All Your Base Transactions Belong to Us

Jeff Vance, Alex Melikian

Verilab, Inc

Austin, TX, USA

www.verilab.com

ABSTRACT

Combining diverse transaction classes into an integrated testbench, while common practice, typically produces a cumbersome and inadequate setup for system-level verification. UVM sequence item classes from different VIPs typically have different conventions for storing, printing, constraining, comparing, and covering behavior. Additionally, these classes focus on generic bus aspects and lack context for system coverage. This paper solves these problems using the mixin design pattern to supplement all transaction classes with a common set of metadata and functions. A mixin solution can be applied to any project with minimal effort to achieve better visibility of system-wide dataflow. This setup enables significantly faster debug time, higher quality bug reports, and increases the efficiency of the verification team. Combining a mixin with an interface class results in a powerful and extensible tool for verification. We show how to define more accurate transaction coverage with extensible covergroups that are reusable with different transaction types. Scoreboard complexity is reduced with virtual mixin utility methods, and stimulus is managed with shared constraints for all interface traffic. Finally, we show this solution can be retrofitted to any legacy UVM testbench using standard factory overrides.

Table of Contents

Table of Figures

1. Introduction

Typical SystemVerilog/UVM testbench environments have to manage a wide variety of transaction classes for driving, monitoring, and reporting stimulus. Transactions usually come from both third party and internal VIP as well as project-specific transactions that are reused from block-level testbenches. All these transactions often model different protocols and have different ways of defining variables, methods, and constraints.

Mixing a variety of transaction types in a single testbench creates many challenges that hinder verification productivity and limit the quality of a system verification in many ways.

- Individual transaction classes lack system context such as datapath source/destination points, statistical information such as transaction counts, the design configuration at the time the transaction occurred, and historic information that allows you to link originating transactions to multiple derived destination transactions.
- Debug is often difficult due to inconsistent printing formats between each transaction class.
- It is hard to collect meaningful functional coverage without a system context.

From a system-level verification perspective, we need consistent and standardized control, reporting, and coverage of system-wide traffic despite dealing with diverse and differing transactions. In addition, the reuse of existing VIP and block-level environments cannot be compromised.

This paper shows how any UVM testbench can take diverse transaction classes and standardize them with common features using the mixin design pattern. We then show how this technique can benefit the primary objectives of a testbench: debug, coverage, stimulus, and scoreboarding. Next we show how to make these solutions even more powerful by having the mixin class implement an interface class. This combination of techniques along with standard UVM factory overrides gives us the ultimate tool for taking ownership of all transactions in any UVM testbench and managing them for our project-specific needs.

2. The Mixin Concept

The goal of a mixin is to share code between multiple classes that do not originate from a common base ([1], [2]). By adding common features to multiple classes, we can treat them the same way. However, achieving this with the standard class inheritance scheme creates the problem of duplicating the same code for each extended class (Figure 1) since they do not share a common base. Not only is this inefficient, but code maintenance across each class becomes difficult.

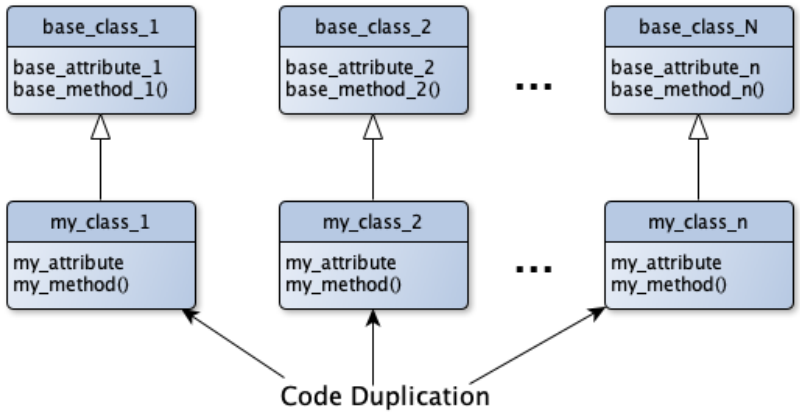

Figure 1: Adding the same code to many extended classes is unideal.

The mixin pattern enables the ideal solution of capturing common functions in a separate class to be exported to all extended classes (see Figure 2). This is similar[1] to a multiple inheritance scheme since each extended class inherits aspects from two different base class types: the original base class, along with the class exporting common functions.

The other programming capability that the mixin design pattern offers is that of "aspect" programming, where a consistent method of applying cross-cutting functionality can be propagated to existing classes. However, it should be noted that the mixin pattern only emulates and does not perfectly replicate true aspect oriented programming (AOP). There are limitations in orthogonality and flexibility with how the mixin pattern can overlay new functionality into existing classes compared to a true AOP language. In other words, the mixin pattern has an "all or nothing" characteristic, whereas a genuine AOP offers more malleability in how code is shared.

3. Mixin Implementation

The mixin is a parameterized class that extends the same type it is parameterized by. It is declared using the format: **"mixin_class#(type T) extends T"**, which allows a mixin to extend any other class. Figure 2 shows multiple classes being extended and "mixed" with the common mixin class features. Each extended class variant needs to be declared with another version of the mixin using a different parameter type. For example: **"typedef mixin_class#(base_class_1) my_class_1;"**.

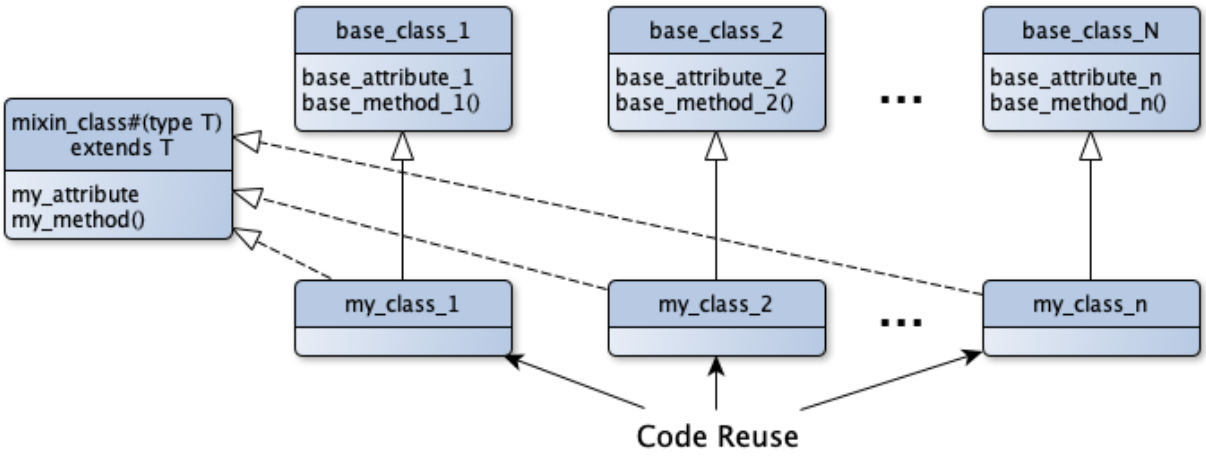

Figure 2: Mixin inheritance allows for code reuse across many extended types.

3.1 Mixins Applied to Transactions

SystemVerilog does not support true multiple inheritance. However, the mixin is a proven design pattern that can solve verification problems requiring classes to share aspects of other classes [3]. The UVM class library already uses mixins to implement several features, such as TLM ports.

