2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
MARCH 4-7, 2024

Leveraging Interface Class to Improve UVM TLM

N Goyal, J Refice

Nvidia Corporation

NVIDIA

accellera SYSTEMS INITIATIVE

Interface Classes

- Introduced in System Verilog in 2012
  - Similar to virtual classes, with some exceptions
  - Helps us implement multiple inheritance
  - Powerful and efficient, but underutilized and underappreciated

Why are we looking at UVM TLM?

- Coded before System Verilog 2012 was released
  - Current code uses workarounds because Interface classes didn't exist

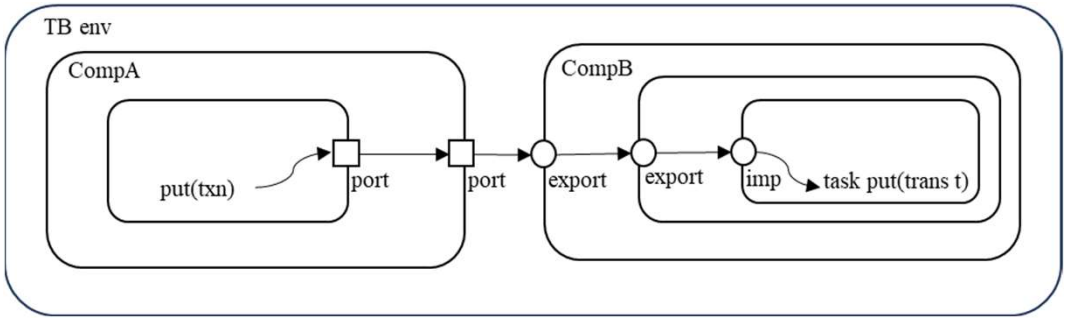
- Powerful use case for Interface classes
  - Moves the following error detection from run-time to compile-time
    - Port Connectivity
    - Missing implementation
    - Interface connectivity
    - Illegal method calls

We will talk about

- UVM TLM ports (Current implementation)
- Design Patterns
  - Interface Classes
  - Curiously Recurring Template Pattern (CRTP)
  - Mixins
- XVM TLM (Our new approach!)
  - xvm is just a name replacement for uvm for the ease of readability and co-existence.

UVM TLM

- Interfaces
  - Provide API requirements (blocking_get, peek, nonblocking_put, etc.)
- Ports
  - Provide connectivity requirements (ports, exports, imps)



UVM TLM port: uvm_blocking_put_port, uvm_peek_imp, etc.

Port Base Class

- Common base class for ports, exports and imps

```
uvm_port_base #(type IF = uvm_void) extends IF;
```

- All ports, exports, imps use `connect` method in common base class
  - Illegal method calls fail at run time
    - `imp.connect();` – nonsensical!
    - `export.connect(port);` – wrong direction!

Interface Base Class

- Common base class used for IF type parameter for port base class

```
uvm_tlm_if_base #(type T1 = int, type T2 = int)
```

- Base class lists all methods for all types of TLM interfaces.
  - No implementation of some methods – missing definition!
  - Not declared pure virtual
    - `uvm_tlm_get_port.connect(uvm_analysis_imp);` – not compatible!

Design Patterns

- Interface Classes
- Curiously Recurring Template Pattern (CRTP)
- Mixins

Interface Classes

- Interface classes added in 2012

```
interface class FooIntf;
  pure virtual function void do_foo();
endclass
```

- Similar to virtual classes, except
  - No constructor
  - All methods are pure virtual
  - No member variable or static methods
- Interface classes may extend other interface classes

```
interface class FooBarIntf
  extends FooIntf, BarIntf;
```

Implementing Interface Classes

- *Classes and Virtual Classes* use implements for interfaces (instead of extends):

```
class Foo
  extends Bar
  implements FooIntf;
```

- Class must always provide an explicit definition of its interface class methods, even pure virtual

```
virtual class Foo
  extends Bar
  implements FooIntf;
  ...
  pure virtual function void do_foo();
endclass
```

