The Universal Translator –
A Fundamental UVM Component for Networking Protocols

David Cornfield
Sr. Principal Verification Engineer
AppliedMicro, Inc.
Kanata, Ontario, Canada
dcornfield@apm.com

Abstract – Network Protocol stacks construct a reliable communications channel between two entities by decomposing the problem such that higher levels of abstraction rely on lower levels of abstraction for a service.

The UVM toolkit does not adequately provide for modeling protocol stacks: The UVM Agent Architecture cannot be stacked, and the Layered Sequencing approach presented by Fitzpatrick has a variety of issues when applied to complex protocol stacks.

This paper presents a translator class for modeling protocol stacks and its associated concepts: semantic independence, the Translation API; Inline Sequencing; dynamic and adaptive translation using Orthogonal Sequencing; and the Layered Architecture.

Keywords – adapter, converter, translator, adaptation, conversion, translation, transform, transformation, stack, layer, cascade, protocol, networking, UVM.

I. INTRODUCTION

The UVM Agent Architecture is very well suited to System-on-a-Chip type devices, as the Agent is easily ported from the unit level to the chip level. Porting is straightforward because the I/O connected to the virtual interface of the Agent is visible in both scopes:

The Sequence based Layering Architecture advanced by Fitzpatrick (1) addresses this critical flaw by pairing a translator sequence sourcing from a child sequencer in the stimulus path with a uvm_subscriber based reconstruction monitor in the analysis path:

![Figure 3: Sequence Based Layering](image)

This promising architecture resolves the cascaded Agent problem, but has a few architectural quirks and suffers from a variety of issues.

A. Asymmetry

The most glaring idiosyncrasy is the asymmetry: the analysis path is composed of a uvm_component that is connected via ports while the stimulus path is composed of a persistent uvm_object that is connected via a convoluted process of pointer passing. Quite simply: the analysis path follows the UVM component/port orthodoxy while the stimulus path does not.

Though it is minor, this asymmetry also introduces an inconsistency UVM messaging, as the translator sequence derives from uvm_object and the object must use either the test or a sequencer component as a messaging proxy.

B. Peripheral Clutter

Furthermore, it should be noted that in the case of the stimulus path, all of the functionally relevant translation code resides within the translator sequence. The intervening sequencers serve no real purpose other than to provide messaging proxies and connection pointers. Hence these sequencers just clutter the Architecture, adding little practical value.

C. Packaging Ambiguity

A third issue that stems from this Architecture distills down to an ambiguity in packaging. On one hand the translator sequence should be packaged with the lower level sequencer because that is the sequencer on which it is started and sequences are conventionally packaged with their associated sequencers. On the other hand, the translator sequence contains all of the relevant translation functionally and so should be packaged with the subscriber and child sequencer. This is not the sequencer on
which it is started and thus makes for an unconventional packaging. A packaging issue may seem trivial, but ambiguities in architectures never scale and distribute well.

A slight variation on the packaging problem is that layers require intimate knowledge of the inbound and outbound transactions, in which case a layer must import the transactions of its peers. This tethers a layer to its peers and makes it far less portable to alternate, unanticipated contexts.

D. Semantic Dependency

A far more subtle issue with serious consequences is the semantic dependency of the architectural pieces. The stimulus path translates using the pull semantic while the analysis path translates with a push semantic. The specific translations are bound to the underlying semantics of the paths in which they reside. What happens if we need to translate with the opposite semantic? With the current Architecture we are left to rewrite and maintain the same translation in two formats. For large, complex, standards based translations, this is a serious problem.

As evidence of the need to run one translation with both semantics, consider the case of the IEEE 802.3 10GBASE-R PCS 66b/64b encoder. At the chip level, needs dictate that MAC packets are driven and recovered from the chip using a PCS encoder/decoder pair:

![Diagram of VIP and Chip level needs]

In this context the encoder VIP operates under a pull semantic in the stimulus path while the decoder VIP operates under a push semantic.

