Leveraging Interface Classes to Improve UVM TLM

N. Goyal, J. Refice
Nvidia Corporation
2888 San Tomas Expwy
Santa Clara, CA 95051

***Abstract*- Interface classes introduced multiple inheritance to System Verilog in 2012. With that a class isn't tied only to it's base class but can also inherit properties from other classes. Users prior to 2012 had to work their way around as the language did not have Interface classes, which includes the code for current Transaction Level Modeling (TLM) specification in the Universal Verification Methodology (UVM). It lacks compile-time checks for port and interface compatibility and missing implementations. Additionally, it leaks APIs between different interfaces, allowing nonsensical and illegal method calls that are only detectable at run-time. With the introduction of Interface classes in SystemVerilog 2012, we can rethink UVM TLM such that illegal and nonsensical behavior can be detected at compile-time, reducing the latency for the user to address these errors.**

I. INTRODUCTION

TLM ports and interfaces are an integral part of any testbench. Unfortunately, the current definition within the UVM Standard [1] is a weak imitation of SystemC's TLM [2]. The two implementations diverge significantly due to limitations in SystemVerilog 2009 [3]. As a result, UVM is functional but cannot check for common errors at compile-time and introduces nonsensical errors that are only detectable during run-time. For example, trying to call the `connect` method on an imp, `peek` on an analysis port, or connecting an export to a port.

In this paper, we explore using Interface classes to redefine UVM TLM; offering compile-time checks for port connectivity, interface compatibility, and incomplete interface implementation while preventing interface APIs from leaking into one another. While the design patterns used in this paper are applicable across both TLM 1 and 2, a redefinition of UVM TLM 2 is not included.

II. INTERFACES AND PORTS

TLM enables transaction-level communication between entities using two concepts: *Interfaces* and *Ports*. *Interfaces* provide the ability to specify API requirements without relying on inheritance. Consumers of an interface interact with implementors of the interface without knowing specific type information about the implementor. All the consumer knows is that the implementor implements the interface.

Alternatively, *ports, exports*, and *imps* control consumer-to-implementor connectivity through a hierarchical testbench. A *port* declares that a component requires an implementation of a specific *interface*, an *export* declares that the component forwards an implementation, and an *imp* declares that the component provides the implementation.

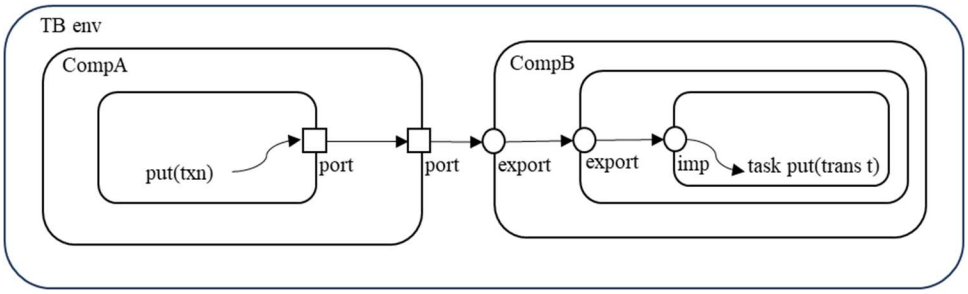

Figure 1. Ports, Exports and Imps

SystemC's TLM [2] has only ports and exports. There are no imps. UVM had to introduce imps because of the lack of single inheritance and method overloading in SystemVerilog 2009 [3]. UVM had to separate the two constructs, one forwarding the implementation, the export and the other providing the implementation, the imps.

III. THE UVM TLM IMPLEMENTATION

A. *Port base class*

The UVM TLM implementation defines the common base class for all ports, exports and imp types as:

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

As all port, export, and imp types use this base class, their common connect method can use the `uvm_port_base#(T)` type for its argument. This ensures connect-compatibility between ports, exports, and imps. Unfortunately, this also means that some methods always exist, even if they are illegal for a particular derivation type. For example, a user can call the `connect` method on an imp, only to fail later at run-time. This failure is because imps don't connect to anything; their actual implementation is defined during their construction. At run-time, the `connect` method checks for the type of `uvm_port_base#(IF)` instance, and if it is of type imp, it fails.

Even if calling the `connect` method is legal, it may have an incompatible port type as an argument. For example, if the `connect` method is called on an export type, with an argument of port type, we will see yet another run-time failure.

B. *Interface base class*

The `IF` type parameter used in `uvm_port_base#(IF)` is derived from the `uvm_tlm_if_base` base class for all TLM ports.