- Classes are $cast() compatible with interface classes, just like base classes

```
Foo FooA = new(), FooB = null;
FooIntf MyIntf = FooA; // Upcast
$cast(FooB, MyIntf); // Downcast
```

Curiously Recurring Template Pattern

- *Curiously Recurring Template Pattern* (CRTP)

```
class CuriousFoo#(type T) extends T;
  ...
endclass

typedef CuriousFoo#(Bar) FooBar;
typedef CuriousFoo#(Baz) FooBaz;
```

- FooBar and FooBaz do not share a common base class
- Common member variables
- Common methods with or w/o default implementations

Curiously Recurring Template Pattern

- *Curiously Recurring Template Pattern* (CRTP)

```
class CuriousFoo#(type T) extends T;
  ...
endclass

typedef CuriousFoo#(Bar) FooBar;
typedef CuriousFoo#(Baz) FooBaz;
```

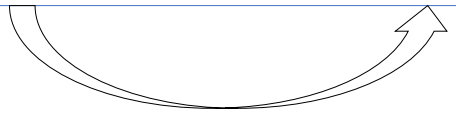
- FooBar and FooBaz do not share a common base class
- Common member variables
- Common methods with or w/o default implementations

…… where have we seen it before?

Curiously Recuring Template Pattern

- In UVM TLM port base class!

```
uvm_port_base #(type IF = uvm_void) extends IF;
```



Curiously Recurring Template Pattern

```
class CuriousFoo#(type T) extends T;
  ...
  int value;
endclass

typedef CuriousFoo#(Bar) FooBar;
typedef CuriousFoo#(Baz) FooBaz;
```

- FooBar and FooBaz have their own copies of value which is not shared.
- We could use an interface class to access this variable!

```
interface class FooIntf;
  pure virtual function void set_value (int v);
  pure virtual function int get_value ();
endclass
```

The Mixin Pattern

- The *Mixin* pattern relies on a combination of *Interface Classes* and *CRTP*
  - The *Mixin* implements the interface
    - provides default implementation, even if it is pure virtual
  - The *Mixin* is your single base class

- FooBar and FooBaz do not share a common base class
  ...but they DO share a common interface!

```
class FooMixin#(type T)
  extends T
  implements FooIntf;

  function new ();
    super.new();
  endfunction : new

  virtual function void do_foo();
    $display("I'm doing foo!");
  endfunction : do_foo
endclass

typedef FooMixin#(Bar) FooBar;
typedef FooMixin#(Baz) FooBaz;
```

Multiple Inheritance

- Parameterized *Mixins* can be passed as parameters to other *Mixins*, or to themselves

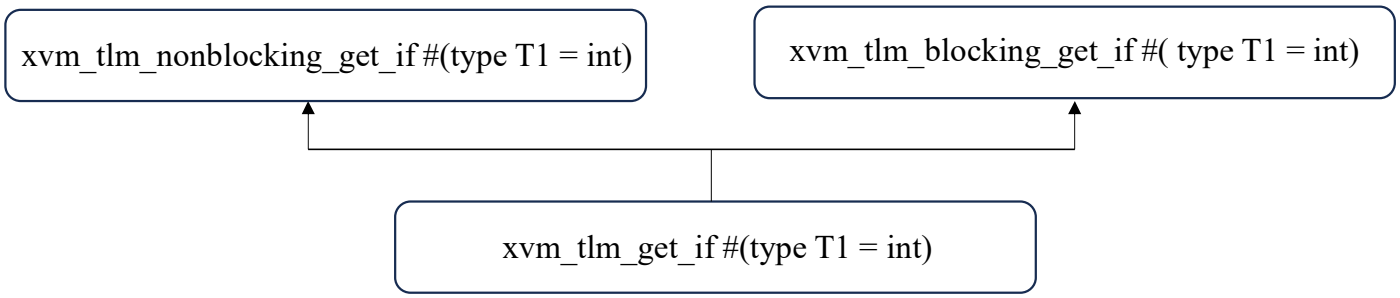
```
// Class declaration with multiple mixins
class MyMegaClass
  extends MegaMixin#(
            FooMixin#(
              BarMixin#( Base )
                       )
                      );
  ...
endclass

// Cast compatibility for all...
MyMegaClass mega = new();
MegaIntf mega_if = mega; // YES
FooIntf foo_if = mega;   // YES
BarIntf bar_if = mega;   // YES
$cast(mega, mega_if);    // YES
$cast(mega, foo_if);     // YES
$cast(mega, bar_if);     // YES
```