At the unit level, however, needs dictate the opposite semantic for the encoder VIP. Though it may seem reasonable to use the PCS decoder VIP to verify the PCS encoder device under test, this in fact does not work:

![Diagram of unit level needs - incorrect]

This does not work because the PCS encoder implements a one-way function — that is: information that is sent into the encoder cannot always be recovered from its output. It is akin to recovering the input to the modulo function by looking at the output: it simply can’t be done because information is lost in the process. What’s required instead is the following hook up:

![Diagram of unit level needs - correct]

The difference here is that now the PCS encoder VIP is in the analysis path and hence has the push semantic. Thus at the chip level we need a pull encoder but at the unit level we need a push encoder. If one-way functions are not identified and mitigated ahead of time, you may find yourself with a translation with the wrong semantic at a time when you can’t afford to be without it.

In conclusion, the analysis path of the current UVM Layered Architecture has an elegance consistent with the UVM component/port orthodoxy that is completely lacking in the stimulus path, and this introduces a variety of problems. If only there were a uvm_component like the uvm_subscriber based reconstruction monitor in the stimulus path, the Layered Architecture would be considerably more symmetric:

![Component Based Layering Diagram]

And if that uvm_component could somehow abstract out push/pull semantics, the same translation could be used in either the stimulus path or the analysis path (or both) as needed...

II. THE TRANSLATOR CLASS

A Translator is a uvm_component that translates a stream of inbound items into a stream of outbound items.

This abstraction is codified in the virtual translator class, where the inbound item type and outbound item type are parameters. The specifics of the translation are left to the pure virtual translate task, which must be implemented in derivatives:

```
virtual class translator #(
    type t_inbound_item = uvm_sequence_item,
    type t_outbound_item = uvm_sequence_item,
) extends uvm_component;

pure virtual task translate();

endclass
```

The translate task is semantically independent, meaning that the inbound items are translated into outbound items by the same task regardless of whether transactions are pushed or pulled through the translator class. The benefit to this is that the translation can be used in either the stimulus path or the analysis path, depending on verification needs.

The push or pull semantic is dictated by the is_active bit of the translator class. When the is_active bit is UVM_ACTIVE the Translator is intended to operate in the stimulus path with a pull semantic. Outbound items are pulled out of the class via the seq_item_export, which will in turn trigger requests for inbound items from its seq_item_port by means of the translate task.

![UVM_ACTIVE Translation Diagram]
When the is_active bit is UVM_PASSIVE the Translator is intended to operate in the analysis path with a push semantic. Inbound items are pushed into the class via the analysis_export, which in turn results in outbound items being pushed out its analysis_port after translation.

![Diagram of UVM_PASSIVE Translation](image)

**Figure 9: UVM_PASSIVE Translation**

Note that the names and semantics of ports on the translator class change depending on the setting of the is_active bit, but that the translation is always from inbound items to outbound items. (Incidentally the setting of the is_active bit is not a random act, but rather a deliberate act that depends on the usage context, so the change in port names and semantics is only really a consideration during environment construction, not during test execution).

### A. The Translation API

A specific translation from one type of item to another type of item is codified in an extension to the translator class where the inbound and outbound item type parameters are specified and the translate task is implemented using the Translation API.

The **Translation API** consists of the following four tasks, of which only two are called in any implementation of the translate task:

```plaintext
get_inbound_item (  
    output t_inbound_item item  
);

try_inbound_item (  
    output t_inbound_item item  
);

put_outbound_item (  
    input t_outbound_item item  
);

put_uncloned_outbound_item (  
    input t_outbound_item item  
);
```

To use the Translation API, the translate task is overridden to:

1. get inbound items using the get_inbound_item or try_inbound_item task calls;
2. transform one or more inbound items into one or more outbound items, as required by the application; and then
3. send out the outbound items using calls to either put_uncloned_outbound_item or put_outbound_item.

The translate task thus always follows a get-transform-put or try-transform-put pattern.

The get_inbound_item task blocks if an item is not available on the inbound port, while the try_inbound_item is non-blocking and will return a null if an item is not available. These semantics hold regardless of the is_active setting. The put_outbound_item task will clone the item before sending it out the outbound port, while a call to the put_uncloned_outbound_item will not.