It is common practice for testbench and VIP developers to provide sequence item base classes for bus protocols by extending the uvm_sequence_item class [8]. These VIP classes model the behavior of the

[1] Mixin inheritance is not true multiple inheritance since you cannot instantiate or declare a generic handle of the mixin class directly. Users can only use sub-classes of a mixin. We show how to solve this problem in section 6. using an interface class.

protocol by declaring transactions fields such as address, payload, tags, checksums, and parity. They also typically include randomization constraints and protocol-specific methods for managing fields. The class often also provides protocol-specific implementations of various uvm_object virtual methods such as **do_compare()**, **do_print()**, and **convert2string()**. Every testbench and VIP developer is free to choose how to define transactions, therefore it is natural for there to be differences in naming conventions, data structures, and method details.

4. Problems with Different Transaction Types

While it is very common for a testbench to include multiple transaction types, the differences in how they are defined often hurts verification productivity when verifying system-wide traffic across different interfaces. Before we explore how the mixin pattern can help, we will explore the individual problem areas brought on by the differences in transaction types: debugging, coverage, checking, and stimulus.

4.1 Logging and Debug Problems

Data is often represented differently in each transaction class since each protocol has different rules in how data is processed. Figure 3 shows an example of two hypothetical VIP sequence items, "vip-x-item" and "vip-y-item", that manage the same data in different ways. Although these are hypothetical protocols, this example highlights typical modern protocol characteristics along with their differences listed below:

- **vip_x_item** collects data using an array of bytes. When this data is propagated to the y-interface, it is captured as an array of 32-bit words.
- **vip_x_item** encodes the direction (read/write) of the transfer with a single bit (0/1). However, the vip_y_item uses a VIP-defined enumerated type of Y_READ/Y_WRITE.
- **vip_x_item** uses a single address variable (addr) for all types of transactions. However vip_y_item has two different address variables: addr is used only for single transfers, while start_addr is only used for bursts.
- Both types have protocol-specific fields that are not applicable to the other (id fields).

```
class vip_x_item extends uvm_sequence_item;
  rand bit         r_w;             // 1 is write, 0 is read
  rand bit[31:0]   addr;
  rand bit[7:0]    data_bytes[];    // Data is array of bytes
  rand bit[4:0]    length;          // Length is number of bytes
  rand bit[3:0]    x_id;            // Protocol-specific ID field
  …
-------------------------------------------------------------------------
class vip_y_item extends uvm_sequence_item;
  rand y_dir_enum  dir;             // VIP enum type: Y_READ, Y_WRITE
  rand bit[31:0]   addr;            // Only used for single transfers
  rand bit[31:0]   start_addr;      // Only used for bursts > 1
  rand bit[31:0]   data[];          // Data is array of words
  rand bit[4:0]    len;             // Length is number of words
  rand bit[3:0]    y_id;            // Protocol-specific ID field
  …
```

Figure 3: Transaction types with different ways of managing the same traffic.

Logging and debugging the flow of system data between interfaces using these transactions has many consequences on verification productivity:

- Parsing logs to debug dataflow becomes non-trivial due to inconsistencies in field names and data representation. You can't easily trace a data word pattern that appears as individual bytes somewhere else.
- Conditional fields like **start_addr** get printed, even when they aren't relevant (e.g., for a single transaction). Although this may seem like a trivial judgement when glancing at a single transaction in the log, this negatively impacts coverage collection and scoreboarding which must determine which field to use.
- Transactions often print a huge number of fields from the base class that users don't care about. These fields are necessary for debugging protocol details but are excessive noise to someone focusing on debugging system traffic.

4.2 System Coverage Problems

Defining system functional coverage using many different transaction types is burdensome. We can't use pre-packaged VIP coverage since it lacks system context about an observed transaction. Examples of system context include design configuration at the time a transaction occurred and source/destination paths. We must define our own covergroups from a system perspective, but we can't reuse and cross system covergroups with VIP or block-level coverage. Since transaction fields may be defined differently, we are forced to create new specialized covergroups for every type, even though we often want to cover the same information. For example, **vip_x_item** and **vip_y_item** both track address, command, and length information. To cover these in a system context, we have to create new covergroups for each type to sample the appropriate fields, then cross them with system observations. Conditional fields like **start_addr/addr** further complicate things since we must only sample the appropriate variable.

4.3 Checking Problems

A typical operation in many testbenches is data integrity checking across a DUT. This would be where data contained in one incoming transaction is transferred, filtered or processed before being output to another interface. Often, the core of the data integrity checking comes down to comparing an expected array of bits or bytes extracted from the incoming transaction item, to those observed on the transaction output by the DUT.

Although the nature of the transactions may differ, this comparison operation between two data sets remain the same. Unfortunately, similar, if not near identical scoreboarding code tends to be repeated for each input-output datapath combination, rather than be sourced from a single location, applicable to any eligible transaction item.

4.4 Stimulus Problems

We often want to control stimulus of different interfaces in consistent ways. In a system-level testbench, we're mostly concerned with controlling command types (write/read), address, and payload. The most common solution in UVM is to implement virtual sequences to provide a consistent API to randomize and control these aspects for each interface, despite having different member variables and protocol-specific constraints. However, one disadvantage of this approach is that we may end up with many layers of encapsulation that make it more challenging to add new constraints on sequence items from tests. Some projects may have an existing sequence library that does not provide control knobs necessary for new tests and can't be easily modified. Many of these problems would be ideally solved by adding randomization control directly to each sequence item, without

impacting existing sequences. But such a solution needs to be common across all transaction types.

5. Solutions Mixins Offer to Transactions

This section highlights how mixins can solve all the problems highlighted in the previous section. We first show how to apply a mixin to add consistency to all UVM transactions in a testbench. An example datapath design and testbench demonstrates the benefits of this solution. We then show how to further capitalize on the mixin solution with improved scoreboarding, functional coverage, and stimulus. These solutions are applicable to all transaction item classes, including those using legacy or third-party source code.

5.1 Adding Consistency to Transaction Types

Mixin inheritance allows us to add consistency to all transaction types in a system so we can standardize logging, coverage collection, and checking across all interfaces. We can add mixins to any UVM testbench in three steps described below and shown in Figure 4 (more detailed code examples are shown in later sections).

- **Step 1**: **Define a generic system transaction.** This is the mixin template class.
  - Include common member variables for all types
  - Include common method implementations for all types
  - Include virtual method prototypes for each type to customize

- **Step 2: Create typedef base class declarations for each mixed transaction type.** This base type will have the attributes from both the generic system transaction and VIP transaction. Note that we generally won't use this class directly and instead use extended versions of it.