XVM TLM - New Implementation

- Interface classes for Interfaces
- Interface classes for Ports
- Interface classes for Connectivity checks
- TLM ports using Interface Classes
- Examples of compile-time checks

Specific Interface classes for TLM Interfaces

- One interface class for each interface instead of a single uvm_tlm_if_base class.



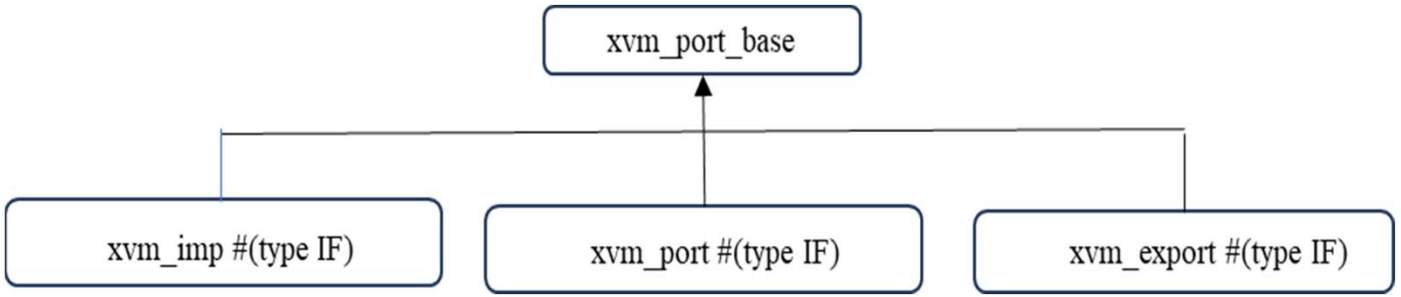
Interface classes for Interfaces

```
101      interface class xvm_tlm_nonblocking_get_if#(type T1=int);
102        pure virtual function bit try_get(output T1 t);
103        pure virtual function bit can_get(T1 t);
104      endclass : xvm_tlm_nonblocking_get_if
105
106      interface class xvm_tlm_blocking_get_if#(type T1=int);
107        pure virtual task get(output T1 t);
108      endclass : xvm_tlm_blocking_get_if
109
110      interface class xvm_tlm_get_if#(type T1=int)
110         extends xvm_tlm_nonblocking_get_if#(T1),
111                 xvm_tlm_blocking_get_if#(T1);
112      endclass : xvm_tlm_get_if
```

- pure virtual declaration means compile-time failure for a missing definition!

Specific Base Classes for TLM Ports

- Separate base classes for port, export and imps
  - Derived from a common non-parameterized base class
  - Common base class only has functionality common to all three types



Specific Base Classes for Ports

```
201    virtual class xvm_port_base;
202
203    virtual class xvm_export #(type IF=uvm_void)
204       extends xvm_port_base
205       implements xvm_export_check_if#(IF);
206
207    virtual class xvm_imp #(type IF=uvm_void)
208       extends xvm_port_base
209       implements xvm_export_check_if#(IF);
210
211    virtual class xvm_port #(type IF=uvm_void)
212       extends xvm_port_base
213       implements xvm_port_check_if#(IF);
```

- xvm_imp does not have a connect method, this guarantees a compile time failure on imp.connect() call!

