SystemVerilog Interface Classes
More Useful Than You Thought

Stan Sokorac
stan.sokorac@arm.com
Introduction

• Concept of “interface” popularized in Java
  – Introduced as “protocol” in Objective-C
  – Most modern language implement something like it
  – Different from SystemVerilog interface!

• Introduced into SV fairly late (2012)
  – Not used in UVM
  – Lack of adoption in DV community

• Heavy use in ARM® CPU verification
  – Clean and flexible testbenches
Observer Pattern

• Observer (also known as Subscriber, or Listener) pattern commonly used in all TBs
• Monitor observes an event
  – Then, it notifies listeners

```c
function void notify_observers(resolve_txn resolve);
    m_scoreboard.notify(resolve);
    m_checker1.notify(resolve);
    m_checker2.notify(resolve);
endfunction
```

“Old School” observer pattern
The UVM way

- UVM made it better
- Generic one-to-many connections
- Great for assembling large components
- Limited in smaller testbenches:
  - Static connections
  - Single transaction
  - Macros!
The Interface Class Way

• Removes the limitations, brings peace and happiness 😊
• Dynamic connections
• Rich communication through custom functions calls
• No macros!

• Let’s dig into an example
  – Functionality similar to a UVM analysis port
• A group of function declarations with no implementations
• A “contract” that describes the interface to the outside world

```
interface class resolve_listener;
   pure virtual function void new_resolve(arm_txn_resolve resolve);
endclass
```

“pure virtual” because interface classes don’t provide implementation
class monitor extends uvm_component;
local resolve_listener
    m_resolve_listeners[$];

function void add_listener(
    resolve_listener listener);
    m_resolve_listener.push_back(listener);
endfunction

virtual task run_phase(uvm_phase phase);
    arm_txn_resolve resolve =
        get_next_resolve();
    foreach(m_resolve_listeners[i])
        m_resolve_listeners[i].new_resolve(resolve);
endtask
endclass

A queue of listeners
Register a new listener
Notify listeners
class resolve_checker
    extends uvm_component
    implements resolve_listener;

    virtual function void
        connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        m_config.monitor.add_listener(this);
    endfunction

    virtual function void
        new_resolve(arm_txn_resolve resolve);
        if (resolve.is_abort())
            error("Aborts are not expected");
    endfunction
endclass
Building up the functionality

• Dynamic connections
• No components necessary
  – Analysis ports for sequences
  – Reactive stimulus bonanza

```plaintext
task run_sequence();
    m_config.monitor.add_listener(this);
    wait(...);
    m_config.monitor.remove_listener(this);
endtask

virtual function void new_resolve(arm_txn resolve);
    if (resolve.is_abort())
        send_flush();
endfunction
```
Passing multiple objects

• Custom function call
  – Pass whatever you like

    ```
    pure virtual function void new_resolve
      (arm_txn_uop uop, arm_txn_resolve resolve);
    ```

    ```
    virtual function void new_resolve
      (arm_txn_uop uop, arm_txn_resolve resolve);
    if (uop.is_load() && resolve.is_clean())
      check_load_data(uop);
    endfunction
    ```

Multiple objects passed for “analysis”

Complex checking of relationships, simplified
Multiple events

- Not limited to one function
- Rich interface for communicating many events
- Encapsulate complexity of tracking and matching transactions in one place

```plaintext
interface class uop_listener;
    pure virtual function void new_resolve
        (arm.txn_uop uop, arm.txn_resolve resolve);
    pure virtual function void new_commit
        (arm.txn_uop uop, arm.txn_commit commit);
    pure virtual function void new_issue(arm.txn_uop uop);
    pure virtual function void uop_flush
        (arm.txn_uop uop, flush_cause_e cause);
endclass
```
Multiple interfaces

- Implementation of multiple interfaces supported directly by the language

```cpp
class ordering_checker extends arm_checker
    implements uop_listener, ace_listener;

    virtual function void new_commit(arm_txn uop);
        if (uop.is_ordered()) m_order_q.push_back(uop);
    endfunction

    virtual function void new_ace_request(ace_txn req);
        if (!m_order_q[0].matches(req))
            error("ACE request doesn’t match oldest micro-op");
    endfunction
endclass
```

