All Your Base Transactions Belong to Us

Using Mixins in Real-World UVM Applications

Jeff Vance, Alex Melikian

(Presented today by: Jason Sprott)
Problem Summary  – dealing with multiple transaction types
Concept  – the mixin design pattern
Solutions  – debug, checking, coverage, stimulus
Enhanced Solution  – mixin + interface class
PROBLEMS WITH MULTIPLE TRANSACTION TYPES
Type of Problem to Solve

• Transactions may come from different libraries/sources
• No common **mutable** root class
• Similar/identical code is repeated in key TB areas
  – Debug
  – Coverage
  – Scoreboards
  – Stimulus

**Goal:** a way to reuse common code across **all** transaction types
THE MIXIN CONCEPT
Add Common Features to Many Classes

Without a Mixin
- Extend *each* class
- Add variables and methods to *all*
- Duplicate code very likely
- Maintain *all* instances over time

With a Mixin
- Add variables and methods to *one* mixin class
- Extend mixing in new behavior

Maintain one class for shared code
Supports any number of classes
What is a Mixin?

Mix two class types to create a new type

A mixin is a parameterized class

Parameter specifies class to extend

class mixin_name#(type T = default_type) extends T;

Mix these 2 types

Define new type

Extend further (if necessary)

typedef mixin_name #(my_type) mixed_my_type;
EXAMPLE: MIXINS APPLIED TO TRANSACTIONS
Typical UVM Sequence Items

- All items extend from `uvm_object`
- All inherit standard methods
  - `convert2string()`
  - `sprint()`
  - `compare()`
- All follow different conventions for managing data
A Closer Look at Different Transaction Types

class vip_x_item extends uvm_sequence_item;
rand bit r_w;
rand bit[31:0] addr;
rand bit[7:0] data_bytes[];
rand bit[4:0] length;
rand bit[3:0] x_id;

class vip_y_item extends uvm_sequence_item;
rand y_dir_enum dir;
rand bit[31:0] addr; //for single
rand bit[31:0] start_addr; //for burst
rand bit[31:0] data[];
rand bit[4:0] len;
rand bit[3:0] y_id;

**Goal:** encapsulate these differences

- **array of bytes vs words**
- **bit vs enum for rd / wr**
- **2 Address fields vs 1**
A Generic System Transaction Mixin

- **Shared variables and methods**
  - `sys_item (mixin)`
- **typedef declarations**
- **Type-specific variables and methods**

```
vip_x_item
vip_y_item
vip_n_item

sys_x_item_base
sys_y_item_base
sys_n_item_base

sys_x_item
sys_y_item
sys_n_item

... extends
... extends
typed-version of mixin
```
Standardize Transaction Types

class sys_item#(type T=uvm_object) extends T;
    `uvm_object_param_utils(sys_item#(T))
...
endclass

typedef sys_item #(vip_x_item) sys_x_item_base;
typedef sys_item #(vip_y_item) sys_y_item_base;

Shared implementation: e.g., convert2string()

Virtual prototypes

implementation

Customized to use new features
VERIFICATION SOLUTIONS WITH MIXINS
class sys_item#(type T=uvm_object) extends T;
`uvm_object_param_utils(sys_item#(T))

rand sys_cmd_enum cmd;
rand sys_master_enum master;
rand sys_slave_enum slave;
sys_path_model refmodel;
...
int master_count;
int slave_count;

virtual function sys_slave_enum get_slave();
  return refmodel.get_slave_from_addr(get_addr());
endfunction
Debug Enhancement Example

common convert2string() implementation for all types

| TRLOG | MASTER= | SLAVE== | CMD==== | ADDR=== | LEN= | DATA===== | ===============
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TRLOG</td>
<td>MST_A</td>
<td>SLV_E</td>
<td>SYS_READ</td>
<td>1e1a1</td>
<td>26</td>
<td>--------</td>
<td>x_id: 1</td>
</tr>
<tr>
<td>TRLOG</td>
<td>MST_B</td>
<td>SLV_E</td>
<td>SYS_READ</td>
<td>180ee</td>
<td>28</td>
<td>--------</td>
<td>x_id: 6</td>
</tr>
<tr>
<td>TRLOG</td>
<td>MST_A</td>
<td>SLV_F</td>
<td>SYS_WRITE</td>
<td>2942c</td>
<td>4</td>
<td>debf8a54</td>
<td>x_id: f</td>
</tr>
<tr>
<td>TRLOG</td>
<td>MST_C</td>
<td>SLV_D</td>
<td>SYS_WRITE</td>
<td>5bea</td>
<td>12</td>
<td>2f135d3c</td>
<td>y_id: 5</td>
</tr>
<tr>
<td>TRLOG</td>
<td>MST_B</td>
<td>SLV_F</td>
<td>SYS_WRITE</td>
<td>25f85</td>
<td>8</td>
<td>c5096c73</td>
<td>x_id: 7</td>
</tr>
<tr>
<td>TRLOG</td>
<td>MST_C</td>
<td>SLV_E</td>
<td>SYS_READ</td>
<td>19e22</td>
<td>40</td>
<td>--------</td>
<td>y_id: c</td>
</tr>
<tr>
<td>TRLOG</td>
<td>MST_C</td>
<td>SLV_E</td>
<td>SYS_WRITE</td>
<td>1025f</td>
<td>8</td>
<td>afbf5954</td>
<td>y_id: 8</td>
</tr>
</tbody>
</table>

