2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Extending functionality of UVM components by using Visitor design pattern

Darko M. Tomušilović
Vtool LTD
Belgrade, Serbia
darkot@thevtool.com

***Abstract*—Visitor is a software design pattern used to add a new operation to classes in an existing class structure, suitable to be implemented for classes within a well-defined class hierarchy, such is the UVM based environment. All the necessary infrastructure for the usage of Visitor is provided within UVM library. The paper will briefly introduce the pattern elements and show examples how the functionality can be incorporated into the verification environment.**

Keywords—visitor design pattern, UVM

I. INTRODUCTION

Over the course of verification environment development cycle, a common requirement that developers face is to add a new operation to each class in existing class hierarchy. The operations may include check of correct static and dynamic configuration, environment connectivity check, reporting and statistics collection, etc.

The most common solution the developers might opt for is the most obvious one: adding code that will perform each operation into each class in the environment. Unfortunately, in many cases, the approach is not even possible to implement due to limitations such as encrypted codebase of VIPs bought from EDA vendors, or company standard proven code, that might become unstable by applying changes to it. Also applying such solution might pollute the existing codebase. The other option of creating derived classes that would perform newly added operations might lead to a very inflexible system, as all the places the base classes are used would need to be changed. This way, code readability and understanding can also be seriously hindered.

However, the alternative to this approach has been well established in the software development world, in a form of Visitor design pattern, and can be equally applicable to tackle similar challenges in the verification world.

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates." [1]

Visitor is a software design pattern used to add a new operation to each class in an existing class structure, suitable to be implemented for classes within a well-defined class hierarchy, such is the UVM based environment. According to Visitor design pattern principles, each operation should be encapsulated within a dedicated visitor class. Managing the operation in a single place – within a dedicated class, rather than having it spread across the whole environment, in its each and every class, significantly decreases code complexity and facilitates its maintenance. As all the changes to be done when extending the functionality are localized, the code stability and overall code quality are improved. An additional ability Visitor brings is to extend a class functionality where the class code cannot be changed.

The situation in which the Visitor can be utilized to its full power is to perform certain action on every item in a defined collection. The object of a Visitor class then traverses each node in the tree, and according to the type of visited object, performs a different set of actions. As a general rule of thumb, Visitor should be applied if the class structure is stable and unlikely to change, otherwise it can create an unnecessary overhead. While adding a new operation using a Visitor is straightforward, adding a new type of object a Visitor should visit is costly, as it requires changes in core Visitor code.

As it relies on polymorphism, the usage of Visitor should be planned in advance, so that every class in the environment would be compliant with the Visitor flow. Fortunately, UVM library provides all the necessary support, and therefore any class directly or indirectly extending from *uvm_component* is capable of handling Visitor access.

II. VISITOR DESIGN PATTERN INFRASTRUCTURE

UVM library [2] defines two main elements that represent the Visitor design pattern infrastructure:

1. *uvm_visitor* – abstract class defining a general *visit* operation on a node. For each of new functionalities the node should support, a concrete visitor that extends *uvm_visitor* is defined, giving implementation to *visit* operation according to the action the visitor needs to accomplish. A simple example is shown in Figure 1. It presents a concrete visitor that displays the name of the component that is being visited. *uvm_visitor* also contains pre-processing and post-processing hooks *begin_v* and *end_v* that can be utilized for initialization and activity summary, respectively. In the example, the hooks are not utilized and are left empty by inheritance.

   It is worth noting that in this example, the visitor uses polymorphic method *get_full_name* defined in base *uvm_object* class to query the name of a component being visited. Therefore, every component that is visited can be processed uniformly, regardless of its actual type, as all of them inherit the method from their base class. The situation is a bit more complex if the visitor needs to utilize some of the fields or methods that are not common across all components that are being visited, but are rather specific only for the derived classes. Examples I, II and III presented later in this paper propose the solution to such problem.

```
class name_display_visitor extends uvm_visitor;

        virtual function void visit(uvm_component node);
          `uvm_info("NAME DISPLAY VISITOR",node.get_full_name(),UVM_LOW)
        endfunction

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

Figure 1. Concrete visitor - name display visitor

2. *uvm_visitor_adapter* – abstract class defining a general *accept* operation that in turn applies the corresponding visitor on every element of the structure that the adapter wraps. The definition of a simple adapter that wraps just a single *uvm_component* is shown in Figure 2, whereas the full-working example flow is displayed in Figure 3, with corresponding log output in Figure 4. It can be seen that the context invokes method *accept* of an object of adapter class, providing the component to be visited (in this case, the environment itself, using reference *this*), and also the visitor object as arguments.