- **Step 3: Extend each base "mixed transaction" with type-specific attributes**
  - Include unique member variables for each type
  - Include unique methods for this type
  - Implement base class virtual methods with type-specific details

This example template code is a standard approach that can be applied to any UVM testbench. Figure 5 illustrates the relationship between the UVM base classes, VIP classes, and extended system transactions. One way of organizing this code is to keep the **sys_item** class in its own file, **sys_item.sv**, and putting each of the typedef declarations at the top of each extended transaction class file (e.g., **sys_x_item.sv**).

```
//-----------------------------------------------------------------
// STEP 1: Define generic system transaction as a mixin template
//-----------------------------------------------------------------
class sys_item#(type T=uvm_object) extends T;
  `uvm_object_param_utils(sys_item#(T))
  // Define common variables to be shared
  // Define common methods to be shared
  // Define virtual methods to be extended per type
endclass
//-----------------------------------------------------------------
// STEP 2: typedef mixin base classes for each transaction type
//-----------------------------------------------------------------
typedef sys_item #(vip_x_item) sys_x_item_base;
typedef sys_item #(vip_y_item) sys_y_item_base;
...
typedef sys_item #(vip_n_item) sys_n_item_base;
//-----------------------------------------------------------------
// STEP 3: extend mixin base classes with type-specific details
//-----------------------------------------------------------------
class sys_x_item extends sys_x_item_base
  //... Custom variables and functions specific to x_item type
endclass
class sys_y_item extends sys_y_item_base
  //... Custom variables and functions specific to y_item type
endclass
```

Figure 4: A system transaction mixin solution for any UVM testbench

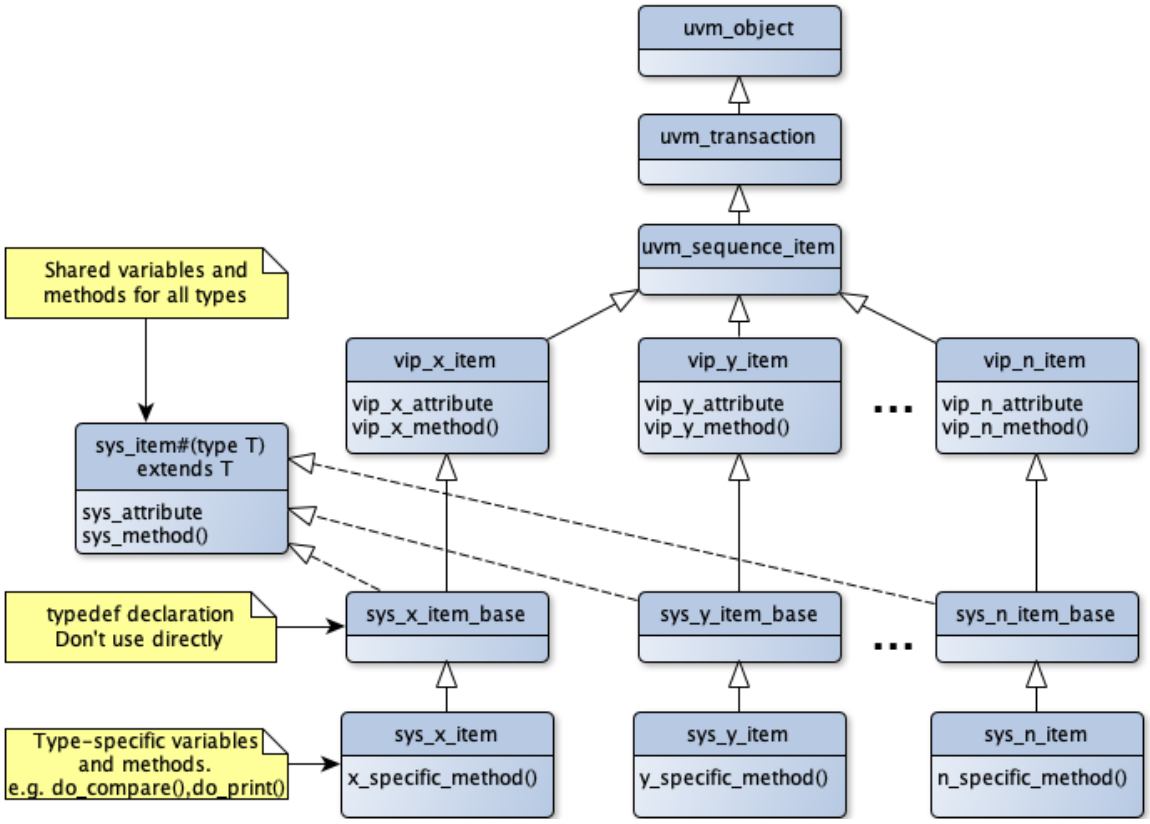

Figure 5: A generic system transaction adds features to all VIP transactions.

5.2 Datapath Example

This section shows how to apply the mixin technique to solve common datapath verification challenges. Figure 6 shows a typical situation of several master ports that must send traffic to several slave ports. The design under test (DUT) will typically have an interconnect with rules of which master-to-slave paths are legal and valid address ranges for each slave. However, not all masters and slaves use the same interface protocol. In this example, three different protocols are used (shown as x, y, and z interfaces). We have existing VIP for each protocol giving us standard class implementations for a UVM Verification Component (UVC): agents (consisting of sequencers, monitors, and drivers) and sequence items.

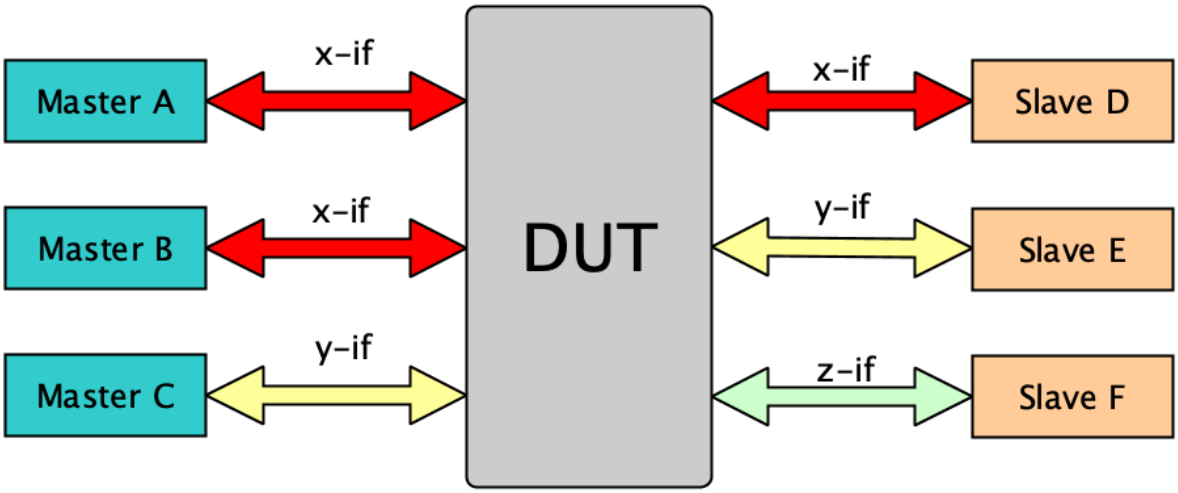

Figure 6: A typical Design has a mix of interface protocols to route traffic.

The traditional testbench approach would be to connect the X-UVC, Y-UVC, and Z-UVC instances to the appropriate interfaces and use them as-is. Monitors will passively observe traffic according to the protocol rules, record transactions in a sequence item class, and publish these transactions for scoreboarding and coverage. However, as highlighted in section 4.1 , each UVC will manage data in different ways using specialized sequence item classes.

5.2.1 Generic Transaction Features

We can improve the visibility and tracking of system traffic using the generic system transaction class previously shown in Figure 4. Instead of using the VIP classes as-is, we can extend them and mix in the generic features we want for managing all transactions. Highlighted below are things we can typically standardize across all transaction types: system metadata variables, **convert2string()** implementation, virtual function prototypes, and UVM policy class configurations (e.g., **uvm_printer**).

