AN OVERVIEW OF TRANSACTION-LEVEL MODELING (TLM) IN UNIVERSAL VERIFICATION METHODOLOGY (UVM)

[1] Ashwin P. Patel 2 Vyom M. Bhankhariya [3] Jignesh S. Prajapati

[1 2 3] P.G Student, Department of Electronics Engineering,
Gujarat Technological University
Gandhinagar, Gujarat, India

[1]ashwinpatel14588@gmail.com
[2]vyom.14289@gmail.com
[3]uranusharshal@rediffmail.com

ABSTRACT: *Transaction –level modeling is a methodology for building models at high level of abstraction. UVM components (Universal Verification Methodology) communicate via standard TLM interface, which improves reuse. Using a System Verilog implementation of TLM in UVM, a component may communicate via its interface to any other component that implements that interface. Each TLM interface consists of one or more methods used to transport data. The UVM TLM library defines several abstract, transaction-level interfaces & ports & exports that facilitate their use.* O*ne component may be connected at the transaction level to others that are implemented at multiple levels of abstraction. The common behavior of TLM communication permits components to be swapped in and out without affecting the rest of the environment.*

KEYWORDS: TLM, transaction, transaction-level communication, peer-to-peer connection, hierarchy connection, generic payload, analysis port and export, socket.

1. INTRODUCTION

For the verification engineer to verification productivity one of the keys to think about the problem at a level of abstraction. When verifying a DUT, one must create a verification environment that supports the abstraction level. It is necessary to manage verification tasks, such as generating stimulus and collecting coverage information at the transaction level.

UVM provides a set of transaction-level communication interfaces and channels that you can use to connect components at the transaction level. TLM interface isolates the verification component from other component throughout the verification environment.

The well-defined semantics of TLM interfaces between components also provide the ideal platform for implementing mixed-language verification environments. In addition, TLM provides the basis for easily encapsulating components into reusable components, called *verification components*, to maximize reuse and minimize the time and effort required to build a verification environment.

2. TRANSFER ISSUES

In a simple port based data transfer Components like producer and consumer are implemented as modules. These modules are connected using module ports or SV interfaces. The advantage of this methodology is, the two above mentioned components are independent. Instead of consumer module, any other component which can understand producer interface can be connected, which gives a great reusability. The disadvantage of this methodology is, data transfer is done at lower lever abstraction.

In Task based data transfer, methods are used to transfer the data between components. So, this gives a better control and data transfer is done at high level. The disadvantage of this method is that components are using hierarchal paths which do not allow the reusability.

To overcome from this above two disadvantages we will use TLM methodology in UVM. Data is transferred at high level. Transactions which are developed by extending the uvm_sequence_item can be transferred between components using method calls. These methods are not hierarchal fixed, so that components can be reused..

Advantage of TLM interface

1. Connect to system C
2. High level abstraction
3. Reusability
4. Less code
5. Simple implementation

6. Low simulation time

3. TLM

First we will understand what is transaction in TLM. Transaction is a class object which is extended from uvm_transaction. It includes the information that are needed to build unit of communication between components.

transaction

```
class trans extends uvm_transaction;
rand data_t data;
rand addr_t addr;
constraint c1 { addr < 16'h1000; }
...
Endclass
```

The transaction object includes variables, constraints, and other fields and methods necessary for generating and operating on the transaction. The transaction could also be extended to include additional constraints. Transactions can thus be composed, decomposed, extended, layered, at any level of abstraction.

Transaction-Level Communication

Transaction-level interfaces define a set of methods that use transaction objects as arguments. A TLM port defines the set of methods (the application programming interface (API)) to be used for a particular connection, while a TLM export supplies the implementation of those methods. Connecting a port to an export allows the implementation to be executed when the port method is called.