```
class basic_adapter extends uvm_visitor_adapter;

        virtual function void accept(uvm_component s, uvm_visitor v, uvm_structure_proxy#(uvm_component) p,  bit invoke_begin_end=1);
                if(invoke_begin_end)
                        v.begin_v();

                v.visit(s);

                if(invoke_begin_end)
                        v.end_v();
        endfunction

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

Figure 2. Basic visitor adapter

```
task visitor_env::run_phase(uvm_phase phase);
  name_display_visitor          name_display_v;
  basic_adapter                 adapter;

  name_display_v     = new("name_display_v");
  adapter            = new("adapter");

  adapter.accept(this, name_display_v, null);
endtask
```

Figure 3. Visitor with basic visitor adapter usage

```
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env
```

Figure 4. Visitor with basic visitor adapter usage - output log

UVM library offers a number of predefined adapters that are used to traverse elements in a complex composite structure in a specific way and apply visitor operation upon each of them: top-down (*uvm_top_down_visitor_adapter*), bottom-up (*uvm_bottom_up_visitor_adapter*), level-by-level *(uvm_by_level_visitor_adapter*). To facilitate the traversal, abstract *uvm_structure_proxy* class used to provide all children subelements of a certain element in a structure is also defined in the UVM library. Its specialization class *uvm_components_proxy* provides all subcomponents for a given UVM component. The traversal is presented in Figure 5 along with the log output in Figure 6. The name of all the components instantiated either directly inside the environment or inside any of its subcomponents is logged recursively.

```
task visitor_env::run_phase(uvm_phase phase);
  name_display_visitor          name_display_v;
  uvm_top_down_visitor_adapter  adapter;
  uvm_component_proxy           proxy;

  name_display_v     = new("name_display_v");
  adapter            = new("adapter");
  proxy              = new("proxy");

  adapter.accept(this, name_display_v, proxy);
endtask
```

Figure 5. Visitor with *uvm_top_down_visitor_adapter* usage

```
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.drv
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.drv.rsp_port
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.drv.seq_item_port
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.mon
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.mon.analysis_port
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.mon.peek_imp
UVM_INFO ../src/visitor_env.sv(7) @ 0: reporter [NAME DISPLAY VISITOR] uvm_test_top.env.master_agent.seqr
...
```

Figure 6. Visitor with *uvm_top_down_visitor_adapter* usage – partial output log

UML diagram

UML [3] is a software modelling language used to represent classes, relations and the flow between them in a graphical form, improving code documentation and understanding. Among software developers, it is a standard notation that, along with design patterns usage, brings many benefits. It provides an efficient way of communicating the intention of developed code, facilitates the ramp-up process of new engineers to the environment, and due to its versitality offers a flexible solution to the problem of class modelling. As verification environments incorporate more and more complex object orienteed constructs, UML might become equally powerful means of expressing the verification environment codebase. The UML class diagram of Visitor design pattern is presented in Figure 7 and UML sequence diagram showing the pattern flow is in Figure 8 [4].