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

This base class lists all the methods needed for all types of TLM interfaces. While we're omitting the complete list for brevity, this means that every UVM analysis port contains 11 TLM APIs that are completely nonsensical, such as `can_put`. There is no actual implementation of these methods in the base class, and they are not declared `pure virtual`.

It is also possible to connect incompatible interfaces because of this common base class. For example, a `uvm_tlm_get_port` can be connected to a `uvm_analysis_imp`.

IV. DESIGN PATTERNS

The following design patterns allow us to create a new implementation of TLM for UVM without falling into traps described in the previous section.

A. *The Curiously Recurring Template Pattern*

The Curiously Recurring Template Pattern [4], or "CRTP," is a design pattern wherein a class extends from one of its parameters, for example:

```
class Baz #(type T)
  extends T;
endclass
```

In the example, the class `Baz` takes a parameter `T`, which it then extends, thus making its inheritance "curiously" recursive. As the base class is a parameter, each declaration of the `Baz` class may derive from a different base class:

```
class Base_Foo;
endclass

class Base_Bar;
endclass

Baz #(Base_Foo) Foo_ext;
Baz #(Base_Bar) Bar_ext;
```

In the example above, both `Foo_ext` and `Bar_ext` are derivations of `Baz`, but each derivation is derived from a different base class. `Foo_ext` derives from `Baz#(Base_Foo)`, which in turn derives from `Base_Foo`, and

`Bar_ext` derives from `Baz#(Base_Bar)`, which in turn derives from `Base_Bar`. `Foo_ext` and `Bar_ext` do not share a common base and are not cast-compatible with each other.

The CRTP should look familiar, as it is the pattern used by the `uvm_port_base#(IF)` class described in the previous section. This lack of a common base class between disparate `uvm_port_base#(IF)` declarations is the root cause of many of the sacrifices made in the original UVM TLM implementation. However, the CRTP can still be a handy tool when combined with additional patterns.

B. Interface Classes

SystemVerilog introduced I*nterface classes* in 2012 [5]. Interface classes are similar to abstract classes in that they specify method prototypes without providing an implementation. Unlike abstract classes, all methods within an Interface class must be declared `pure virtual`. Additionally, Interface classes are not a part of standard class inheritance. They do not have a constructor, and no member variables or static methods are allowed.

Instead of extending an Interface class, a class *implements* an Interface class. The following example from SystemVerilog 2012 LRM demonstrates the use of Interface classes.

```
interface class PutImp#(type PUT_T = logic);
   pure virtual function void put(PUT_T a);
endclass

interface class GetImp#(type GET_T = logic);
   pure virtual function GET_T get();
endclass

class Fifo#(type T = logic, int DEPTH=1) implements PutImp#(T), GetImp#(T);
  T myFifo [$:DEPTH-1];
  virtual function void put(T a);
    myFifo.push_back(a);
  endfunction
  virtual function T get();
    get = myFifo.pop_front();
  endfunction
endclass

class Stack#(type T = logic, int DEPTH=1) implements PutImp#(T), GetImp(T);
  T myFifo [$:DEPTH-1];
  virtual function void put(T a);
    myFifo.push_front(a);
  endfunction
  virtual function T get();
    get = myFifo.pop_front();
  endfunction
endclass
```

The implementing class must implement every method in the Interface class, including re-declaring `pure virtual` prototypes in abstract implementation classes. For instance, in the example above, the class `Stack` implements Interface class `PutImp #(T)`. If the user missed the definition of function `put()`, it would result in a failure during compile-time. We use this feature to help us detect missing definitions in the next section.

An Interface class establishes the protocol for how classes can declare various methods. The classes implementing this Interface class can define the methods in their own unique way. A class is allowed to implement multiple Interface classes, thus allowing for something like multiple inheritance. As with non-Interface classes, Interface classes support both type and value parameters.

C. The Mixin Pattern

The mixin pattern uses a combination of Interface classes and CRTP to provide multiple inheritance in System Verilog. It provides a default implementation of an interface and extends from a class that is a type parameter for the mixin class. The name “mixin” comes from how the class “mixes” the interface “into” the base class. In this way, two classes derived from the mixin class implement the same interface but may be derived from different base classes.

For instance, in the example below, FooBar and FooBaz implement FooIntf, even though they derive from different base classes:

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

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

A mixin class can also be used as the "base" parameter for other mixin classes. In the example below, class MyMegaClass is an extension of MegaMixin, which in turn extends FooMixin, which then extends BarMixin, which finally extends Base. As such, MyMegaClass contains all the properties of these classes.

```
class MyMegaClass
  extends MegaMixin#(
            FooMixin#(
              BarMixin#( Base )
                      )
                    );
  ...
endclass
```

V. NEW IMPLEMENTATION

As Interface classes allow type parameters and multiple inheritance, we use them to define ports and interfaces with the benefit of adding compile-time checks. Our new implementation uses Interface classes extensively in both the port and interface types, providing compile-time checks for missing methods, and both port and interface compatibility.