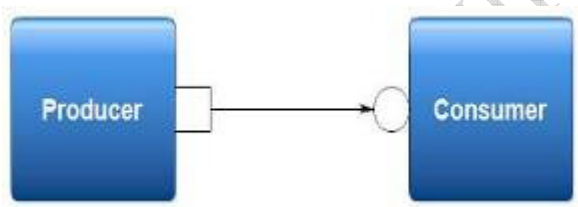

Fig 1-basic TLM communication

The square box on the producer indicates a port and the circle on the consumer indicates the export. The producer generates transactions and sends them out its put_port:

```
class producer extends uvm_component;
uvm_blocking_put_port #(trans) put_port;

function new( string name, uvm_component parent);
put_port = new("put_port", this);
...
endfunction
virtual task run();
simple_trans t;
for(int i = 0; i < N; i++) begin
// Generate t.
put_port.put(t);
end
endtask
```

The actual implementation of the put() call is supplied by the consumer.

```
class consumer extends uvm_component;
uvm_blocking_put_imp #(trans, consumer) put_export;
...
task put(trans t);
case(t.kind)
READ: // Do read.
WRITE: // Do write.
endcase
endtask
endclass
```

The semantics of the put operation are defined by TLM. In this case, the put() call in the producer will block until the consumer's put implementation is complete. Other than that, the operation of producer is completely independent of the put implementation (uvm_put_imp). In fact, consumer could be replaced by another component that also implements put and producer will continue to work in exactly the same way.

The converse operation to put is get.

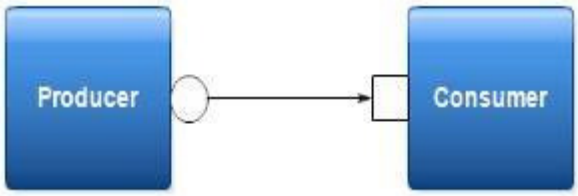

Fig 2- Consumer gets from producer

```
class consumer extends uvm_component;
uvm_blocking_get_port #(trans) get_port;
function new( string name, uvm_component parent);
get_port = new("get_port", this);
...
endfunction
virtual task run();
simple_trans t;
for(int i = 0; i < N; i++) begin
// Generate t.
get_port.get(t);
end
endtask
```

The get() implementation is supplied by the producer.

```
class producer extends uvm_component;
uvm_blocking_get_imp #(simple_trans, producer)
get_export;
...
task get(output simple_trans t);
simple_trans tmp = new();
// Assign values to tmp.
t = tmp;
endtask
endclass
```

As with put() above, the get_consumer's get() call will block until the get_producer's method completes. In TLM terms, put() and get() are blocking methods.

Communicating between Processes

In above example, the consumer will be active only when its put() method is called. Some time in many cases, it may be necessary for components to operate independently, where the producer is creating transactions in one process while the consumer needs to operate on those transactions in another.

UVM provides the uvm_tlm_fifo channel to facilitate such communication. The uvm_tlm_fifo implements all of the TLM interface methods, so the producer puts the transaction into the uvm_tlm_fifo, while the consumer independently gets the transaction from the fifo, as shown in figure.

Fig 3- Using a UVM tlm fifo

When the producer puts a transaction into the fifo, it will block if the fifo is full, otherwise it will put the object into the fifo and return immediately. The get operation will return immediately if a transaction is available (and will then be removed from the fifo), otherwise it will block until a transaction is available.

Connecting Transaction-Level

In verification environment the transaction-level component define ports and exports, the actual connection between them is accomplished via the connect() method, with an argument that is the object (port or export) to which it will be connected. In a verification environment, the series of connect() calls between ports and exports establishes a netlist of peer-to-peer and hierarchical

Peer-to-Peer connections

In verification environment at the same level of hierarchy, ports are always connected to exports.

```
class my_env extends uvm_env;
...
virtual function void connect();
component.port.connect(target.export);
producer.blocking_put_port.connect(fifo.put_export);
get_consumer.get_port.connect(fifo.get_export);
...
endfunction
endclass
```

All TLM connection are checked for compatibility before test runs. In order for a connection to be valid, the export must provide implementations for at least the set of methods defined by the port and the transaction type parameter for the two must be identical.

Hierarchical Connections

Making connections across hierarchical boundaries involves some additional issues, which are discussed here.