Interface Classes for Connectivity Checks

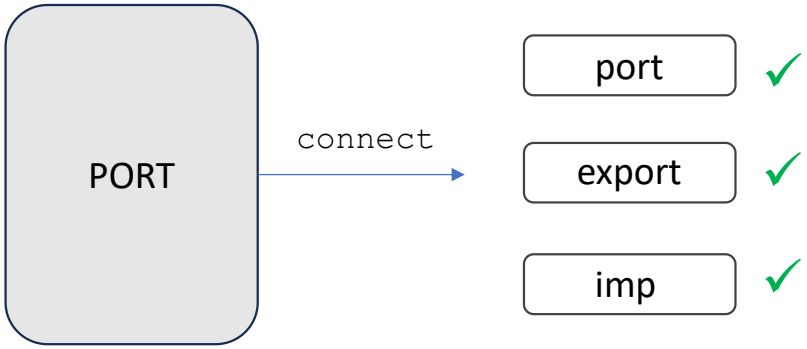
- ports, exports and imp implements one of the following two classes:

```
301    interface class xvm_port_check_if#(type IF);
302    endclass: xvm_port_check_if
303
304    interface class xvm_export_check_if#(type IF)
305      extends xvm_port_check_if#(IF);
306    endclass: xvm_export_check_if
```

- The connect method in xvm_port and xvm_export uses one of these class types as an argument

connect method in xvm_*_port

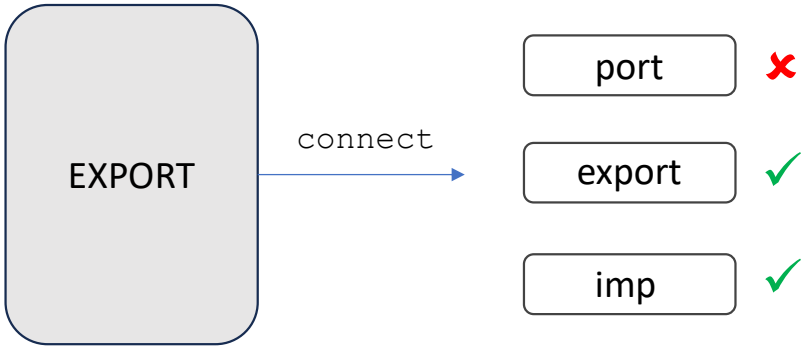
```
virtual function void connect (xvm_port_check_if #(IF) provider);
…
endfunction
```




```
// Cast compatibility ...

$cast(xvm_port_check_if, xvm_*_port);     // YES
$cast(xvm_port_check_if, xvm_*_export);   // YES
$cast(xvm_port_check_if, xvm_*_imp);      // YES
```

connect method in xvm_*_export

```
virtual function void connect (xvm_export_check_if #(IF) provider);
…
endfunction
```




```
// Cast compatibility ...

$cast(xvm_export_check_if, xvm_*_port);   // NO
$cast(xvm_export_check_if, xvm_*_export);// YES
$cast(xvm_export_check_if, xvm_*_imp);    // YES
```

- export.connect(port) will be a compile-time failure!

XVM TLM Ports

- Putting all the three definitions together we get:

```
class xvm_get_export #(type T=int)
  extends xvm_export #(xvm_tlm_get_if #(T))
  implements xvm_export_check_if #(xvm_tlm_get_if #(T)),
             xvm_tlm_get_if #(T);
   ...
endclass
```

XVM TLM Ports

- Putting all the three definitions together we get:

```
class xvm_get_export #(type T=int)
  extends xvm_export #(xvm_tlm_get_if #(T))
  implements xvm_export_check_if #(xvm_tlm_blocking_get_if #(T)),
             xvm_tlm_get_if #(T);
   ...
endclass
```

XVM TLM Ports

- Putting all the three definitions together we get:

```
class xvm_get_export #(type T=int)
  extends xvm_export #(xvm_tlm_get_if #(T))
  implements xvm_export_check_if #(xvm_tlm_get_if #(T)),
             xvm_tlm_get_if #(T);
   ...
endclass
```

XVM TLM Ports

- Putting all the three definitions together we get:

```
class xvm_get_export #(type T=int)
  extends xvm_export #(xvm_tlm_get_if #(T))
  implements xvm_export_check_if #(xvm_tlm_get_if #(T)),
             xvm_tlm_get_if #(T);
   ...
endclass
```