A system testbench can add related metadata fields as member variables common to all interfaces to capture system information about the transaction. For example, Figure 7 shows a system transaction class that adds enumerated types for the master that initiated the transaction, the slave destination,

and the command[2] (read/write). Additional metadata can include:

- **transaction counts** - easily track the flow of transactions between interfaces with unique count numbers. This may require both master-count and slave-count in situations where a large burst on one interface is split into multiple transactions at the destination.
- **timestamps** - record simulation time of start/end times of bursts. This can be used to collect performance metrics for dataflow through all paths.
- **design configuration** - record configuration at the time of the transaction, which may come from a configuration object or a register model mirror. For some designs this can be important for coverage and scoreboarding (especially if configurations can change throughout a test).
- **reference model objects** - capture DUT-specific information such as memory maps and valid datapaths. These can be used to derive certain metadata fields, such as which slave corresponds to the specified address.

```
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;    //derive slave enum from addr mem map
       sys_path_model    refmodel; //ref object to get slave from address
  …
  int  master_count;     // Transaction number from this master
  int  slave_count;      // Transaction number for this slave
  time burst_start_time; // collect for performance metrics
  time burst_end_time;

  virtual function sys_slave_enum  get_slave();       // metadata accessor
    return refmodel.get_slave_from_addr(get_addr());
  endfunction
```

Figure 7: This system item mixin defines metadata and methods for all transaction types.

Note that some metadata fields should be declared random, while others shouldn't be. For example, a constraint model that randomizes the transaction can directly derive fields like cmd, master, and slave using constraints (see section 5.5). However, fields such as counts, timestamps, and configuration should never be randomized and must be set by passive components such as monitors and scoreboards. Also note that we can't pass an address variable directly to **get_slave_from_addr()** since this variable name will be different for each transaction type. We must use an accessor method to get the address (see next section).

The **convert2string()** method is a **uvm_object** virtual function intended to print an object as a concise string (typically in a single-line). This is useful to get a short summary of a

[2] While each protocol typically already encodes the read/write command in the base transaction, we've found much more flexibility in creating a standard enumerated command type. This allows for standardized logs, randomization constraints, coverage, and checks of these commands for all interfaces.

transaction without excessive details.[3] From a system perspective, we are mostly interested in tracking the flow of traffic between interfaces, without the protocol details. Therefore it is natural to standardize this function to be the same for all transactions. Figure 8 shows an example **convert2string()** implementation that will print all transactions in a uniform format.

```
virtual function string convert2string();

  //Only print the header if it hasn't already been printed this time-step.
  if(($realtime() == 0) || ($realtime() > last_print_time)) begin
    convert2string = {"\n[TRLOG]", get_fields_header(),
                      "\n[TRLOG]", get_fields_string()};
     last_print_time = $realtime();
  end
  else convert2string = {"\n[TRLOG]", get_fields_string()};
endfunction

//Get all generic fields to print into a single line string.
virtual function string get_fields_string();
  string field_strings[];
  string all_fields;
  string format;

  field_strings = '{ get_master_string(),
                     get_slave_string(),
                     cmd.name(),
                     $sformatf("%0h", get_addr()),
                     $sformatf("%0d", get_num_bytes()),
                     get_data_string()
                   };

  // Generate a format string for each field using array of width values
  foreach(field_widths[i]) begin
    format = $sformatf(" %%%0ds |", field_widths[i]);
    all_fields = {all_fields, $sformatf(format, field_strings[i])};
  end

  // Add custom fields for each transaction type at end of string
  all_fields = {all_fields, get_custom_fields()};
  return all_fields;
endfunction
```

Figure 8: Standardized convert2string() prints all transaction types with a common format.

In this example, we enforce standardization of fields printed for all transactions, but we still allow some flexibility to print custom fields that are unique to each transaction type by adding these after the common fields. A virtual function **get_custom_fields()** is optionally implemented by each extended transaction type. After running a test, we can parse the entire log for a unique label (e.g.,

[3] This is opposed to **uvm_object** functions **print()** and **sprint()** which can generate massive tables of transaction details. In many cases, someone debugging a simulation won't care about these details. Excessive details sometimes create distractions that can make it harder to debug problems.

using grep) to track the flow of traffic on each interface, as shown in Figure 9.

```
[TRLOG]=MASTER=|=SLAVE==|====CMD====|===ADDR===|=LEN=|=====DATA=====|====================
[TRLOG]  MST_A |  SLV_E |  SYS_READ |   1e1a1  |  26 |       ------ | x_id: 1 | x_tag: e |
[TRLOG]  MST_B |  SLV_E |  SYS_READ |   180ee  |  28 |       ------ | x_id: 6 | x_tag: 4 |
[TRLOG]  MST_A |  SLV_F | SYS_WRITE |   2942c  |   4 |     debf8a54 | x_id: f | x_tag: d |
[TRLOG]  MST_C |  SLV_D | SYS_WRITE |   5bea   |  12 |  2f135d3c,.. | y_id: 5 |          |
[TRLOG]  MST_B |  SLV_F | SYS_WRITE |   25f85  |   8 |  c5096c73,.. | x_id: 7 | x_tag: 2 |
[TRLOG]  MST_C |  SLV_E |  SYS_READ |   19e22  |  40 |       ------ | y_id: c |          |
[TRLOG]  MST_C |  SLV_E | SYS_WRITE |   1025f  |   8 |  afbf5954,.. | y_id: 8 |          |
```

Figure 9: Searching for "TRLOG" in the simulation log reports the flow of all transactions in a system for easy debug.

Another benefit of the system item mixin class is we can standardize virtual function prototypes for all transactions. For example, the **convert2string()** implementation in Figure 8 requires *virtual accessor functions* to get the fields to print from each transaction type. This is necessary since the variable names and types of each transaction are different. Virtual function prototypes gives us a standard API for **convert2string()** that will work for all transactions. These accessor methods must be customized in the extended transaction classes (see section 5.2.2). Other virtual functions can be added for standardizing scoreboarding and coverage (see sections 5.3 5.4). In many cases it is useful to provide a default print string such as "---" or "NA" to indicate a field is unset or not-applicable. This helps avoid printing misleading information in the log that may be misinterpreted. In other cases, it is better to force users to redefine these functions in the extended class by either declaring these as **pure virtual**, or by issuing a **`uvm_fatal** message by default[4].

```
virtual function string get_master();
 return "---";  // indicates field is unset or not applicable
endfunction


virtual function string get_slave();
 return "---";  // indicates field is unset or not applicable
endfunction

virtual function bit[31:0] get_addr();
  `uvm_fatal(get_type_name(),
             "Must redefine get_addr() in extended class")
  return 0;
endfunction

virtual function string get_custom_fields();
 return "";   // default to printing an empty string
endfunction
```

Figure 10: The mixin defines virtual methods common to all transaction types.

There are other UVM features that can be useful to standardize across transactions by adding them to the system transaction mixin class. For example, a **uvm_printer** object is used by **print()/sprint()** functions to display detailed tables and trees of transactions: rather than using the default

[4] See section 6.1 for an explanation why default fatal messages are often necessary.

uvm_printer, we can assign a custom printer configuration inside the mixin for more uniform control of how transactions are printed [8]. This can be done independently of the rest of the testbench code which can continue using the default **uvm_printer**.

5.2.2 Extended Transaction Features

To fully take advantage of the mixin solution, we must define extended classes for each transaction type. These will inherit all the VIP features, the generic system transaction features, and allow us to add unique features for each transaction type. Figure 11 shows the class hierarchy and relationships of VIP sequence items, the system item mixin template, and extended system items.