For a given translation, there are no requirements on periodicity and no limitations on the quantity of outbound items produced from a quantity of inbound items. A translation can be one-to-one, one-to-many, many-to-one or any flavor of many-to-many, and be either periodic or aperiodic, as long as the translate task follows the get/try-transform-put pattern. Quite simply the only requirement is that you must always get enough input before putting out output.

If the translation is periodic, the translate task of derivatives need only implement one cycle, as the task invoked on an as needed basis – i.e. when inbound items arrive in a push scenario or when outbound items are requested in a pull scenario.

### B. Inline Sequencing

Given that outbound items are an output of the translate task, they are no longer directly controllable – that is you can only produce a sequence of output items by sending in a sequence of input items. Thus scenarios could arise where there is no possible input sequence that can be provided that will produce the desired output sequence. Though this is true in general, it is generally only an **issue** for stimulus generation.

To address this situation, the translator class has an is_sequence bit that when set instantiates and enables the **inline sequencer**, a t_outbound_item typed uvm_sequencer named inline_sqr.

![Diagram of Inline Sequencer](image)

**Figure 10: The Inline Sequencer**

Enabling the inline sequencer provides direct control over the outbound item stream. Doing so does not change the ports of the translator class; it only changes from where the outbound items are internally sourced – from the inline sequencer when enabled or from the translate task when disabled. The fact that the ports do not change implies that inline sequencing can be done in situ on an as needed basis without changing peer connections.

The inline sequencer is not instantiated if the is_sequence bit is not set.

### C. Debug Hooks

In terms of debug, the translator class will: log inbound items to a file if the inbound_log property is configured to a file name; write inbound items out the inbound_tap analysis port if the has_inbound_tap bit is set; log outbound items to a file if the outbound_log property is configured to a file name; and/or write outbound items out the outbound_tap analysis port if the has_outbound_tap bit is set.

![Diagram of Debug Hooks](image)

**Figure 11: Debug Hooks**
The translator class also issues GET, TRY and PUT info reports if the verbosity is UVM_HIGH or higher.

Log entries and reports use the convert2string method of the inbound and outbound items.

Tap ports are not instantiated unless the associated has_*bound_tap bit has been set.

D. Configuration

The is_active, issequenced, hasinbound_tap, and hasoutbound_tap configuration bits, and the inbound_log and outbound_log configuration strings are public and can be set directly or by using the configuration database with like-named identifier strings (i.e. set issequenced with the “issequenced” identifier string). The bits must be set before the build phase while the strings must be set before the run phase.

E. Performance

The translator class is optimized for performance: no objects within the class are instantiated unless required by the application and/or specifically configured to be present; all reports check verbosity levels before constructing and reporting the message; and background threads are kick-started only when required.

Furthermore, Wilcox & D’Onofrio (2) demonstrated that sequencers are outperformed by sequences as the instance count grows, so the inline sequencer was designed as an optionally instantiated component rather than inherited functionality to mitigate this effect.

F. Limitations

A Translator is unidirectional. It cannot translate inbound items to outbound items in one direction and translate outbound responses back into inbound responses in the reverse direction. This is due to the arbitrary relationship between inbound items and outbound items, which is not necessarily one-to-one, as required by the request/response semantic.

A Translator also cannot use the try_inbound_item API with a push semantic (is_active is UVM_PASSIVE), as this would initiate an infinite zero-time loop. The translator class will issue a fatal in this context.

Inline sequencing is only available when is_active is set to UVM_ACTIVE, despite the controllability issue existing when the bit is set to UVM_PASSIVE.