The new implementation uses the prefix xvm_ instead of uvm_. This prefix allows it to coexist with the current UVM definitions and avoids confusion caused by similar naming patterns.

A. Use of Interface classes for interfaces

The new implementation uses different Interface classes for each interface instead of a singular uvm_tlm_if_base class. The following is an example of the get interface type, which extends both the blocking and non-blocking get interface types.

```
interface class xvm_tlm_nonblocking_get_if#(type T1=int);
   pure virtual function bit try_get(output T1 t);
   pure virtual function bit can_get(T1 t);
endclass : xvm_tlm_nonblocking_get_if

interface class xvm_tlm_blocking_get_if#(type T1=int);
   pure virtual task get(output T1 t);
endclass : xvm_tlm_blocking_get_if

interface class xvm_tlm_get_if#(type T1=int)
  extends xvm_tlm_nonblocking_get_if#(T1),
          xvm_tlm_blocking_get_if#(T1);
endclass : xvm_tlm_get_if
```

All new XVM TLM ports use Interface classes like these in their declaration. The pure virtual guarantee a compile-time failure for any missing definitions in the classes implementing these interfaces. For example, if the user forgets to define the put() method in their class which extends xvm_blocking_put_imp, they are going to get a compile-time error.

B. Port specific base classes

The new implementation uses separate base classes for ports, exports and imps. We can eliminate methods and variables not needed in the specific types. They do share a common base class; however, it does not contain any functionality that doesn't apply across all three types. As all the classes do not have all the methods, calling an illegal method will fail at compile-time. For example, the `xvm_imp` class does not have a `connect` method. Hence, calling a `connect` method on an `xvm_imp` instance will result in a compile-time failure.

Note that the new implementation does not strictly require a corollary to imps in UVM. Using Interface classes, providing `xvm_port` and `xvm_export` would be sufficient. However, we continue to maintain an `xvm_imp` type to provide backward compatibility.

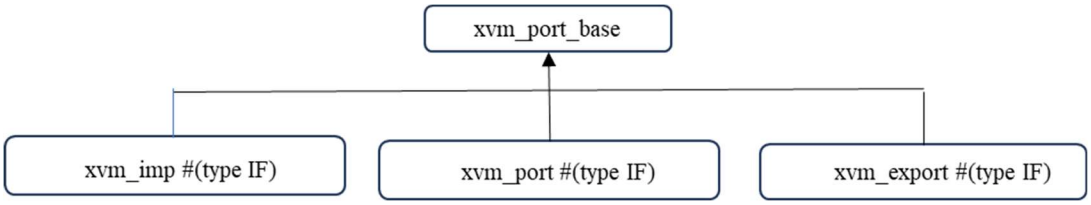

Figure 2. Class hierarchy for port classes

C. Use of Interface classes for connectivity checks

We provide two additional Interface classes port connectivity checks: `xvm_port_check_if` and `xvm_export_check_if`. Each extension of port, export and imp must implement one of these two classes.

```
interface class xvm_port_check_if#(type IF);
endclass: xvm_port_check_if

interface class xvm_export_check_if#(type IF)
  extends xvm_port_check_if#(IF);
endclass: xvm_export_check_if
```

Note that the Interface class `xvm_export_check_if` extends `xvm_port_check_if`. This may seem backwards since ports can connect to everything whereas exports can only connect to exports and imps. The reasoning for this definition is that the `connect` method uses one of these Interface classes as the argument type.

The connect method in the xvm_port_derived is declared as:

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

Whereas the connect method in xvm_export_derived is declared as:

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

All exports and imps implement the `xvm_export_check_if` Interface class, and all ports implement `xvm_port_check_if` class. An `xvm_export_check_if` is cast-compatible with `xvm_port_check_if` as it extends that Interface class. Hence the `xvm_port_check_if` argument for the `xvm_port connect` method will accept all ports, exports, and imps. On the other hand, an `xvm_export` will only accept export and imps as arguments to its `connect` method. This allows for a compile-time compatibility check between different TLM port types.

D. Declaration of TLM ports using Interface classes

Below is an excerpt from the definition of the get export class:

```
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
```

As you can see the class definitions get very verbose because of the implementation of multiple Interface classes. This is where the mixin pattern can be used once again to prevent code duplication. We declare mixin classes separately and use those mixin classes to have a more readable definition of TLM ports.