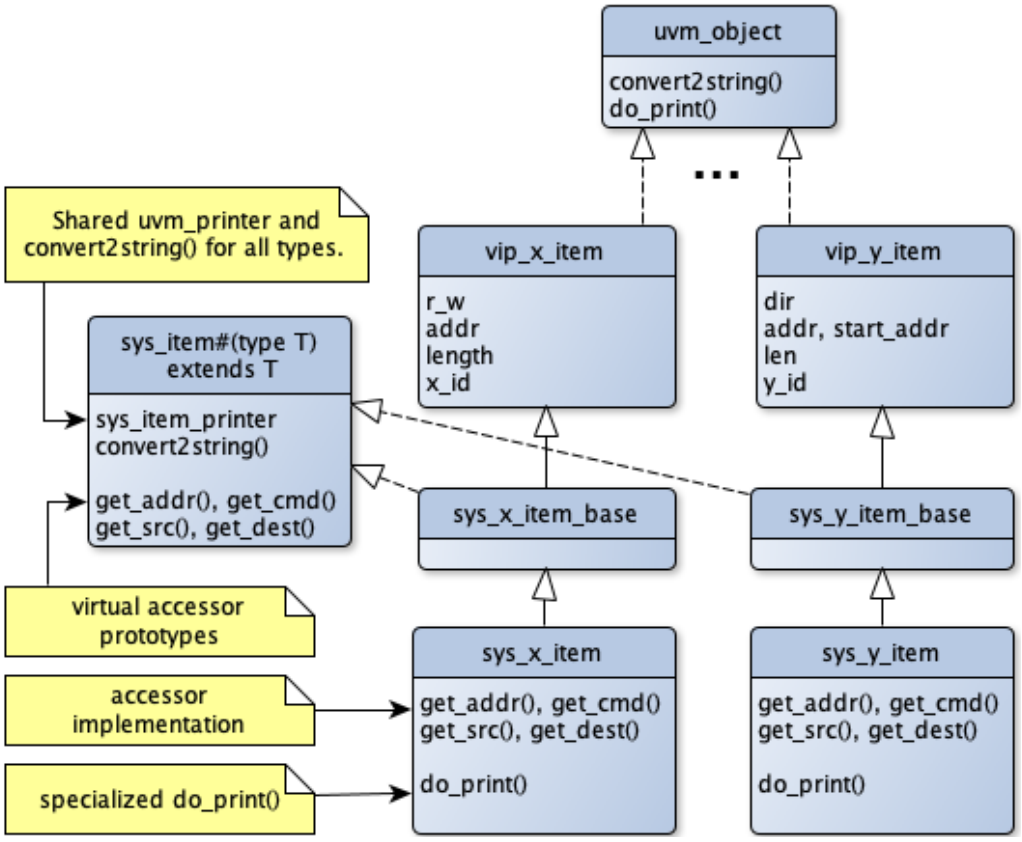

Figure 11: The sys_item mixin template defines standard transaction features while extended items add customizations.

There are four types of specialized features we can add:

- customizations of **uvm_object** virtual methods
- introspection methods
- utility/helper methods
- type-specific metadata variables

Each transaction type typically requires implementation details that cannot be standardized. For example, the **do_compare()** method customizes how fields should be compared and is invoked by the uvm_object base class **compare()** function. Similarly, the **do_print()** function customizes which fields to print (and how to format them) when calling **print()/sprint()**. While it is useful to standardize **convert2string()** for all transactions, we still need **do_print()** to be specialized for each

type to see all the protocol details in a log for each interface.[5]

5.3 Supporting Scoreboarding with Mixins

In addition to enhanced system level debugging support, the mixin paradigm can also help considerably in moving actual scoreboarding operations into the various transactions. Data integrity checking is one of the most typical scoreboarding activities of any test bench, where the comparison of transaction class instances are centrally involved. These common traits create strong potential to add scoreboarding processing into the mixin pattern implementation, producing once again the benefits of functionality propagation.

To realize the benefits, the biggest challenge to overcome is establishing a scheme of processing data integrity which can be applied globally across any transaction item class. Fortunately, it can be safely assumed that data transactions at a DUT interface will always be in the form of bits or bytes. Therefore, we propose to distill or process transaction items of interest down to an array of bits or bytes. In other words, to 'pack' the transaction item given a criteria representing how the transaction would be output from a given DUT bus interface.

It is with this 'packing' approach that the solution can be devised using the following three methods:

```
virtual function void pack_for_sb(ref bit[7:0] packed_bytes[],
                                  ref int num_bytes);

virtual function string disclose_pack_member_for_sb(int pack_bit_position);

function bit compare_packed_data_for_sb(input bit [7:0] external_pack_bytes,
                                        input int external_num_bytes,
                                        output int disagree_bit_position);
```

Figure 12: Utility and built-in scoreboarding methods in the mixin

Note that the **pack_for_sb( )** and **disclose_pack_member_for_sb( )** methods are virtual and are intended to be implemented by the transaction class that extends the mixin. Whereas the **compare_packed_data_for_sb( )** will have implemented code in the mixin, ready to be used for any item.

The **pack_for_sb( )** method will be extended to implement the packing of the member items of interest for a DUT output interface. In the example of a packet transaction, this method can be used to extract out the payload portion of the packet that is expected to be routed forward out of another interface. Whereas the **disclose_pack_member_for_sb()** is intended to be a query method, where given a bit position of the packed bit/bytes, a string representing the field member is disclosed. While this method is not essential to data integrity checking, it helps testbench users debug issues.

Once the straightforward virtual methods are implemented, they are meant to be used as shown in Figure 13 in a testbench scoreboard.

[5] Tip: we can take advantage of UVM verbosity settings to manage both ways of printing transactions. Call **convert2string()** on lower verbosities and **print()/sprint()** on higher verbosities.

```
x_item.pack_for_sb(sb_pack_bytes, sb_num_bytes);
if (!y_item.compare_packed_data_for_sb(sb_pack_bytes, sb_num_bytes,
                                       sb_disagree_bit_pos))
  `uvm_info("", $sformatf("Disagreement: X item %s and Y item %s",
                  x_item.disclose_pack_member_for_sb(sb_disagree_bit_pos),
                  y_item.disclose_pack_member_for_sb(sb_disagree_bit_pos)),
                  UVM_NONE)
else
  `uvm_info("", "Match between X item and Y item", UVM_NONE)
```

Figure 13: Example of a scoreboard using mixin utility methods for checking

This mixin approach enables the scoreboard to leverage off **compare_packed_data_for_sb( )** to execute the data integrity check, as well as **disclose_pack_member_for_sb( )** for disclosing additional information when debugging is needed.

The sample code in Figure 13 shows only one packing method for the sake of simplicity. In the case that a transaction can be routed to various interfaces, a packing and filtering method for each interface destination can be devised. Therefore, all scoreboard routing, filtering or data integrity checking methods can be represented and distributed by the mixin.

Note that scoreboard checking of other fields (e.g., address, length) can be checked using the standard UVM **compare()** method with a customized **do_compare()**, the same as a typical UVM testbench.

5.4 Supporting Coverage with Mixins

Mixins can help solve several major challenges with functional coverage of transactions. First, the metadata provided by a mixin class can add context to provide more accurate and meaningful coverage. Second, our mixin solution can help encapsulate details and provide introspection[6] of aspects to cover, keeping covergroup definitions lean and reusable. Finally, we can define generic covergroups that are reusable for all transaction types, while customizing each covergroup instance with unique bins using functions in the mixin class. This section describes each of these solutions in detail.

5.4.1 Add Context and Accuracy

One of the major challenges of functional coverage is ensuring our results indicate we truly reached the verification intent of a feature. It is very common for coverage to be implemented with inaccuracies that go unnoticed and give a false indication of what was truly verified [4]. Coverage sampled at the wrong time or in the wrong context generates false coverage reports. For example, it would be incorrect to sample coverage of a transaction in all design configurations if it only has special meaning in certain configurations. It would also be wrong to sample if it does not actually reach its destination because it is filtered out or dropped due to congestion. Additionally, even coverage that's sampled at the right time may not be precise enough if it doesn't include context in the sampling data (e.g., we saw a write on an interface when it completed at the destination, but we didn't record where the transaction originated from).