No macros! :)

The Interface Class Way

• Peace and happiness through:
  – Dynamic connections, giving us more flexibility when writing stimulus
  – Rich communication interfaces, which push the tracking and matching of transactions into common classes…
  – …which, in turn, makes our checkers simpler
  – Subscription to multiple producers are supported natively, making debug and maintenance easier
Conclusion

• Interface classes heavily used in CPU verification at ARM®

• Other examples showing off interface class flexibility in the paper:
  – Pseudo-multiple inheritance
  – Data serialization
  – Object clocking

• Will you try interface classes on your next project?
Bonus Slides
Second Example: Pseudo-Multiple-Inheritance
Pseudo-Multiple-Inheritance

- No true multiple inheritance in SV
- Interface classes give us some flexibility there

A Load-Release is both a Load and a barrier
- Where does it go without multiple inheritance?
Pseudo-Multiple-Inheritance

• An interface class can describe “barrier” behaviour

```cpp
interface class barrier;
    pure virtual function bit affects_uop
        (arm_txn_uop uop, dir_e direction);

    pure virtual function bit is_barrier_older
        (arm_txn_uop uop);
...
endclass
```

• Both DMB and Load-Release are now barriers

```cpp
class load_release extends load implements barrier;
...
class dbm extends non_addressable implements barrier;
...
Pseudo-Multiple-Inheritance

• A checker can work on micro-ops from different parts of the class tree

```plaintext
class barrier_checker;
    function void check_out_of_order_resolve
        (arm_txn_uop first, arm_txn_uop second);
        barrier bar;
        if ($cast(bar, second) &&
            bar.affects_uop(first, YOUNGER))
            error("Uop bypassed a barrier it isn’t allowed to.");
    endfunction
endclass
```

$\text{cast to barrier interface removes dependence on common base class}$
Third Example: Data Serialization
Data Serialization

- Process of dumping data of different types into a file
  - post-processing, debugging, performance analysis
- For example, dumping all interesting simulation events into a SQL database
  - Use interface class to define common fields

```cpp
interface class arm_event;
    pure virtual function int event_id();
    pure virtual function time_t event_timestamp();
    pure virtual function cpu_t event_cpu();
    pure virtual function string event_location();
    pure virtual function string event_type();
    pure virtual function string event_description();
endclass
```
Data Serialization

- Central event manager knows how to write an `arm_event` object to SQL
- For example, dumping all interesting simulation events into a SQL database
  - Use interface class to define common fields

```cpp
class event_manager;
  function void record_event(arm_event e);
    m_sql.insert(e.event_id(), e.event_timestamp(),
                  e.event_cpu(), e.event_location() ... );
  endfunction
endclass
```

Could be any base type
Data Serialization

• Let’s be even more flexible!
• True data serialization allows each class to define their own way of being recorded
• A dumpable interface class allows any class to define its own SQL table and fields

```cpp
interface class dumpable;
    pure virtual function string sql_create_table();
    pure virtual function string sql_insert();
endclass
```
Data Serialization

Implement **dumpable** interface class

Define SQL table fields

Populate the table with values

```cpp
class physical_address_txn extends txn implements dumpable;

virtual function string sql_create_table();
return "CREATE TABLE IF NOT EXISTS phys_addr_txn(id PRIMARY KEY, pa, cache_attr);
endfunction

virtual function string sql_insert();
return "INSERT INTO phys_addr_txn VALUES(m_id, m_pa.to_int(), m_cache_attr.convert2string());"
endfunction
...
endclass
```
Data Serialization

• Data recorder is completely generic
• Any object can be tested for *dumpable* interface, and recorded if it is present

```cpp
class data_recorder;
    function void dump_to_db();
        foreach (m_events[i]) begin
            dumpable d;
            if ($cast(d, m_events[i]))
                execute_sql(
                    {d.sql_create_table(),
                    d.sql_insert});
        end
    endfunction
endclass
```

Test `$cast` for *dumpable*

Generic "execute sql" call to record the object data