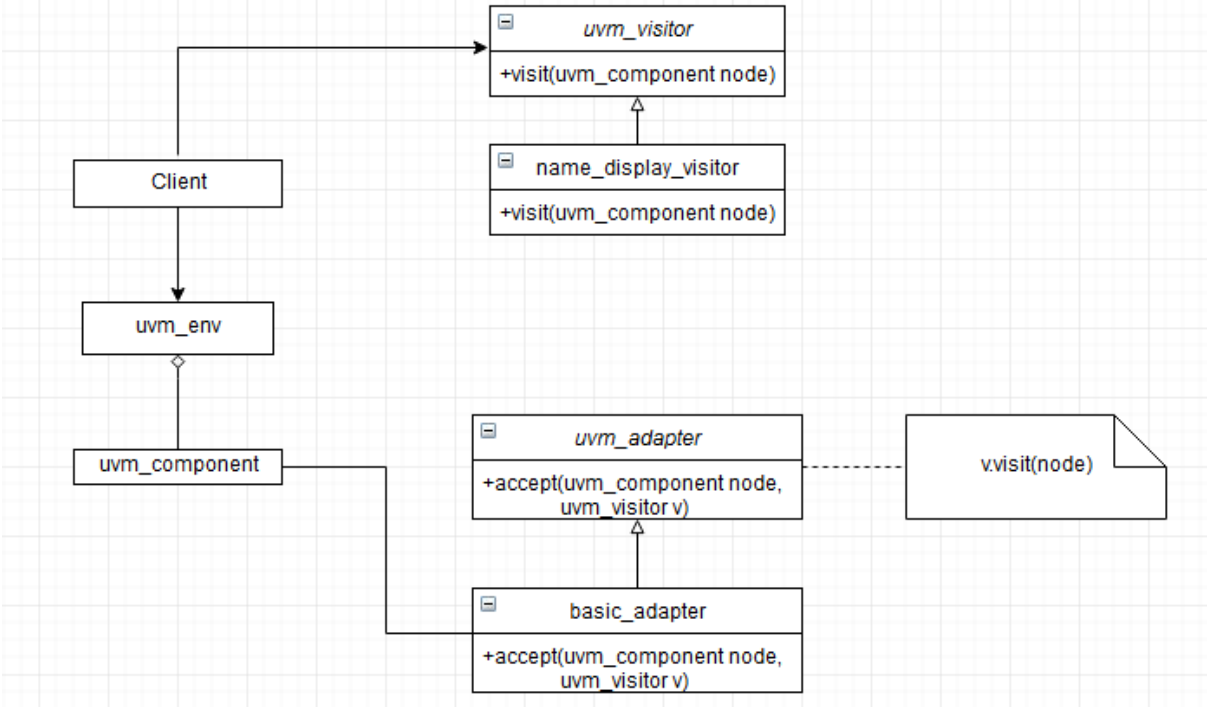

Figure 7. Visitor design pattern - UML class diagram

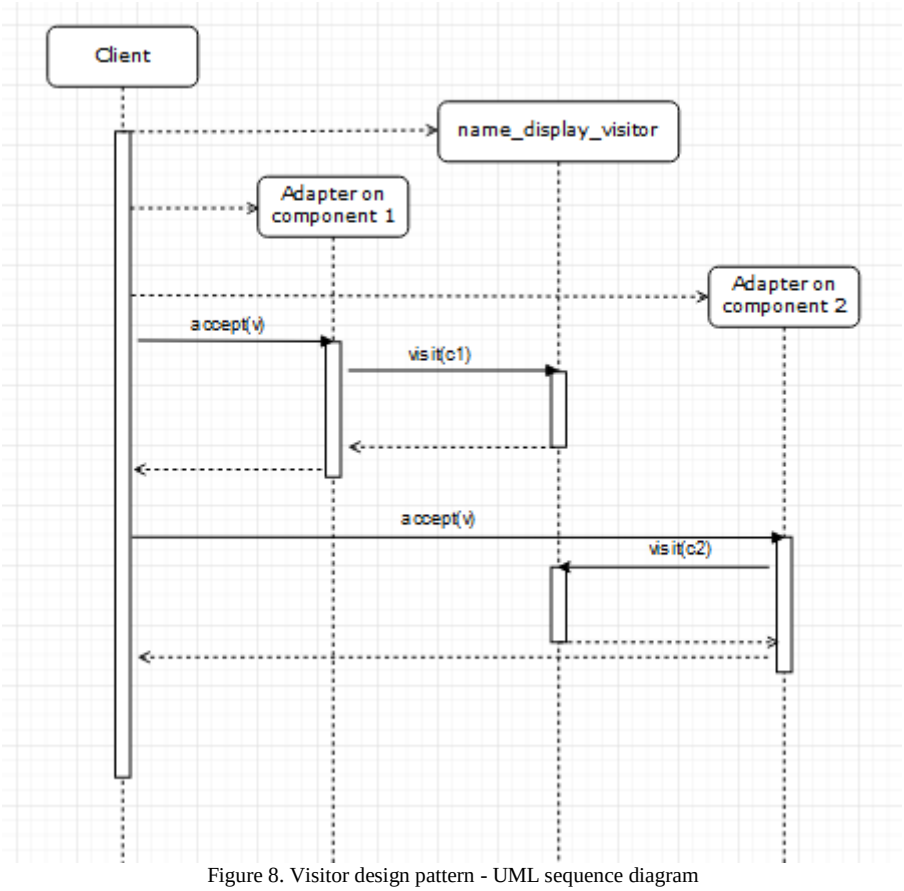

Figure 8. Visitor design pattern - UML sequence diagram

A. Example I: Component configuration check visitor

At the simulation start, after the environment is built, it is recommended to check that every component in the environment has been properly configured with a configuration object and with a pointer to the bus interface it

should drive or observe. Also, over the course of a simulation, many times it is necessary to check whether the content of the configuration object assigned to a component is valid. Fulfilling such requirements, as well as debugging in the case any issues appear, becomes straightforward using the visitor given in Figure 9. The output log in Figure 10 shows that monitor and driver used in the environment are properly configured.

In this example, the visitor needs to access some class members that are not universal for all visited classes, but are rather common just for some derived classes, e.g. configuration field *cfg* belongs to user-defined driver and monitor class. Such requirement is natively supported in some languages mainly used for OOP, such as C++ or Java, through the usage of method overloading[5]. It allows for methods to have the same name, but a different signature with different argument types, leading to very elegant visitor implementations. As such feature is not supported in SystemVerilog, the implementation presented in the example relies on type checking and type casting. As a future enhancement, the author will explore the possibility of improving the implementation by utilizing class parametrization.

B. Example II: Reset check visitor

One problem that is very commonly encountered during the verification development cycle is debugging the rootcause of a simulation getting stuck. As an initial sanity check, it can be helpful to assure that the components in the environment are not under reset and that they are provided with a properly generated clock. At a point when a simulation is stuck, a visitor can be utilized to detect and capture the state of the environment. Figure 11 displays a visitor that observes the values of the reset signal provided to the components of interest. Figure 12 shows the log at different timepoints, clearly indicating when a certain component is put under reset, without having to pollute the codebase of the components with such content.

