### Transaction-Level Modeling (TLM)

#### TLM basics
- In OVM, a transaction is a class derived from ovm_transaction (which in turn derived from ovm_object) that includes whatever information is needed to model a unit of communication between two components.
- The transaction-level interfaces define a set of methods that use transaction objects as arguments. A TLM port defines the set of methods (the API) to be used for a particular connection, while a TLM export supplies the implementation of those methods. Connecting a port to an export allows the implementation to be executed when the port method is called.
- The blocking interfaces means that the tasks block execution until they complete. They are not allowed to fail. There is no mechanism for any blocking call to terminate abnormally or otherwise alter the flow of control. They simply wait until the request is satisfied. In a timed system, this means that time may pass between the time the call was initiated and the time it returns. There are several blocking interfaces: put(), get(), peek(), etc.
- In contrast, a nonblocking call returns immediately. The semantics of a nonblocking call guarantee that the call returns in the same delta cycle in which it was issued, that is, without consuming any time, not even a single delta cycle. In OVM, nonblocking calls are modeled as functions. There are several blocking interfaces: try_put(), try_get(), try_peek(), etc.
- The most basic transaction-level operation (simple producer/consumer) allows one component to put a transaction to another. The actual implementation of the put() call is supplied by the consumer.
- The converse operation to put is get. In this case, the consumer requests transactions from the producer via its get port. The get() implementation is supplied by the producer.

#### Analysis communication
- The ovm_analysis_port is a specialized TLM port whose interface consists of a single function, write(). The analysis port contains a list of analysis_exports that are connected to it. When the component calls analysis_port.write(), the analysis_port cycles through the list and calls the write() method of each connected export. If nothing is connected, the write() call simply returns. Thus, an analysis port may be connected to 0, 1, or many analysis exports, but the operation of the component that writes to the analysis port does not depend on the number of exports connected. Because write() is a void function, the call will always complete in the same delta cycle, regardless of how many.
components (for example, scoreboards, coverage collectors, and so on) are connected.

- The simple mapping is one-to-one mapping. OVM analysis port is one-to-many, i.e., one analysis port can connect to multiple analysis exports. If we want a more complicated many-to-many mapping, we need to use SystemVerilog's mailbox.

- The mailbox class and its prototype is as follows:

```
class mailbox #(type T = dynamic_singular_type) ;
    function new(int bound = 0); function int num( );
    task put (T message); function int try_put(T message);
    task get(ref T message); function int try_get(ref T message);
    task peek(ref T message); function int try_peek(ref T message);
endclass
```

**Constraint mechanism for Randomization**

- SystemVerilog has two kinds of random variables defined inside classes: `rand` and `randc`. The `rand` modifier is for pure random, and `randc` modifier is for random-cycle variables (contain one or more permutations with exhaustive random without repeat for any round of permutation before going to the next permutation.) After you instantiate an object from the class, the call instance.randomize( ) will randomize those random variables in this instance. When you call randomize( ), you can add in-line constraints by keyword `with`. You can use `knobs` (of enum type) to control ranges of random variables to make more control on the randomizations. You can specify distribution of a random (but not randc) variable by the `dist` keyword. You can override the pre_randomize( ) and post_randomize( ) for anything before/after the randomization. You can also use `rand_mode( )` to enable/disable any random variable. You can put random variables or `null` as arguments to call randomize( ) to achieve in-line randomization control or for constraint checker. If random variables are passing as arguments, then non-passed random variables are treated as state variable. If null is passed as the argument, then no random variable is assigned and it only checks if constraints are satisfied.

- The keyword `randcase` and `endcase` introduces a random weighted case as a case statement that random selects one of its branches according to the weight. The probability to select anyone is implied by the weights of the one to the total weight.

- The randsequence grammar is composed of one or more productions. Each production contains a name (with optional arguments) and a list of production items (terminals or nonterminals). The probability that a production list is generated can be changed by assigning weights to production lists. A production can be make conditionally by `if` ... `else`, or selected from a set of alternatives by `case` ... `endcase`, or iterate a production to specified number of times using `repeat`, or use the `randjoin` production control to randomly interleave two or more production sequences while maintaining the relative order of each sequence, or use `break` or `return` to abort productions.

- SystemVerilog allow you define any named constraint block as Boolean expressions, set membership with keyword `inside`, iterative constraints with keyword `foreach`, or implication/guard with keywords `->` or `if ... else`. The name or tag for a particular constraint is useful because it allow operations on those constraints by applying to their tags. You can also use `constraint_mode( )` to enable/disable any named constraint. A constraint block can be define as static (i.e., apply to all instances of the class) by including the `static` keyword. The "std::randomize( ) with” form of the scope randomize function allow users to specify random constraints to be applied to local scope variables.