Coverage using base transaction classes is typically insufficient for system coverage since they only

[6] Introspection in this context means we allow a transaction to self-examine its own properties. For example, it can detect what type of packet it is if this requires a higher-level interpretation of various fields.

contain interface protocol details. Transactions are generic for any project, therefore there is nothing in the class that places the transaction in context of what is happening in the system. This usually puts the burden of coverage on other testbench components to figure out the context.

The mixin technique gives us another option for solving all these coverage challenges. This approach manages coverage context directly inside the transaction. As described in section 5.2.1 , extended transactions can have project-specific metadata that is standard across all transaction types. These can be directly sampled for coverage and crossed with protocol-specific fields. Figure 14 shows an example for transactions sent to "Slave E" which has a "y-interface." Recall from Figure 3 that the **vip_y_item** has two different address fields, depending on whether the transaction is single or a burst. The accessor method ensures we always sample the correct address field for all cases.

```
covergroup slv_e_cg with function sample(sys_y_item  tr);
  e_master_cp: coverpoint tr.master;     //sample mixin metadata field
  e_addr_cp:   coverpoint tr.get_addr(); //method gets proper addr field

  // conditionally sample vip base class field on system context
  e_yid_cp:   coverpoint tr.y_id iff (sys_yif_cfg != IGNORE_ID)

  //cross system context with protocol-specific details
  e_master_yid : cross e_master_cp, e_yid_cp;
endgroup
```

Figure 14: Coverage that relies on both system metadata fields and protocol-specific fields

This example highlights several advantages:

- **Add system context to protocol coverage:** The master that sourced the transactions can be sampled and crossed with protocol-details like **y_id**.
- **Improved coverage accuracy:** fields are sampled only in the context that they are truly relevant. For example, **y_id** is only sampled when the DUT isn't configured to ignore them.
- **Manage complexity of sampling correct fields:** Rather than code complex covergroups that cover different fields under different conditions, we can simply call an accessor function that returns the appropriate value to sample for us[7]. It is generally much easier to manage these details inside a function than in a covergroup. Additionally, these accessor functions can be declared virtual so they are more extensible and can be reused for printing logs and scoreboarding.

[7] Recall from Figure 7 that the mixin can contain a reference model object for accessor methods to derive fields for coverage.

5.4.2 Provide Introspection for Coverage

Another way mixins can help functional coverage is by encapsulating the details of how to interpret a transaction inside the extended transaction class and provide introspection methods for coverage sampling. For example, it is often insufficient to simply cover all the variables declared in a transaction. Sometimes we need to interpret certain bit fields to understand what we're sampling. In a system context, certain transaction commands may be invalid for certain address ranges. Figure 15 shows an example of a covergroup that takes advantage of mixin methods. All sampling is performed on the return values of methods called in the transaction. Because the details of how to interpret the transaction are decoupled from the covergroup definition, it becomes much leaner, easier to understand, and can be reused for different transaction types (see section 6.3).

```
class sys_x_item extends sys_x_item_base
  // Extract and interpret portions of vip bass class fields
  virtual function bit[23:0] get_frame_header();
    return {data_bytes[2], data_bytes[3], data_bytes[4]};
  endfunction
...
endclass
-------------------------------------------------------------------------
covergroup mst_a_cg with function sample(sys_x_item  tr);
  header_cp: coverpoint tr.get_frame_header();
  ...
```

Figure 15: Coverage relies on transaction method to extract fields to cover

5.5 Stimulus with Mixins

It's possible to use common features in the mixin class to standardize stimulus control across all transaction types. As discussed in section 4.4 , there are some problems that can be more easily solved by adding stimulus control directly to transactions. One way of doing this is placing additional constraints inside a shared object, such as the reference model object previously mentioned in Figure 7. This reference model can capture valid address ranges per slave and valid paths from each master to the slaves. Constraints in the reference model object ensure that any randomized combination of master, slave, and address are valid (Figure 16).

```
class sys_datapath_model extends uvm_object;
  rand sys_master_enum  master_e;
  rand sys_slave_enum   slave_e;
  rand bit[31:0]        addr;

  // memory map
  bit[31:0] addr_map_start[sys_slave_enum];
  bit[31:0] addr_map_end  [sys_slave_enum];

  // valid paths
  sys_slave_enum  valid_paths_by_master[sys_master_enum][] = '{
    MST_A:  {SLV_D, SLV_E, SLV_F},
    MST_B:  {SLV_D, SLV_F},
    MST_C:  {SLV_E, SLV_F}
  };
  virtual function void initialize_mem_map();
    addr_map_start[SLV_D] = 32'h0;
    addr_map_end  [SLV_D] = 32'h0000_FFFF;
    addr_map_start[SLV_E] = 32'h0001_0000;
    addr_map_end  [SLV_E] = 32'h0001_FFFF;
      ...
  endfunction
  virtual function sys_slave_enum  get_slave_from_addr(bit[31:0] addr);
    foreach(addr_map_start[s]) begin
      if(addr inside {[addr_map_start[s]:addr_map_end[s]]}) return s;
    end
    return SLV_UNDEF;
  endfunction

  constraint valid_paths_c{      //accessed slave must be valid for master
    foreach(valid_paths_by_master[m]){
      (master_e == m) -> (slave_e inside {valid_paths_by_master[m] });
    }}
  constraint mem_map_c{          //addr must map to corresponding slave enum
    foreach(addr_map_start[s]){
      (slave_e == s)<-> (addr inside {[addr_map_start[s]:addr_map_end[s]]});
    }}
```

Figure 16: Example datapath reference model to constrain master, slave, and address for all transaction types

To consistently apply these reference model constraints across all transaction types, we simply need to declare a random instance of the datapath model inside the mixin and constrain the transaction fields to be influenced by this model (see Figure 17). Since the object is declared *rand*, all transaction constraints and reference model constraints are solved concurrently.[8] Note that constraints on type-specific variables must be defined in the extended transactions (e.g., **sys_x_item)**, while constraints shared by all types should be in the **sys_item** mixin class.

[8] Be careful not to create conflicts with constraints in sequences that are randomizing the transaction. This may cause constraint solver failures. This approach might not work well in some testbenches that have tightly controlled constraints from sequences (without disabling the sequence constraints).

```
class sys_item#(type T=uvm_object) extends T;
  rand sys_path_model    refmodel;  //common to all transaction types
  constraint valid_paths{
    master == refmodel.master_e;     //constrain generic metadata fields
    slave  == refmodel.slave_e; }
-------------------------------------------------------------------------
class sys_x_item extends sys_x_item_base;

  //constrain type-specific fields
  constraint  addr_path_c{ addr == refmodel.addr }
```

Figure 17: Adding shared reference model constraints to a mixin to control all transaction types.