C. Example III: Adding messages and improving reporting system using Visitor

Visitor can be utilized efficiently to improve reporting system in the environment. The visitor, as an external component can be attached to certain events in the environment and upon their triggering, perform proper reporting. In this example, the visitor is attached to a queue implemented in the scoreboard. The scoreboard stores the data received from one interface monitor and later performs matching with data received from another interface monitor – the case that is very common in verification environments. It is very likely that the queue content and its changes will be of great interest for the developer, especially during the debugging process. The visitor shown in Figure 13 upon each change within the queue displays its every element, with output log in Figure 14. Similarly, additional important content within other classes can be observed and displayed by using the same approach and having related messages about various components localized within one single visitor might be convenient for the developer. It is also worth noting that in this example, the visitor triggers a time consuming method *visit_sb_tcm* as a background task, by using *fork..join_none* SystemVerilog construct.

SUMMARY

The examples show how some tasks in the verification environment can be performed very elegantly using Visitor design pattern. They can be integrated alongside with standard UVM compliant components and utilized to improve overall code quality. Also they can be modelled using UML, improving code documentation. Finally, the paper aims into introducing verification engineers to techniques and concepts commonly utilized within software development world. The author hopes that the engineers who try out the solutions presented in the paper, and get to understand their benefits will also attempt to explore other common software techniques, some of which are presented in [6]. They can as well become a part of their work and improve and facilitate the verification process.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley, 1994.