Such a mixin class is show below:

```
virtual class xvm_get_export_pure_mixin #(type T=int, type BASE=int)
   extends xvm_blocking_get_export_pure_mixin #(T,
              xvm_nonblocking_get_export_pure_mixin #(T, BASE))
   implements xvm_tlm_get_if #(T),
              xvm_export_check_if #(xvm_tlm_get_if #(T));
   ...
endclass // xvm_get_export_pure_mixin
```

This “pure” mixin class does not provide any concrete implementation for the get interface, instead it declares all of the methods as `pure virtual`. This allows the export, and imp classes associated with the get interface to share a single mixin instead of constantly re-declaring all of the interfaces that they implement. A similar mixin class can be declared for the ports.

The final declaration of a TLM Port and interface looks like the following with the use of this mixin class:

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

class xvm_get_imp #(type T=int)
 extends xvm_get_export_pure_mixin#(T, xvm_imp #(xvm_tlm_get_if #(T)));
   ...
endclass
```

These “pure” mixin classes can also be used to recursively to reduce the verbosity of other TLM Ports. For example:

```
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
```

E. Examples of compile-time checks

The following example user code contains 4 errors, all of which would previously have been caught during run-time, but are now detectable at compile-time:

```
class txn extends uvm_sequence_item;
  `uvm_object_utils(txn);
  rand bit[31:0] data;
  ...
endclass

class producer extends uvm_component;
  `uvm_component_utils(producer)
  xvm_put_export#(txn) prod_exp;
  xvm_blocking_put_export#(txn) prod_b_exp;
  ...
endclass

class consumer_foo extends uvm_component;
  `uvm_component_utils(consumer_foo)
  xvm_put_port#(txn)cons_port;
  xvm_nonblocking_put_imp#(txn, consumer_foo)cons_nb_imp;
  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 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 new(string name = "test", uvm_component parent = null);
  super.new(name, parent);
endfunction

producer prod;
consumer_foo cons_foo;
consumer_bar cons_bar;
```

```
function void build_phase(uvm_phase phase);
    prod = producer::type_id::create("prod", this);
    cons_bar = consumer_bar::type_id::create("cons_bar", this);
    cons_foo = consumer_foo::type_id::create("cons_foo", this);
endfunction

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

task run_phase(uvm_phase phase);
   phase.raise_objection(this);
   ...
   phase.drop_objection(this);
endtask
endclass
```

In the example shown above, we are going to see two compile-time failures, which would otherwise have been a run-time failure. Here is a description of all the failures mentioned above as FAILURE #

1. Missing Implementation – There will be a compile-time failure at this line as there is a missing implementation of `can_put()` method in the consumer class. The consumer class which has a `xvm_put_imp` should have all the three methods, `put()`, `try_put()` and `can_put()` implemented as it implements the Interface classes `xvm_blocking_put_if` and `xvm_nonblocking_put_if` which has these three methods defined as pure virtual.
2. Illegal Port Connectivity - The attempt to use port type as an argument to the connect method of an export will fail at compile-time because the port type is of `xvm_port_check_if` Interface class and the argument to an export's connect method can only be of `xvm_export_check_if` type.
3. Illegal Interface Connectivity – The attempt to connect an import of nonblocking types with an export of blocking type will fail as the blocking export does not implement the `xvm_nonblocking_put_if` Interface class and hence the nonblocking import will not be class compatible with the blocking export type.
4. Illegal Method Calls – Calling `connect` method on an imp will fail at compile-time now as this method does not exist in the base class `xvm_imp`. It would have failed at run-time in the previous implementation as these checks would have been done at run-time.

VI. CONCLUSION

The UVM TLM implementation provides the features a user may need but lacks the ability to check legality at compile-time, burdening the user with run-time. Taking advantage of modern design patterns such as Interface classes and Mixins, we have rearchitected our TLM implementation to detect common errors, such as illegal port connections, illegal method calls, incompatible interfaces, and incomplete interface implementations at compile-time. By shifting these checks from run-time to compile-time, we have reduced the latency to detect such errors, allowing DV engineers to complete their work faster.

The code for 'xvm' classes is released in GitHub under Apache 2.0 License [6].

REFERENCES

[1] IEEE Std 1800.2™, IEEE Standard for Universal Verification Methodology Language Reference Manual, 2020.
[2] OSCI, TLM-2.0 Language Reference Manual, 2009.
[3] IEEE Std 1800™, IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language, 2009.
[4] J.O. Coplien, "Curiously Recurring Template Patterns," C++ Report, February 1995.
[5] IEEE Std 1800™, IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language, 2012.
[6] https://github.com/nv-negoyal/uvm-core-official/tree/xvm_tlm