XVM TLM Ports

- Putting all the three definitions together we get:

```
class xvm_get_export #(type T=int)
  extends xvm_export #(xvm_tlm_get_if #(T))
  implements xvm_export_check_if #(xvm_tlm_get_if #(T)),
             xvm_tlm_get_if #(T);
   ...
endclass
```

… it could get verbose

```
class xvm_get_peek_export #(type T=int)
 extends xvm_export #(xvm_tlm_get_peek_if_base #(T))
 implements xvm_export_check_if #(xvm_tlm_blocking_get_peek_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_blocking_get_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_blocking_peek_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_nonblocking_get_peek_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_nonblocking_get_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_nonblocking_peek_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_get_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_peek_if_base #(T)),
            xvm_export_check_if #(xvm_tlm_get_peek_if_base #(T)),
            xvm_tlm_get_peek_if_base #(T);
 . . .
endclass
```

XVM TLM ports with mixins

- Declare a mixin class

```
virtual class xvm_get_peek_export_pure_mixin #(type T=int, type BASE=int)
   extends xvm_peek_export_pure_mixin #(T,
           xvm_get_export_pure_mixin #(T,
           xvm_blocking_get_peek_export_pure_mixin #(T,
           xvm_nonblocking_get_peek_export_pure_mixin #(T, BASE))))
   implements xvm_tlm_get_peek_if #(T),
              xvm_export_check_if #(xvm_tlm_get_if #(T)),
              xvm_export_check_if #(xvm_tlm_peek_if #(T)),
              xvm_export_check_if #(xvm_tlm_get_peek_if #(T));
  . . .
endclass // xvm_get_peek_export_pure_mixin
```