[2] Accellera, “UVM User Guide, v1.1”, page 131, www.uvmworld.org

[3] http://www.uml.org/

[4] https://www.vainolo.com/2012/07/30/the-visitor-design-pattern-with-sequence-diagrams/

[5] https://www.tutorialspoint.com/method-overloading-in-Java

[6] Darko Tomušilović, Hagai Arbel, “UVM Verification Environment Based on Software Design Patterns”, DVCon U.S. 2018

```
class component_check_visitor extends uvm_visitor;

        virtual function void visit(uvm_component node);
          if (node.get_object_type() == visitor_master_driver::type_id::get()) begin
            visit_driver(node);
          end
          if (node.get_object_type() == visitor_monitor::type_id::get()) begin
            visit_monitor(node);
          end
        endfunction

        virtual function void visit_driver(uvm_component node);
          visitor_master_driver drv;
          $cast(drv, node);

          if (drv.visitor_if == null)
            `uvm_error("COMPONENT CHECK VISITOR", $sformatf("%s: Interface not set", drv.get_full_name()))
          else
            `uvm_info("COMPONENT CHECK VISITOR",$sformatf("%s: Interface set", drv.get_full_name()),UVM_LOW)
          if (drv.cfg == null)
            `uvm_error("COMPONENT CHECK VISITOR",$sformatf("%s: CFG not set", drv.get_full_name()))
          else
            `uvm_info("COMPONENT CHECK VISITOR",$sformatf("%s: CFG set", drv.get_full_name()),UVM_LOW)
        endfunction
        virtual function void visit_monitor(uvm_component node);
          visitor_monitor mon;
          $cast(mon, node);

          if (mon.visitor_if == null)
            `uvm_error("COMPONENT CHECK VISITOR", $sformatf("%s: Interface not set", mon.get_full_name()))
          else
            `uvm_info("COMPONENT CHECK VISITOR",$sformatf("%s: Interface set", mon.get_full_name()),UVM_LOW)
          if (mon.cfg == null)
            `uvm_error("COMPONENT CHECK VISITOR",$sformatf("%s: CFG not set", mon.get_full_name()))
          else
            `uvm_info("COMPONENT CHECK VISITOR",$sformatf("%s: CFG set", mon.get_full_name()),UVM_LOW)
        endfunction

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

Figure 9. Component configuration check visitor

```
UVM_INFO ../src/visitor_env.sv(50) @ 0: reporter [COMPONENT CHECK VISITOR] uvm_test_top.env.master_agent.drv: Interface set
UVM_INFO ../src/visitor_env.sv(54) @ 0: reporter [COMPONENT CHECK VISITOR] uvm_test_top.env.master_agent.drv: CFG set
UVM_INFO ../src/visitor_env.sv(63) @ 0: reporter [COMPONENT CHECK VISITOR] uvm_test_top.env.master_agent.mon: Interface set
UVM_INFO ../src/visitor_env.sv(67) @ 0: reporter [COMPONENT CHECK VISITOR] uvm_test_top.env.master_agent.mon: CFG set
```

Figure 10. Component configuration check visitor - output log

```
class reset_check_visitor extends uvm_visitor;

        virtual function void visit(uvm_component node);
          if (node.get_object_type() == visitor_master_driver::type_id::get()) begin
            visit_driver(node);
          end
          if (node.get_object_type() == visitor_monitor::type_id::get()) begin
            visit_monitor(node);
          end
        endfunction

        virtual function void visit_driver(uvm_component node);
          visitor_master_driver drv;
          $cast(drv, node);

          if (drv.visitor_if.reset_n === 1'b1)
            `uvm_info("RESET CHECK VISITOR",$sformatf("%s: reset deasserted", drv.get_full_name()),UVM_LOW)
          else
            `uvm_info("RESET CHECK VISITOR",$sformatf("%s: reset asserted", drv.get_full_name()),UVM_LOW)
        endfunction

        virtual function void visit_monitor(uvm_component node);
          visitor_monitor mon;
          $cast(mon, node);

          if (mon.visitor_if.reset_n === 1'b1)
            `uvm_info("RESET CHECK VISITOR",$sformatf("%s: reset deasserted", mon.get_full_name()),UVM_LOW)
          else
            `uvm_info("RESET CHECK VISITOR",$sformatf("%s: reset asserted", mon.get_full_name()),UVM_LOW)
        endfunction