6. Enhance Mixin with an Interface Class

The mixin solution shown so far allows for multiple transaction types to share common attributes and methods so we can treat them in a standard way for logging, scoreboarding, coverage, and stimulus. One problem with the solution so far is we cannot treat each transaction instance generically using a common object handle type. While we can use **uvm_object** as a generic handle to manage all the different types, we cannot call our mixin methods from a **uvm_object** handle since they are not defined in the **uvm_object** base class. To call our methods, we must cast the **uvm_object** to the extended type, but this is often not possible for two reasons. First, we often don't know what type to cast it to. For example, we may collect a queue of **uvm_objects** that are a collection of various extended types. We can't tell from the **uvm_object** base class what extended type this object really is. While it might be possible to write code to figure out what type to cast it to (see Figure 18), we are then faced with the second problem: Code written to explicitly manage every type becomes more complex, requires code duplication, and is not easily maintainable. We must continue to maintain separate if/case statements for all versions of the class. We're also then required to continue updating this code to handle future classes we wish to support.

```
uvm_object q[$];
…
if($cast(x_tr, q[0]) == 1) begin
  master = x_tr.master;
  x_tr.other_method();
  …
end
else if($cast(y_tr, q[0]) == 1) begin
  master = y_tr.master;
  y_tr.other_method();
end
else if(...) //repeat above code for every extended type
```

Figure 18: This attempt to determine an object type requires ongoing maintenance and is not extensible.

What we need is a single generic class handle for all transaction types that gives us direct access to mixin features. It isn't possible to declare a base handle using the mixin class since it is a parameterized class. A parameterized class is just a template for other class types to be defined and it is impossible for all templated versions to be known in advance in order to use as handles. However,

one way the mixin can provide a generic handle is by having it implement an **interface class**[9].

6.1 Using Interface Classes

The interface class not only encapsulates a common set of behaviors for other classes to treat in a generic fashion, but in this case it can also serve as a class handle, which a template class by itself cannot provide. Figure 19 shows how some of the critical, but un-implemented mixin methods can be declared in the interface class. The **sys_item** mixin class definition now implements this interface class and provides the same base implementation shown previously.

```
interface class sys_item_cif;
  pure virtual function bit[31:0] get_addr();
endclass

class sys_item#(type T=uvm_object) extends T implements sys_item_cif;
  virtual function bit[31:0] get_addr();
    `uvm_fatal(get_type_name(),
            "Must redefine get_addr() in extended class")
    return 0;
  endfunction
endclass
```

Figure 19: Declaring mixin methods in an interface class

The mixin class must implement every pure virtual method from the interface class to avoid compile errors. In many cases these implementations can be empty functions left for extended classes to optionally implement[10]. In other cases, a default fatal message is required to mandate an extended implementation. This scheme ultimately makes it easier for users since they are given a complete list of all methods that must be implemented or can choose to redefine for custom behavior.

When applying this technique to transactions, we have the flexibility to manage transaction objects using either **uvm_object** handles, or **sys_item_cif** handles. Given a **sys_item_cif** handle, we can only call the pure virtual methods declared in that interface class. Even though all UVM transactions extend **uvm_object**, we can't call methods like **convert2string()** or **get_name()** without first casting it to **uvm_object** (using **$cast()**). However, it is also possible to expose **uvm_object** base class methods through the interface class API. SystemVerilog supports this by redeclaring the **uvm_object** base virtual methods as pure virtual methods inside the interface class[7][11]. This can add extra convenience since you can call these methods directly from the interface class handle.

6.2 Enhancing the Scoreboard

In previous sections, we proposed solutions on how scoreboarding functionality could be propagated via the mixin applied to various transaction items. The interface class scheme with the mixin pattern is not only compatible to the scoreboarding solutions, but also enhances this area. With the minor modification of shifting API methods to the interface class, the implementation and deployment of

---

[9] Interface classes were introduced to SystemVerilog in the 2012 LRM release

[10] See Reference [6] for detailed explanation of using mixins in SystemVerilog to make implementation of interface class methods optional.

[11] This only works for **uvm_object** methods declared as *virtual*. Unfortunately, functions **compare()**, **print()**, and **sprint()** are *not* virtual. So, you must cast from **sys_item_cif** to **uvm_object** to call these functions.

the scoreboarding features are greatly simplified.

We'll demonstrate this by using the previous scoreboarding mixin code scheme example, where two virtual methods **pack_for_sb( )** and **disclose_pack_member_for_sb( )** were combined with a defined **compare_packed_data_for_sb()**. In the interface class version of this solution, the two virtual methods are naturally moved into the interface class section as pure virtual methods. The new location and scheme of these two methods are shown in Figure 20.

```
interface class sys_item_cif;
  ...
pure virtual function void pack_for_sb(ref bit[7:0] packed_bytes[],
                                       ref int num_bytes);

pure virtual function string disclose_pack_member_for_sb(int pack_bit_pos);
 ...
endclass
```

Figure 20: Scoreboarding method declaration of mixin moved to interface class

The defined **compare_packed_data_for_sb( )** method will remain in the mixin class, however its API will be simplified with the ability to reference an interface class like a base class. Therefore, the mixin class type can process data directly with another mixin class type instead of indirectly via generalized data types. This is shown in Figure 21.

```
function bit compare_packed_data_for_sb_cif(sys_item_cif si_cif,
                                            output string disagree_str )  ;
  ...

  // call for packing of local mixin-ed transaction item
  this.pack_for_sb(my_pack_bytes, my_num_bytes);
  // call external transaction item pack for comparison
  si_cif.pack_for_sb(external_packed_bytes, external_num_bytes);
  ...
```

Figure 21: Mixin now using interface class for scoreboarding operations

Note that the API no longer has to abide by a generalized data type (array of bits/bytes), but rather leverage off the interface class to access any information needed. Hence, a portion of the code that had to be implemented in the scoreboard is now brought into the mixin class. This makes more scoreboarding related code exportable to other transaction items, thus simplifying the chip-level scoreboard code. This is demonstrated in the updated scoreboard code in Figure 22, where the array of bytes is no longer needed to be used as an intermediary, but rather one mixin-type instance is directly passed as an argument to another:

```
if (!y_item.compare_packed_data_for_sb_cif(x_item, sb_disagree_string))
   `uvm_info("", sb_disagree_string, UVM_NONE)
else
   `uvm_info("", "Match between X item and Y item", UVM_NONE)
```

Figure 22: Simplified code of testbench scoreboard using the mixin with interface class

6.3 Enhancing Coverage with Generic Covergroups

A powerful coverage technique is to define a generic covergroup that can have multiple instances, with each instance specialized for a particular situation. Figure 23 shows an enhanced version of an example from Reference [5]. This shows how to define a covergroup that can have different min/max bins for each instance using a covergroup constructor argument list. The mixin technique now enhances the solution to manage coverage of different transaction types using a single covergroup definition.

```
class sys_tr_coverage;
  covergroup value_cg(int min, int max, int num_bins)
    with function sample(int x);
      coverpoint x {
      bins lowest = {min};
      bins highest = {max};
      bins middle[num_bins] = {[min+1:max-1]};
  }
  endgroup

  function new(sys_item_cif tr); // pass interface class handle
    // Call accessor methods to shape the bins appropriately
    int max = tr.get_max();
    int min = tr.get_min();
    int values_per_bin = tr.get_values_per_bin();
    int n_bins = (max - min)/values_per_bin;
    value_cg = new(min, max, n_bins);
  endfunction

  function sample(sys_item_cif tr);      //pass interface class handle
    value_cg.sample(tr.get_value());     //call accessor function
  endfunction
endclass
```

Figure 23: This covergroup is reusable for all transaction types and is extensible to reshape the bins.