XVM TLM ports with mixins

- Declare the TLM using the mixin

```
class xvm_get_peek_export #(type T=int)
 extends xvm_get_peek_export_pure_mixin#(T, xvm_export #(xvm_tlm_get_peek_if #(T)));
 . . .
endclass
```

```
class consumer_bar extends uvm_component;
  `uvm_component_utils(consumer_bar)
  xvm_put_imp#(txn, consumer_bar)cons_imp; // FAILURE 1
  function bit try_put(txn t);
  `uvm_info("TRY_PUT", $sformatf("txn value is %d", t.data), UVM_NONE);
  return 1;
  endfunction
  function bit can_put();
    `uvm_info("CAN_PUT", "Yes, we can put", UVM_NONE);
    return 1;
  endfunction
...
endclass

class test extends uvm_test;
`uvm_component_utils(test)
 ...

 function void connect_phase(uvm_phase phase);
  prod.prod_exp.connect(cons_foo.cons_port); // FAILURE 2
  prod.prod_b_exp.connect(cons_foo.cons_nb_imp);// FAILURE 3
  cons_foo.cons_nb_imp.connect(prod.prod_exp); // FAILURE 4
 endfunction

endclass
```

```
class consumer_bar extends uvm_component;
  `uvm_component_utils(consumer_bar)
  xvm_put_imp#(txn, consumer_bar)cons_imp; // FAILURE 1 - MISSING IMPLEMENTATION
  function bit try_put(txn t);
  `uvm_info("TRY_PUT", $sformatf("txn value is %d", t.data), UVM_NONE);
  return 1;
  endfunction
  function bit can_put();
    `uvm_info("CAN_PUT", "Yes, we can put", UVM_NONE);
    return 1;
  endfunction
...
endclass

class test extends uvm_test;
`uvm_component_utils(test)
 ...

 function void connect_phase(uvm_phase phase);
  prod.prod_exp.connect(cons_foo.cons_port); // FAILURE 2
  prod.prod_b_exp.connect(cons_foo.cons_nb_imp);// FAILURE 3
  cons_foo.cons_nb_imp.connect(prod.prod_exp); // FAILURE 4
 endfunction

endclass
```

```
class consumer_bar extends uvm_component;
  `uvm_component_utils(consumer_bar)
  xvm_put_imp#(txn, consumer_bar)cons_imp; // FAILURE 1
  function bit try_put(txn t);
  `uvm_info("TRY_PUT", $sformatf("txn value is %d", t.data), UVM_NONE);
  return 1;
  endfunction
  function bit can_put();
    `uvm_info("CAN_PUT", "Yes, we can put", UVM_NONE);
    return 1;
  endfunction
...
endclass

class test extends uvm_test;
`uvm_component_utils(test)
 ...

 function void connect_phase(uvm_phase phase);
  prod.prod_exp.connect(cons_foo.cons_port); // FAILURE 2 - ILLEGAL CONNECTION
  prod.prod_b_exp.connect(cons_foo.cons_nb_imp);// FAILURE 3
  cons_foo.cons_nb_imp.connect(prod.prod_exp); // FAILURE 4
 endfunction

endclass
```

```
class consumer_bar extends uvm_component;
  `uvm_component_utils(consumer_bar)
  xvm_put_imp#(txn, consumer_bar)cons_imp; // FAILURE 1
  function bit try_put(txn t);
  `uvm_info("TRY_PUT", $sformatf("txn value is %d", t.data), UVM_NONE);
  return 1;
  endfunction
  function bit can_put();
    `uvm_info("CAN_PUT", "Yes, we can put", UVM_NONE);
    return 1;
  endfunction
...
endclass

class test extends uvm_test;
`uvm_component_utils(test)
 ...

 function void connect_phase(uvm_phase phase);
  prod.prod_exp.connect(cons_foo.cons_port); // FAILURE 2
  prod.prod_b_exp.connect(cons_foo.cons_nb_imp);// FAILURE 3 - INCOMPATIBLE PORTS
  cons_foo.cons_nb_imp.connect(prod.prod_exp); // FAILURE 4
 endfunction

endclass
```

```
class consumer_bar extends uvm_component;
  `uvm_component_utils(consumer_bar)
  xvm_put_imp#(txn, consumer_bar)cons_imp; // FAILURE 1
  function bit try_put(txn t);
  `uvm_info("TRY_PUT", $sformatf("txn value is %d", t.data), UVM_NONE);
  return 1;
  endfunction
  function bit can_put();
    `uvm_info("CAN_PUT", "Yes, we can put", UVM_NONE);
    return 1;
  endfunction
...
endclass

class test extends uvm_test;
`uvm_component_utils(test)
 ...

 function void connect_phase(uvm_phase phase);
  prod.prod_exp.connect(cons_foo.cons_port); // FAILURE 2
  prod.prod_b_exp.connect(cons_foo.cons_nb_imp);// FAILURE 3
  cons_foo.cons_nb_imp.connect(prod.prod_exp); // FAILURE 4 - ILLEGAL METHOD CALL
 endfunction

endclass
```

```
class consumer_bar extends uvm_component;
  `uvm_component_utils(consumer_bar)
  xvm_put_imp#(txn, consumer_bar)cons_imp; // FAILURE 1
  function bit try_put(txn t);
  `uvm_info("TRY_PUT", $sformatf("txn value is %d", t.data), UVM_NONE);
  return 1;
  endfunction
  function bit can_put();
    `uvm_info("CAN_PUT", "Yes, we can put", UVM_NONE);
    return 1;
  endfunction
...
endclass

class test extends uvm_test;
`uvm_component_utils(test)
 ...

 function void connect_phase(uvm_phase phase);
  prod.prod_exp.connect(cons_foo.cons_port); // FAILURE 2
  prod.prod_b_exp.connect(cons_foo.cons_nb_imp);// FAILURE 3
  cons_foo.cons_nb_imp.connect(prod.prod_exp); // FAILURE 4
 endfunction

endclass
```

Multiple Inheritance in SV

- We created a more powerful version of UVM TLM
- What can you do with it?

Questions?

- Code available at https://github.com/nv-negoyal/uvm-core-official/tree/xvm_tlm