G. Examples

As an example, the following class is the semantically independent 10GBASE-R PCS 64-bit/66-bit encoder needed for both unit and chip level testing:

class pcs_encoder extends translator
#(t_mii_transfer, t_block);

task translate();
  t_mii_transfer t1,t2;
  t_block block;

  // (1) Get inbound items:
  get_inbound_item(t1);
  get_inbound_item(t2);

  // (2) Transform the inbound items into outbound items:
  block = encode(t1,t2);

  // (3) Send out the outbound item:
  put_outbound_item(block);
endtask

function t_block encode (input t_mii_transfer t1, input t_mii_transfer t2);
  // Convert two MII transfers into a block according to IEEE 802.3 Clause 49.
  endfunction

endclass

Notice the get-transform-put pattern. Also note that the periodicity of the translation is well defined by the IEEE – i.e. 2 inbound items (MII transfers) are translated into 1 outbound item (a BLOCK) – so only one cycle of translation is implemented.

The following example is particularly interesting because the periodicity of the translation is unknown and is a function of the class parameters, which could be set to something quite aperiodic.

class gearbox #(BWI, BWO) extends translator
#(t_bitstream_item#(BWI), t_bitstream_item#(BWO));

  t_bitstream_item#(BWO) ob; int j=0;

task translate();
  t_bitstream_item#(BWI) ib;

  // (1) Get
  get_inbound_item(ib);
  for (int i=0;i<BWI;i++) begin
    // (2) Transform
    ob.data[j++] = ib.data[i];
    if (j == BWO) begin
      // (3) Send
      put_outbound_item(ob);
      j = 0;
    end
  end
endtask

endclass

Again notice the get-transform-put pattern.

In this example, when BWI < BWO, several inbound items are converted into one outbound item and the gearbox class implements a deserialization function. When BWI > BWO, a single inbound item is converted into several outbound items and the gearbox class instead implements a serialization function.

Note the following subtlety when operating as a deserializer: the put_outbound_item is not called with each invocation of the translate task and so the outbound item itself must span multiple invocations. This is perfectly legal because the get-transform-put pattern is followed despite spanning multiple invocations of translate. This highlights that and outbound item should only go out when enough has come in.

The following example is an implementation of the G.709 OTU frame synchronous scrambler. The scrambler also happens to be the descrambler. Though it is a trivial translation, it demonstrates that the same Translator can be used in both the stimulus (scrambler) and analysis (descrambler) paths if the application requires it.

class otu_scrambler extends translator
// Favor memory over compute time, so compute once and save the result

bit [0:130559] mask;

function new();
bit [1:16] lfsr;
lfsr = 16’hffff;
for (int i=49;i<130560;i++) begin
    mask[i] = lfsr[16];
end
endfunction

function void put_mii (bit [3:0] ctrl, bit [31:0] data ) {
    t_mii_transfer mii;
mii.ctrl = ctrl;
mii.data = data;
put_outbound_item(mii);
endfunction

endclass

Note that put_outbound_item was not directly called from within the translate task but that the translation still has the try-transform-put pattern.

III. ORTHOGONAL SEQUENCING

A Translator generally produces legal outbound items from legal inbound items. Typically, however, the stimulus path must drive illegal items to test the error recovery of the device under test. Inbound item types can be overloaded with control knobs to produce illegal outbound items, but this practice introduces control knob pollution and control knob explosion when translators are cascaded:

Figure 12: Control Knob Pollution

Notice that to access the control knobs, Z’, of Translator A, Translator B must pass the control knobs for A through its translation. This is control knob pollution, as the X item now contains control knobs completely unrelated to either X or Y items. Now consider what happens when more and more translators are chained together and/or used in multiple contexts – the result is control knob explosion:

Figure 13: Control Knob Explosion

It should be evident from the input of the shaded translator, which is used in two relatively simple contexts, that overloading inbound items with control knobs is not a sustainable practice.

The solution to this problem is to use orthogonal sequencing. **Orthogonal sequencing** means to separate the control knobs from the data and to source the control from an alternate “orthogonal” sequence item port. The concept carries the term “orthogonal” because of the way this sequencing is depicted:
Orthogonal sequencing was introduced as a means to produce illegal outbound items, but there is no need to limit the concept to error insertion. Orthogonal sequencing can be used for dynamic translation, which simply means to modulate the translation of legal items over time. Note that the modulation can be either synchronous or asynchronous to the translation.

