UVM Report Catcher

There are situations where you may need to change a message generated by the messaging system, and the uvm_report_catcher is built-in call-back mechanism for doing this. Typical applications might be to downgrade an error to a warning, or to modify a message. The report catcher can modify the severity, verbosity, id, action or the string content of a message before it is passed to the message server for printing.

The report catcher is implemented as an extended version of the uvm_callbacks object. Multiple report catchers can be registered and potentially each message will be processed by all the registered call-backs in the order in which

they were registered. The report catcher's catch() method is called as the message is generated, and the implementation of the catch() method determines how the message is processed. If the catch() method returns a THROW value, then the message is passed onto other registered report catchers. If it returns a CAUGHT value then the message is passed to the report server. Inside the catch() method, the issue() call can be made which will immediately send the message to the report server.

Each uvm_report_catcher object needs to be constructed and then registered either as a global report catcher or as a report catcher for a group of components or a single component.

Report Catcher Examples

The first report catcher example defines a call-back that checks for a message with the "green_id" and if the message type is UVM_INFO, then it modifies the message. The same call-back also demotes any errors with a "green_id" to warnings. The call back returns a THROW value, which means that the message will be passed to the next report catcher call-back, if there is one.

```
// Example report catchers:
class message_mod extends uvm_report_catcher;
`uvm_object_utils(message_mod)

function new(string name = "message_mod");
  super.new(name);
endfunction

function action_e catch();
  string message;
  string id;

  if((get_id() == "green_id") & (get_severity() == UVM_INFO)) begin
    set_message("Message modified");
  end
  else if((get_id() == "green_id") & (get_severity() == UVM_ERROR))
begin
    set_severity(UVM_WARNING);
    set_message("This warning was an error");
  end

  return THROW;

endfunction
```

```
endclass
```

The following report catcher example is designed to catch a UVM_FATAL severity message with a "red_id" and demote it to an error. This type of report catcher might be useful in the early stages of debugging a testbench as a means of continuing past a fatal error.

```
class fatal_mod extends uvm_report_catcher;
`uvm_object_utils(fatal_mod)

function new(string name = "fatal_mod");
  super.new(name);
endfunction

function action_e catch();
  string message;
  string id;

  if((get_id() == "red_id") & (get_severity() == UVM_FATAL)) begin
    set_message("Something went very wrong but was demoted to error");
    set_severity(UVM_ERROR);
  end
  return THROW;

endfunction

endclass
```

The following UVM code shows how these two call-backs are declared, constructed and then registered. The message_mod call back is registered only against the env.green component, so it will only be called when an object in the env.green generates a message. The fatal_demoter call-back has its add() method type argument set to null, which means that it applies to all messages generated in the UVM testbench.

```
// A test class that uses them:
class message_test extends uvm_component;

message_env env;
message_mod mess_mod;
fatal_mod fatal_demoter;

function void build_phase(uvm_phase phase);
  env = message_env::type_id::create("env", this);
  mess_mod = new("mess_mod");
  fatal_demoter = new("fatal_demoter");
  // The Message_mod report catcher is only applied to the env.green
component:
  uvm_report_cb::add(env.green, mess_mod);
  // The fatal_mod report catcher is applied to all component messaging
  uvm_report_cb::add(null, fatal_demoter);
```

```
endfunction
```

At the end of the UVM simulation, the report server will generate a summary of all the fatal, error and warning messages that had their severity demoted using the report catcher.

Testing Message Status

At the end of a UVM simulation, the report server issues a messaging summary to the transcript of the simulation. This will detail the number of each type of message severity that has been generated by the messaging system. If the simulation has been run with no report modifications, then this summary can be parsed and used as part of determining whether the simulation passed or not. For instance, the test case might be deemed to have passed if the scoreboard issues a pass message and there are no UVM_ERRORS or UVM_WARNINGS.

However, there may be more complicated scenarios where messages have been downgraded, or a specific error should have been provoked a certain number of times during the test case. In this situation the report server can be queried during the report phase to determine whether the test behavior is as expected.

The report server has two methods that are useful for this:

```
function int get_id_count(string id);
function int get_severity_count(uvm_severity severity);
```

The following code is an example of how these API calls might help to determine whether a test has passed or failed:

```
    function void report_phase(uvm_phase phase);
      uvm_coreservice_t cs;
      uvm_report_server svr;
      cs = uvm_coreservice_t::get();
      svr = cs.get_report_server();

      if (svr.get_severity_count(UVM_FATAL) == 0 &&
          ((svr.get_id_count("ASSERT_PARITY_ERROR") == 5)  &&
(svr.get_severity_count(UVM_ERROR) == 5))) begin
         $write("** UVM TEST PASSED **\n");
      end
      else begin
         $write("!! UVM TEST FAILED !!\n");
      end
    endfunction
```

In this example, the test is expected to provoke 5 parity errors, therefore if there not 5 UVM_ERROR messages reported with the "ASSERT_PARITY_ERROR" id, then something has gone wrong in the test.

Note: - This approach to determining whether a test has passed or not is only relevant if all the messages generated in the UVM testbench code have used the UVM messaging system. It should be used in combination with other checks, like checking the status of a scoreboard. For instance, you could have a scenario where there are no UVM_ERRORs reported, but the test has failed because no transactions were sent to the DUT. Also, if assertions are being used, it would be unusual for those to be using the UVM messaging system, so you typically need an alternative way of checking for any assertion errors.