- The constraint solver must assure that the random values are selected to give a uniform value distribution over all legal value combinations in order to allow randomization to better explore the whole design space. However, sometime it is desirable to force certain combinations to occur more frequently. SystemVerilog provide the variable ordering mechanism using keyword `solve ... before`.

- There are random number system functions and methods: $urandom[(int seed)] for unsigned random numbers with seed, $urandom_range( ), srandom(int seed), get_randstate( ), set_randstate( ), etc.

- Layering constraints for testing is achieved using class inheritance. Create a new class that inherits logic from the parent class, defined the same constraints you need to override.

**Functional overage**

- SystemVerilog uses `covergroup/endgroup` to define the coverage model, and the covergroup construct encapsulates a named specification of a coverage model. You can also define covergroups inside a class. The named covergroup with event control is similar to a type declaration,
and you need to instantiate an object (by new( )) from it in order to use
it.

• A covergroup contains named **coverpoints** or named **cross** coverpoints.
Each coverpoint can be guarded by **iff** (expression).

• A coverpoints contains name coverage **bins** or named arrays of bins. A
named bins or named array of bins can have values, list of values, range of
values (using ‘[ : ]’), list of ranged values (using ‘,’), or default (for
the rest values unspecified); it also can have transitions (using ‘=>’),
sequenced/cascaded transitions, set of transitions (combinational),
consecutive repetitions of transitions (using ‘*’) with optional ranges,
nonconsecutive repetitions of transitions (using ‘=' for exact repetition
and ‘->’ for at least repetition with optional ranges), list of transitions, or
default sequence.

• If a coverage point does not define any bins, SystemVerilog
automatically create state bins, one for each possible values/transitions
for the coverage point, subject to the limit by a default value of
auto_bin_max for maximum number of bins. We can specify values/transition with wildcard using named **wildcard bins**. We can exclude coverage point values/transitions by named **illegal_bins**, or
specify named **ignore_bins**. Coverage points can be combined to
become cross coverage points, but which grows exponentially. The
**binsof** construct yields the bins of its expression argument, and the
resulting bins can be further selected by including (by intersect( )) or
excluding (by ‘!’) only the bins whose associated values intersect a
desired set of values. The binsof can combine with other binsof using logical operator && and ||. The open value range syntax can use ‘$’ to
specify an open range.

• The coverage option can be access through option data member using syntax option.option_name = expression. The options are weight (default 1), goal (default 90%), name, comments, at least (default 1),
detect overlap (default false), auto_bin_max (default 64),
cross_num_print_missing (default 0), per_instance (default false). The
coverage type options can be access using syntax type_option.option_name = expression. The coverage type options are:
weight (default 1), goal (default 100%), comments, strobe (default
false).

• The predefined coverage methods are: void sample( ), real
get_coverage([ref int, ref int]), real get_inst_cover([ref int, ref int]),
void set_inst_name(string), void start( ), void stop( ). The predefined coverage system tasks and functions are:
$set_coverage_db_name(string), $load_coverage_db(string),
$get_coverage( ).

• The coverage of a coverage group, **Cg** is the weighted average of the
coverage of all items defined in the coverage group, and it is computed
according the following formulae: **Cg** = (∑i Wi * Ci) / ∑i Wi, where i ∈
set of coverage items defined in the coverage group, **Wi** is the weight
associated with item i, and Ci is the coverage of item i. The coverage of
a cross item is computed according to the following formulae: **Ci** = |binscovered| / (Bc + Bu), and Bc = (∇j Bj) – Bb, where j ∈ set of

---

**Direct programming interface (DPI)**

• DPI is an interface between SystemVerilog and a foreign programming
language. It consists of two separate layers: the **SystemVerilog layer**
and a **foreign language layer**. Both sides of DPI are fully isolated, and
neither the SystemVerilog compiler nor the foreign language compiler is
required to analyze the source code in the other’s language. The
memory spaces owned and allocated by foreign code and SystemVerilog
code are disjoined. Each side is responsible for its own allocated
memory. However, we can do scenarios where foreign code allocates a
block of memory and then passes a handle (i.e., a pointer) to that block
to SystemVerilog code, which in turn calls an imported function that
directly or indirect free that block. Different programming languages can
be used and supported with the same intact SystemVerilog layer. For
now, however, SystemVerilog define a foreign language only for the C
programming language. Though SystemVerilog doesn’t define a C++ for
the foreign language layer, we can use extern “C” to mix C++ code in C
DPI.
The imported functions implemented in a foreign language can be called from SystemVerilog, and the SystemVerilog exported functions can be called from a foreign code. SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction.