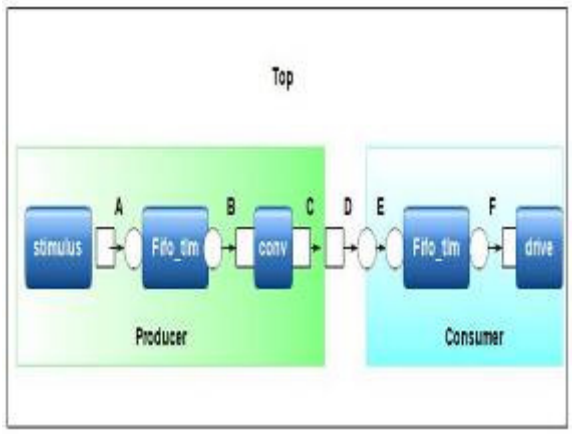

Fig 4- Hierarchical connection

The hierarchy of this design contains two components, producer and consumer. producer contains three components, stimulus, fifo_tlm, and cov. Consumer contains two components, fifo_tlm and drive. As shown in the figure the producer's put_port is connected to the consumer's put_export. The two fifos are both unique instances of the same uvm_tlm_fifo component. Connections A, B, D, and F are standard peer-to-peer connections as we discussed above.

Connections C and E are of a different than what have been shown. Connection C is a port-to-port connection, and connection E is an export-to-export connection. These two kinds of connections are necessary to complete hierarchical connections. Connection C imports a port from the outer component to the inner component. Connection E exports an export upwards in the hierarchy from the inner component to the outer one. Ultimately, every transaction-level connection must resolve so that a port is connected to an export. We use port-to-port and export-to-export connections to bring connectors to a hierarchical boundary to be accessed at the next-higher level of hierarchy.

All export-to-export connections in a parent component are of the form

```
export.connect(subcomponent.export)
```

so connection E would be coded as:

```
class consumer extends uvm_component;
uvm_put_export #(trans) put_export;
uvm_tlm_fifo #(trans) fifo;
...
function void connect();
put_export.connect(fifo.put_export); // E connection export
to export
bfm.get_port.connect(fifo.get_export); // F
endfunction
...
endclass
```

Conversely, port-to-port connections are of the form:

```
subcomponent.port.connect(port);
```

so connection C would be coded as:

```
class producer extends uvm_component;
uvm_put_port #(trans) put_port;
conv c;
...
function void connect();
c.put_port.connect(put_port);
...
Endfunction
```

Connection Types

Connection types	Connect() method
Port-to-export	Comp1.port.connect(comp2.export)
Port-to-port	Subcomponent.port.connect(port)
Export-to-export	Export.connect(subcomponent.export)

Table 1- Connection types

For the connection like peer-to-peer & hierarchical connection the argument to the port.connect() method is export and port respectively. For export.connect() method argument is always export.

Analysis Communication

Each component is responsible for communicating through its TLM interface(s) with other components in the system in order to stimulate activity in the DUT and/or respond its behavior. In any reasonably complex verification environment, however, particularly where randomization is applied, a collected transaction should be distributed to the rest of the environment for end-to-end checking (scoreboard), or additional coverage collection. For analysis, the emphasis is on a particular component, such as a monitor, being able to produce a stream of transactions, regardless of whether there is a target actually connected to it. Modular analysis components are then connected to the analysis_port, each of which processes the transaction stream in a particular way.

Analysis Ports

The uvm_analysis_port (represented as a diamond on the monitor in Figure 9) is a specialized TLM port whose interface consists of a single function, write(). The analysis port contains a list of analysis_exports that are connected to it. When the component calls analysis_port.write(), the analysis_port cycles through the list and calls the write() method of each connected export. If nothing is connected, the write() call simply returns. Thus, an analysis port may be connected to zero, one, or many analysis exports.

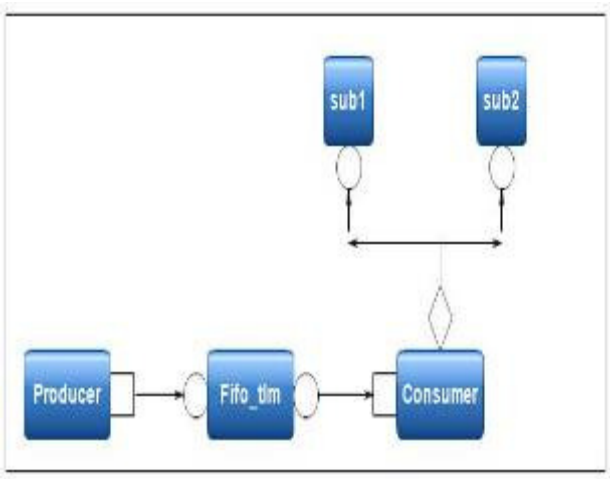

Fig 5- Analysis Port

```
class consumer extends get_consumer;
uvm_analysis_port #(trans) anls_port;
function new(...);
super.new()
anls_port = new("analysis_port", this);
...
endfunction
task run;
...
for(int i=0; i<10; i++)
if(get_port.try_get(t)) begin
//Do something with t.
anls_port.write(t); // Write transaction.
...
end
endtask
```

In the parent environment, the analysis port gets connected to the analysis export of the desired components,
such as coverage collectors and scoreboards.