Easily identify source, destination, command

Addr, Length, & Data are consistent

A unique string to grep

Protocol-specific fields can be added
covergroup slv_e_cg with function sample(sys_y_item tr);
    e_master_cp: coverpoint tr.master;
    e_addr_cp: coverpoint tr.get_addr();
    e_yid_cp: coverpoint tr.y_id iff (sys_yif_cfg != IGNORE_ID)

    e_master_yid : cross e_master_cp, e_yid_cp;
endgroup
Mixin Solution for Scoreboarding

Common data integrity check functionality

• Encapsulate the scheme in a mixin

```c
virtual function void pack_for_sb(ref bit[7:0] pb[],
                      ref int num_bytes);
virtual function string disclose_pack_member(int position);
function void compare_pack_data(input bit[7:0] external_bytes,
                      input int external_num_bytes,
                      output int disagree_bit_pos)
```

General data types (bytes, string) used in method args

• `pack_for_sb`: pack transaction content into list of bit/bytes
• `disclose_pack_member`: get member string from a pack position
• `compare_pack_data`: compare with another transaction type
Using the Scoreboard Mixin

- Scoreboard uses the mixin methods:
  - Package `x_item` transaction so it can be compared
  - Use that data when comparing to `y_item` transaction
  - Common way to find actual field names from position

```
x_item.pack_for_sb(sb_pack_bytes, sb_num_bytes, disagree_str);
if (!y_item.compare_pack_data(sb_pack_bytes, sb_num_bytes, disagree_pos))
  `uvm_info(..., $sformatf("X item %s mismatch Y ...", x_item.disclose_pack_member(disagree_pos), [...]
```

Straightforward, but can it be improved?
MIXIN SOLUTION WITH INTERFACE CLASS
Enhance Mixin with an Interface Class

```cpp
x_item.pack_for_sb(sb_pack_bytes, sb_num_bytes);
if (!y_item.compare_pack_data(sb_pack_bytes,
                               sb_num_bytes,
                               disagree_pos))
    `uvm_info([...], $sformatf("X item %s mismatch Y ...",
                            x_item.discard_pack_member(disagree_pos), [...])
```

- So, how can we replace the above with this?

```cpp
if (!y_item.compare_pack_data(x_item, disagree_str)
```

Use an Interface Class
Changing the Mixin Template

• Move scoreboard methods into interface class

```plaintext
interface class sys_item_cif;
    pure virtual function void pack_for_sb(ref bit[7:0] pb[], ref int nb);
    pure virtual function string disclose_member_for_sb(int p);
endclass
```

• Implement the interface class on the mixin

```plaintext
sys_item #(type T=uvm_object)extends T implements sys_item_cif;
function bit compare_pack_data(sys_item_cif si_cif, ...);
    this.pack_for_sb(local_pb, local_nb);
    si_cif.pack_for_sb(other_pb, other1_nb);
    // compare local and other ...
endfunction
endclass
```

Anything using mixin can use the interface

We can now pass args of this interface type

When called we get concrete implementation
Using the Mixin with Interface Class

- No intermediary data members
- Can now do **type specific** pack and report inside compare

```c
x_item.pack_for_sb(sb_pack_bytes, sb_num_bytes);
if (!y_item.compare_pack_data(sb_pack_bytes, sb_num_bytes, disagree_pos))
    `uvm_info([...], $sformatf("X item %s mismatch Y ...",
               x_item.disclose_pack_member(disagree_pos), [...])

if (!y_item.compare_pack_data(x_item, disagree_str))
    `uvm_info([...], disagree_str, [...])
```

Interface Class typed argument
Covergroup Using Mixin with Interface

```verilog
covergroup value_cg(int min, max, num_bins);
    covergpoint x {
        bins lowest = {min};
        bins highest = {max};
        bins middle[num_bins] = {[min+1:max-1]}
    }
endcovergroup

function new(sys_item_cif tr);
    int max = tr.get_max();
    int min = tr.get_min();
    int n_bins = (max - min)/tr.get_values_per_bin();
    value_cg = new(min, max, n_bins)
endfunction

function void sample(sys_item_cif tr)
    value_cg.sample(tr.get_value)
endfunction
```

- **Covergroup with adaptable bin count**
- **Construct covergroup using mixin interface class**
- **Ditto for sampling**
Conclusions

• **Take control:** handle any UVM transaction type

• **Better manage**
  – Consistency: standardize look, feel, metrics across interfaces
  – Debug: correlate transactions of any type
  – Stimulus: share constraints between any type of transaction
  – Retrofit: factory override with mixin

• **Adaptable**— to any UVM testbench, including legacy projects

• **Extensible**— easily support future transaction types

Download the original paper & code at [www.verilab.com](http://www.verilab.com)  
“All Your Base Transactions Belong to Us”, 2019