In this example, we have a universal covergroup definition that is reusable for every transaction type in a testbench, but at the same time has the flexibility to generate unique bins per type. We get this reuse due to the combination of mixin virtual methods **get_max(), get_min()** which can have unique definitions per type, along with a generic interface class handle **sys_item_cif** to call these methods given any transaction type. Figure 24 shows an example coverage monitor applying this same covergroup to both the **x_item** and **y_item** transaction types. Note that we don't have to cast transactions to interface class handles in this example. Casting is done implicitly through the functions which declare the arguments as **sys_item_cif** type.

```
class cov_monitor;
  sys_tr_coverage x_item_cov, y_item_cov;
  sys_x_item x_tr;
  sys_y_item y_tr;


  function new();
    sys_x_item x_item = new("x_item");  // Used only for cg construction
    sys_y_item y_item = new("y_item");

    x_item_cov = new(x_item);     // Implicit cast to interface class handle
    y_item_cov = new(y_item);
  endfunction

  task run_phase(uvm_phase phase);
    ...
    x_item_cov.sample(x_tr);      // Implicit cast to interface class handle
    y_item_cov.sample(y_tr);
    ...
```

Figure 24: A coverage monitor can apply this same coverage class to many transaction types.

This approach also gives us flexibility to overcome some of the SystemVerilog limitations on basic covergroup definitions. SystemVerilog covergroups are not extensible and must be hardcoded, requiring unique versions for each transaction type. But combining mixin inheritance with interface classes allow for the details of covergroup construction to be managed inside virtual methods of classes.[12] This means users can extend these classes and redefine these virtual methods as needed, rather than redefining coverage from scratch. This allows for VIP coverage reuse on different projects as well as vertical reuse of coverage from block-level to system-level testbenches.[13]

6.4 Enhancing Metadata with Linked Transactions

Interface class handles give us another major enhancement in how we accumulate metadata and system context for all transactions in a system. Since every transaction type can be generically referenced using the interface class handle **sys_item_cif**, we can now link relationships between transactions in a generic way inside the **sys_item** mixin class (see Figure 25). This gives us ability to add even more contextual information about transactions observed in a datapath:

- **Source context:** Destination transactions can generically link *back* to the originating source transaction, even if each originated from a different source interface/transaction type.
- **Destination context:** Source broadcast transactions can link *forward* to all corresponding destination items that actually occurred (even for different interface types).
- **List context:** Create a linked list of single transactions that originated from a common burst. This allows each transaction to directly extract context of what came before/after without relying on external data structures (e.g., queues of transactions in a scoreboard).

[12] It may be tempting to put the covergroups directly inside the mixin class rather than a separate coverage class. We recommend avoiding this since typically many thousands of transactions are generated in a simulation and the simulator may not handle merging coverage across so many instances.
[13] Block-level coverage typically has more comprehensive coverage that can't be reached at the system level.

```
class sys_item#(type T=uvm_object) extends T implements sys_item_cif;
 ...
 sys_item_cif  src_tr;   //if set, links back to source transaction
 sys_item_cif  prev_tr;  //if set, links to prev transaction in a series
 ...
```

Figure 25: Interface class handles declared inside the mixin class can link related transactions

This technique adds even more flexibility to functional coverage since we can sample transaction details in the context of related transaction information (e.g., sample a destination address, crossed with the command type of the original source transaction). This also greatly simplifies debugging system-wide data flow. We can now print and trace the flow of data between all interfaces using the common **convert2string()** method on all linked transactions.

Note that assigning these transaction relationships requires a scoreboard to match up predicted transactions to actual observed transaction published by monitors. Passive monitors typically can't assign any metadata since they have limited visibility of a single interface. Predicted transactions can, however, contain all the known metadata from system monitors, scoreboards, environment configuration, and register models.

7. Retrofitting and Factory Override

The solutions and examples we presented in the paper show how the mixin design pattern was used to create multi-inherited sub-classes, where these sub-classes are directly instantiated. However, the SV mixin design pattern implementation also allows factory overriding of the original transaction class objects with the enhanced mixin sub-class. After all, the enhanced mixin class is a sub-class of the original transaction class. This opens the possibility to retrofit new functionality introduced by the mixin sub-class into an existing testbench which only uses original transaction classes.

A handy application where one can exploit the retrofit capability would be to override the original transaction classes in an existing testbench to introduce enhanced debug transaction printing. Taking the previously presented examples of an originating **vip_x_item/vip_y_item** classes and their mixin enhanced **sys_x_item/sys_y_item** sub-classes, the latter's enhanced debug printing capabilities can be added to the existing testbench with two simple steps. First, simply include and declare the mixin sub-classes for each corresponding original transaction class along with the existing code. Second, use the UVM factory to apply the override of each original transaction class with its mixin enhanced class, as demonstrated in Figure 26.

```
// UVM factory override calls to swap in enhanced mixin sub-classes
// instead of the original transaction classes
 set_type_override_by_type(.original_type(vip_x_item::get_type(),
                           .override_type(sys_x_item::get_type());

 set_type_override_by_type(.original_type(vip_y_item::get_type(),
                           .override_type(sys_y_item::get_type());
```

Figure 26: Standard factory overrides allow any testbench to use the mixin class

The factory override declarations must be called at an early stage of the simulation phases for the override to take effect, before the build phase of the underlying VIP environments are called. We

recommend making these override calls at the build phase of a **uvm_test** class that serves as a common parent class for all test cases.

Once again, no code changes are necessary to the existing VIP or test bench components that work with original transaction classes, including any third party of legacy code. The above technique of leveraging the factory override will suffice to retrofit any new functionality brought in by the mixin sub-classes, even on completed and established test benches.

8. Conclusions

The mixin design pattern allows SystemVerilog testbench developers to emulate multiple inheritance and standardized code from a common source to different classes. Typical UVM transaction item classes are excellent targets for this solution since additional system context can be applied to them without requiring any modifications to their original code. Combined with the application of an interface class, the mixin extended transaction item classes can also be accessed with generic handles, providing leaner and extensible code that must work with multiple types.

Debug, coverage, stimulus and checking are all areas that are greatly enhanced with this technique. It is especially well suited for the needs of system level verification. Teams will debug problems more rapidly, transaction coverage will be more accurate with added system context, and stimulus can be controlled in a centralized way with shared constraints. We also gain flexibility that is not possible in a traditional SystemVerilog/UVM testbench. This includes extensible covergroups that are reusable for different transaction types and directly linked relationships of transactions. Furthermore, the mixin pattern code structure is compatible with the UVM factory override to easily retrofit new functionality to native, legacy, or third-party code.

Ultimately this means we no longer have to make compromises in how we design a system testbench based on the choices of UVC developers. We have the power to take complete control of all transaction classes and repurpose them to meet our specific verification challenges. This ultimately allows us to deliver higher quality, bug-free designs while ensuring we meet aggressive schedules.

9. References

[1] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and mixins” in Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 171-183. ACM Press, 1998.

[2] T. Mens, M. van Limberghen, “Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems”, Object Oriented Systems, 3(1):1-30, 1996.

[3] T. Timi, “Fake it ‘til You Make It – Emulating Multiple Inheritance in SystemVerilog”, http://blog.verificationgentleman.com/2014/09/emulating-multiple-inheritance-in-system-verilog.html

[4] M. Litterick, “Lies, Damned Lies, and Coverage”, DVCon USA, 2015.

[5] J. Bromley, M. Litterick, “Effective SystemVerilog Functional Coverage: design and coding recommendations”, SNUG UK, 2016

[6] S. Sokorac, “SystemVerilog Interface Classes – More Useful Than You Thought”, DVCon, USA, 2016

[7] IEEE Computer Society, “1800-2017 - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language”, 2018.

[8] IEEE Computer Society, “1800.2-2017 – IEEE Standard for Universal Verification Methodology Language Reference Moanual”, May 2017.