Analysis Exports

As with other TLM connections, it is up to each component connected to an analysis port to provide an implementation of write() via an analysis_export. The uvm_subscriber base component can be used to simplify this operation, so a typical analysis component would extend uvm_subscriber as:

As with put() and get() described above, the TLM connection between an analysis port and export, allows the export to supply the implementation of write(). If multiple exports are connected to an analysis port, the port will call the write() of each export, in order. Since all implementations of write() must be functions, the analysis port's write() function completes immediately, regardless of how many exports are connected to it.

```
class env extends uvm_env;
consumer con;
sub1 s1;
sub2 s2;
...
function void connect();
con. anls_port.connect(s1.analysis_export);
con. anls_port.connect(s2.analysis_export);
```

```
...
endfunction
endclass
```

Each write() implementation must make a local copy of the transaction and then operate on the copy to avoid corrupting the transaction contents for any other subscriber that may have received the same. UVM also includes an analysis_fifo, which is a uvm_tlm_fifo that also includes an analysis export, to allow blocking components access to the analysis transaction stream.

Generic Payload

TLM-2.0 defines a base object, called the generic payload, for moving data between components. It is the default transaction type in System Verilog. Each attribute in the SystemC version has a corresponding member in the SystemVerilog generic payload. All of the members of the generic payload have the rand qualifier. This enables instances of the generic payload to be randomized. SystemVerilog allows arrays, including dynamic arrays to be randomized. In SystemC, all of the attributes are private and are accessed through accessor methods. In SystemVerilog, all members are protected and similarly accessed through accessor methods.

```
virtual function bit is_read();
virtual function void set_read();
```

The accessor functions let you set and get each of the members of the generic payload. All of the accessor methods are virtual.

Sockets

In TLM-1 connection between two component is done through ports and exports, where as in TLM-2.0 it is done through sockets. A socket is derived from base class uvm_port_base. Unlike port and export provides both forward and backward path. Now we can enable the communication in bi-direction way by connecting sockets together. Sockets contains both a port and an export.

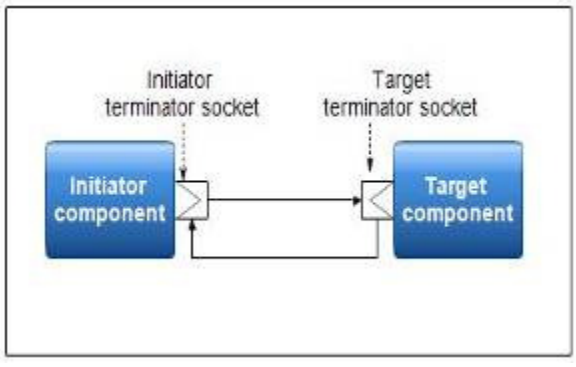

Fig.6- Socket connection

Components that initiate transactions are called initiators and components that receive transactions sent by an initiator are called targets. Initiators have initiator sockets and targets have target sockets. Initiator sockets can only connect to target sockets; target sockets can only connect to initiator sockets.

Figure shows the diagramming of socket connections. The socket symbol is a box with a triangle with its point indicating the data and control flow direction of the forward path. The backward path is indicated by an arrow connecting the target socket back to the initiator socket. Sockets come in several flavors: Each socket is an initiator or a target, a passthrough, or a terminator. Furthermore, any particular socket implements either blocking interfaces or nonblocking interfaces.

4. CONCLUSION

It can be conclude that it is not easy to connect different verification component in environment at high level abstraction. With TLM interface data is transferd at high level .transaction can be transfered between components using call method. The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate efficient verification.

5. REFERENCES

[1] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, Byeong Min ," Beyond UVM for practical Soc Verification" , year 2011,pp 158-162.

[2]http://www.accellera.org/downloads/standards /uvm/uvm_users_guide_1.1.pdf

[3]http://www.accellera.org/downloads/standards /uvm/ uvm_ref_guide_1.0_ea.pdf

[4] A Prectical Guide to Adopting UVM.