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

Figure 11. Reset check visitor

```
UVM_INFO ../src/visitor_env.sv(94) @ 10000: reporter [RESET CHECK VISITOR] uvm_test_top.env.master_agent.drv: reset asserted
UVM_INFO ../src/visitor_env.sv(104) @ 10000: reporter [RESET CHECK VISITOR] uvm_test_top.env.master_agent.mon: reset asserted
UVM_INFO ../src/visitor_env.sv(92) @ 110000: reporter [RESET CHECK VISITOR] uvm_test_top.env.master_agent.drv: reset deasserted
UVM_INFO ../src/visitor_env.sv(102) @ 110000: reporter [RESET CHECK VISITOR] uvm_test_top.env.master_agent.mon: reset deasserted
```

Figure 12. Reset check visitor - output log

```
class queue_display_visitor extends uvm_visitor;

        virtual function void visit(uvm_component node);
          if (node.get_object_type() == visitor_sb::type_id::get()) begin
            fork
              visit_sb_tcm(node);
            join_none
          end
        endfunction

        virtual task visit_sb_tcm(uvm_component node);
          visitor_sb sb;
          $cast(sb, node);

          `uvm_info("QUEUE DISPLAY VISITOR",$sformatf("Start monitoring scoreboard queue"),UVM_LOW)

          forever begin
            @(sb.data_q.size());
            `uvm_info("QUEUE DISPLAY VISITOR",$sformatf("Scoreboard queue size changed. New size: %d", sb.data_q.size()),UVM_LOW)
            `uvm_info("QUEUE DISPLAY VISITOR",$sformatf("Scoreboard queue content: %p", sb.data_q),UVM_LOW)
          end
        endtask

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

Figure 13. Queue display visitor

```
UVM_INFO ../src/visitor_env.sv(130) @ 10000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           1
UVM_INFO ../src/visitor_env.sv(131) @ 10000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h5a}
UVM_INFO ../src/visitor_env.sv(130) @ 12000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           2
UVM_INFO ../src/visitor_env.sv(131) @ 12000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h5a, 'h6b}
UVM_INFO ../src/visitor_env.sv(130) @ 15000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           3
UVM_INFO ../src/visitor_env.sv(131) @ 15000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h5a, 'h6b, 'h7c}
UVM_INFO ../src/visitor_env.sv(130) @ 18000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           2
UVM_INFO ../src/visitor_env.sv(131) @ 18000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h6b, 'h7c}
UVM_INFO ../src/visitor_env.sv(130) @ 21000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           3
UVM_INFO ../src/visitor_env.sv(131) @ 21000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h6b, 'h7c, 'h8d}
UVM_INFO ../src/visitor_env.sv(130) @ 22000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           2
UVM_INFO ../src/visitor_env.sv(131) @ 22000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h7c, 'h8d}
UVM_INFO ../src/visitor_env.sv(130) @ 23000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           1
UVM_INFO ../src/visitor_env.sv(131) @ 23000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h8d}
UVM_INFO ../src/visitor_env.sv(130) @ 25000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           2
UVM_INFO ../src/visitor_env.sv(131) @ 25000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h8d, 'h9e}
UVM_INFO ../src/visitor_env.sv(130) @ 26000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           1
UVM_INFO ../src/visitor_env.sv(131) @ 26000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{'h9e}
UVM_INFO ../src/visitor_env.sv(130) @ 27000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue size changed. New size:           0
UVM_INFO ../src/visitor_env.sv(131) @ 27000000: reporter [QUEUE DISPLAY VISITOR] Scoreboard queue content: '{}
```

Figure 14. Queue display visitor - output log