text


    def settext(self):
        self.text.config(state=NORMAL)
        scroll_location = self.text.yview()
        if self.file:
            text = self.parse_log()
        self.text.delete('1.0', END)
        self.text.insert('1.0', text)
        self.text.mark_set(INSERT, '1.0')
        self.text.focus()
        self.text.config(state=DISABLED)
        self.text.yview('moveto', scroll_location[0])

    def gettext(self):
        return self.text.get('1.0', END+'-1c')

    def parse_log(self):
        log_data = ""
        data=parse(self.file)
        log=data.getroot()
        verbosity_value = 101
        verbosity_level = self.verbosity.get()
        if verbosity_level in uvm_verbosities:
            verbosity_value = uvm_verbosities[verbosity_level] + 1

        severity_value = self.severity.get()
```

```
        context_value = self.context.get()
        id_value = self.id.get()
        source_file_value = self.filer.get()


        message_template = string.Template(self.format.get() + '\n')
        for msg in log:
          if eval(msg.attrib['verbosity']) < verbosity_value:

            if context_value != ' ':
                if msg.attrib['context'] != context_value:
                    continue

            if id_value != ' ':
                if msg.attrib['id'] != id_value:
                    continue
            if source_file_value != ' ':
                if msg.attrib['file'] != source_file_value:
                    continue
            if severity_value != ' ':
                if msg.attrib['severity'] != severity_value:
                    continue
            try:
                verbosity_string = uvm_verbosity_strings[eval(msg.attrib['verbosity'])]
            except:
                verbosity_string = msg.attrib['verbosity']
            log_data += message_template.substitute({x:msg.attrib[x] for x in
msg.keys()}, verbosity_str = verbosity_string, msg=msg.text)
        return log_data

    def pre_parse_log(self):
        data=parse(self.file)
        log=data.getroot()

        severity_set = set()
        file_set = set()
        id_set = set()
        context_set = set()
        for msg in log:
            severity_set.add(msg.attrib['severity'])
            file_set.add(msg.attrib['file'])
            id_set.add(msg.attrib['id'])
            context_set.add(msg.attrib['context'])

        return(severity_set, file_set, id_set, context_set)


class SimpleEditor(ScrolledText):
    def __init__(self, parent=None, file=None):
        self.root = Tk()
        self.file = file
        frm = Frame(parent)
        self.root.title(self.file)

        frm.pack(fill=X)

        self.format = StringVar()
        self.format_widget = Combobox(frm, values = DEFAULT_FORMATS, textvar =
self.format)
        self.format_widget.pack(side=BOTTOM, fill=X)

        self.format_widget.current(0)

        self.format_widget.bind('<Return>', self.updateLog)
        self.format_widget.bind('<<ComboboxSelected>>', self.updateLog)

        Button(frm, text='Find',  command=self.onFind).pack(side=LEFT)

        (severity_set, file_set, id_set, context_set) = self.pre_parse_log()

        values = list(uvm_verbosities.keys())
        values.sort(key = lambda x:uvm_verbosities[x])
        self.verbosity = Combobox(frm, values = values , state='readonly')
```

```
        self.verbosity.bind('<<ComboboxSelected>>', self.updateLog)
        self.verbosity.set("UVM_LOW")
        self.verbosity.pack(side=LEFT)

        severities = list(severity_set)
        severities.sort()
        self.severity = Combobox(frm, values = [' ',] + severities, state='readonly')
        self.severity.current(0)
        self.severity.bind('<<ComboboxSelected>>', self.updateLog)
        self.severity.pack(side=LEFT)

        contexts = list(context_set)
        contexts.sort()
        self.context = Combobox(frm, values = [' ',] + contexts, state='readonly')
        self.context.current(0)
        self.context.bind('<<ComboboxSelected>>', self.updateLog)
        self.context.pack(side=LEFT, expand=YES, fill=X)

        ids = list(id_set)
        ids.sort()
        self.id = Combobox(frm, values = [' ',] + ids, state='readonly')
        self.id.current(0)
        self.id.bind('<<ComboboxSelected>>', self.updateLog)
        self.id.pack(side=LEFT, expand=YES, fill=X)

        files = list(file_set)
        files.sort()
        self.filer = Combobox(frm, values = [' ',] + files, state='readonly')
        self.filer.current(0)
        self.filer.bind('<<ComboboxSelected>>', self.updateLog)
        self.filer.pack(side=LEFT, expand=YES, fill=X)

        Button(frm, text='Quit', command=frm.quit).pack(side=LEFT)

        ScrolledText.__init__(self, parent, file=file)
        self.text.config(font=('courier', 12, 'normal'))

    def onFind(self):
        target = askstring('SimpleEditor', 'Search String?')
        if target:
            where = self.text.search(target, INSERT, END)
            if where:
                print(where)
                pastit = where + ('+%dc' % len(target))
                self.text.tag_add(SEL, where, pastit)
                self.text.mark_set(INSERT, pastit)
                self.text.see(INSERT)
                self.text.focus()

    def updateLog(self, *args):
        self.status.set(self.verbosity.get())
        self.settext()

if __name__ == '__main__':
    try:
        SimpleEditor(file=sys.argv[1]).mainloop()
    except IndexError:
        SimpleEditor().mainloop()
```