Encapsulation is an example of a general class of dynamic translation, where time varying overhead is added to a payload using orthogonal sequencing:

Dynamic translation can further blossom into the advanced area of adaptive translation by using the orthogonal response channel to feed information from the Translator back into the orthogonal sequence:

An example application of this might be to request inter-packet gap between MAC packets via an orthogonal sequence and feedback the actual inter-packet gap inserted in order to model the oscillating effect of back-pressure loop time.

Package Isolation

Orthogonal sequencing is a powerful complement to the Translator. It can be used inject errors, and/or modulate and/or adapt a translation. And it can be synchronous or asynchronous to the flow of translated items.

But it can also help with a packaging problem alluded to in the introduction – and that is the tethering of a layer to its peers via imported transaction types. If absolutely everything other than data is pushed into orthogonal sequences, then layer interfaces distill down to a very small set of simple and interchangeable data ports.

In fact at AppliedMicro our suite of VIP all interoperate on one of four fundamental data classes, each with a parameterized property called data:
- a packet (bit [7:0] data[]);
- a frame (bit [0:FL-1][7:0] data);
- a bitstream (bit [DW-1:0] data); and
- a bundle (bit [0:LC-1][DW-1:0] data) which used for lane based protocols like CAUI or Interlaken and is simply an array of bitstreams, one bitstream per lane.

IV. THE LAYERED ARCHITECTURE

Given that we have covered Translators and the concept of orthogonal sequencing, we now are ready to construct a formal, component based Layered Architecture.

A. Layers

A Layer (L) is defined as a uvm_component with an is_active bit that: translates inbound items arriving in the high abstraction interface into outbound items departing the low abstraction interface; and translates inbound items arriving in the low abstraction interface into outbound items departing the high abstraction interface.

The device under test generally resides closer to the low abstraction interface, so that the high abstraction to low abstraction direction forms the stimulus path while the low abstraction to high abstraction direction forms the analysis path.

The high abstraction interface is implemented as a seq_item_port-analysis_port pair, while the low abstraction interface is implemented as a seq_item_export-analysis_export pair.

The stimulus path is implemented by cascading one or more Translators, each with one or more optional orthogonal sequencers. The analysis path is implemented by cascading one or more Translators, each with one or more optional analysis taps. Ports of the first and last Translators in each path are “wired-out” of the Layer.

Like an Agent, the analysis path is always present and the stimulus path is present only if the the is_active bit is set to UVM_ACTIVE.
In general Layers should be as symmetric as possible, in that:
(1) the number of Translators required to convert between high
abstraction items and low abstraction items is generally the same
in both paths; (2) if orthogonal sequences are used to insert
information into the flow of data in the stimulus path, then there
is a corresponding translation in the analysis path that is used to
extract that information and optionally send it out an analysis tap;
and (3) the port types on the high and low abstraction interfaces
should be the same. This last point implies that a Layer can be
self-tested by looping back the low abstraction interface.

As an example, the following is an IEEE 10GBASE-R layer
that converts between packets and a bitstream:

![Figure 19: 10GBASE-R Layer](image)

Note that occasionally the constituent Translators must be
restructured in different contexts, as evidenced by the PCS
64b/66b encoder example previously discussed. But also note
that Layers imply a semantic on the constituent Translators but
they do not impose it, so that restructuring, although rare, is not as
onerous as recoding the translation.

B. Attachment Agents

Given that Layers do not have virtual interfaces, they cannot
directly connect to a device under test. This connection is made
via an Attachment Agent.

An Attachment Agent (AA) is defined an Agent (in the not-
layered sense) that does not have a sequencer connected to the
driver:

![Figure 20: An Attachment Agent, (AA)](image)

Like an Agent, the monitor is always present and the driver is
only present if the is_active bit is set to UVM_ACTIVE.

C. Chains

A Chain is defined as a uvm_component with an
is_active bit that connects a sequencer to an Attachment Agent
and has zero or more intervening Layers.

![Figure 21: A Chain, (C)](image)

A Chain is a Simple Chain (SC) if it has only one
intervening Layer and a Chainable Agent (CA) if it has no
intervening Layers. Note that a Chainable Agent is the
degenerate case, as the only difference between a Chainable
Agent and the traditional Agent is that the driver and monitor are
first bundled into an Attachment Agent:

![Figure 22: A Chainable Agent, (CA)](image)

Like an Agent, the sequencer and connection are absent if the
is_active bit is set to UVM_ACTIVE. The is_active bit of
the Attachment Agent and all intervening Layers, if any, are
bound to the is_active bit of the Chain.

V. USAGE CONTEXTS

Layers and Attachment Agents are the fundamental units
of portability between unit level testing and chip level testing.
Chains and Chainable Agents are structural integrations of Layers
and Attachment Agents and are generally used in only one scope.

An edge unit is a design under test that is verified at the unit
level where the low abstraction I/O at the unit level is also
exposed at the chip level. Edge units are tested with a Chainable
Agent at the unit level and the Attachment Agent is ported to a
Chain at the top level.

![Figure 23: Edge Units](image)

An internal unit is a design under test that is verified at the
unit level but does not have I/O exposed at the chip level.
Internal units are tested with a Simple Chain at the unit level and
the Layer is ported to a Chain at the top level.

![Figure 24: Internal Units](image)

If an edge unit is also an internal unit some other mode of
operation, the unit level environment must favor the internal unit
mode and use a Simple Chain instead of a Chainable Agent. This
may make the unit level Attachment Agent trivial, but a Layer is
required in at least one top level context.

An end unit is a design under test that is verified at the unit
level where the high abstraction interface is adjacent to the
protocol divide, which is the point at where no higher protocols
exist and the data flow crosses over from being received to being
transmitted. The end unit is very much an internal unit, as all
other units serve to bring data to it. End units are tested with a Simple Chain at the unit level and the Layer and Sequencer is ported to the top level.

Figure 25: End Units

VI. IMPLEMENTATION GUIDELINES

As Layers and Attachment Agents are the fundamental units of portability between unit and top level, care should be taken to isolate material targeted for porting from material targeted for a single scope when packaging.

Layers should be favored over Attachment Agents where possible, as Layers are more adaptable to new usage contexts.

For maximum portability and usability, Layers and Attachment Agents should behave in a legal fashion with minimal configuration.

Chain configuration objects must instantiate Layer and Attachment Agent configuration objects using the factory to allow for randomized control over those components in the Chain.

Layer and Attachment Agent configuration should be designed so that if no configuration object is passed or if a newly created, but unrandomized configuration object is passed that all translations are valid and legal. This implies that all configuration variables in a configuration object have valid, legal default values set within the new function.

Translators with orthogonal sequences must use the try_next_item interface and implement a valid, legal translation when either no orthogonal sequence is started or all orthogonal sequences have completed. Failure to use the try_next_item interface causes the translation to stall unless an orthogonal sequence is started. This burdens Chain designers, forcing them to find, create start persistent orthogonal sequences when in most top level contexts the valid, legal option is all that is required.

Lastly, a translate task must never call disable fork. Doing so not only disables the current translation but all cascaded translations that feed into the current translation, including any threads they may have started. Calling disable fork in your Translator will waste the better part of 3 days of someone else’s time, since a translation some layers away will appear to suddenly no longer work. (The corollary is that if you’ve spent 3 days pouring over why your translation suddenly no longer works in some context, suspect a disable fork by someone else).

VII. CONCLUSION

The Layered Architecture and the translator class upon which it is built were deployed at AppliedMicro in the spring of 2012, resulting in a suite of 16 Layers, 3 Attachment Agents and 2 utility Translators that are powering the unit level and chip level verification of our next generation of Datacom devices. The translator class now has over 400 extensions and seen over 240,000 simulation runs across upwards of 16,000 tests, so is very much a key component in the AppliedMicro verification arsenal. Much of the success resides in the familiarity of the UVM component/port connection orthodoxy, the simplicity of the Translation API in constructing new translations, and the power of Orthogonal Sequencing.

VIII. REFERENCES