A function whose result depends solely on the values of its input arguments and with no side effects can be specified as pure. This can usually allow for more optimizations and thus can result in improved simulation performance. An imported task or function that is intended to call exported tasks or functions or to access SystemVerilog data objects other than its actual arguments must be specified as context. Context tasks and functions can impair SystemVerilog compiler optimizations.

Every task or function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every task or function exported from SystemVerilog defines a global symbol. Import and export declarations allow users to specify a global name for a function in addition to its declared name. An escaped identifier with leading backslash (\) character can be used to avoid name clash with SystemVerilog keywords or reserved names. For example:

```
Export “DPI” foo_plus = function \foo+ ; // “foo+” exported as “foo_plus”
Export “DPI” function foo;  // “foo” exported under its own name
```

Here is an example code for application with complex mix of types:

```c
#include “svdpi.h”
typedef struct {
  int a;
  svBitVecVal b[64][SV_PACKED_DATA_NELEMS(6*8)];
  int c;
} triple;
/* Note that ‘b’ is defined as for ‘bit [6*8-1:0] b[63:0]’ */
extern void exported_sv_func(int, svLogicVecVal*);
/* imported from systemVerilog */
void foo(const triple *t) {
  int i;
  svBitVecVal aB;
  /* aB holds results of part-select from packed array ‘b’ in struct triple */
  svLogicVecVal aL[SV_PACKED_DATA_NELEMS(64)];
  /* aL holds the packed logic array filled in by the export function. */
  printf(“%d $d\n”, t->a, t->c);
  for (i = 0; i < 64; i++) {
    /* read least significant byte of each word of b into aB, then process... */
    svGetPartSelBit(&aB, t->b[i], 0, 8);
  }
  /* call SystemVerilog */
  exported_sv_func(2, aL);
  /* export function writes data into output arg “aL” */
}
```

```systemverilog
typedef struct { int a; bit [6:1][1:8] b [65:2]; int c; } triple;
// troublesome mix of C types and packed arrays
import “DPI” function void foo(input triple t);
export “DPI” function exported_sv_func;
function void exported_sv_func(input int I, output logic [63:0] o);
begin ...
endfunction
```

### Reusability

In OVM, reuse comes from modularity and configurability. There are two kinds of reuse: vertical reuse and horizontal reuse. In OVM, horizontal reuse is done above OVC level by creating an environment, and vertical reuse is done by adding an environment layer. Then use hierarchical configuration mechanism to customize behavior of each environment. All verification IPs is block-specific but context-sensitive.

#### Top-down or hierarchical configuration

- We can configure any parameter by set_config_*. However, those parameters need to be registered using ovm_*_utils macros first before they can be configured. The set_config_* may be set for int, string, ovm_object, etc. The set_type_override_by_type( ) and the set_inst_override_by_type( ) can be called inside an OVM component to create test specific scenarios.
- To configure hierarchical parameters, simply use the parameter instance path name separated by ‘.’. Also, we can use wildcard * to match any.
- By chaning the +OVM_TESTNAME argument in tests to an extended test class, we can run multiple tests without having to recompile or re-elaborate the design or testbench

#### Utility functions

The necessary customization operations are: randomization, printing, cloning, comparing, copying, packing, and transaction recording. Among those operations, SV provides randomization, and OVM provides the rest. We need to use `ovm_object_utils, and `ovm_field_* macros to automatically include all those utility functions (so called the built-in automations: get_type_name( ), create( ), print( ), compare( ), copy( ), clone( ), pack( ), record( ), etc.) The `ovm_field_* macros exist for integer types, strings, arrays, queues, events, class objects, etc. Use OVM_ALL_ON to enable all utility functions. The other flags are OVM_COPY/OVM_NOCOPY, OVM_COMPARE/OVM_NOCOMPARE, OVM_PRINT/ OVM_NOPRINT, OVM_DEEP/OVM_SHALLOW/OVM_REFERENCE, OVM_BIN, OVM_DEC,
The OVM library provides a message action for trace information. Its syntax is `message(<verbosity>, (<message string>))`, where `<message string>` is a formatted string with argument similar to `$display`, and `<verbosity>` is one of the following: OVM_NON, OVM_LOW (the default), OVM_MED, OVM_HIGHT, and OVM_FULL. The output contains `[<time>] hier=<scope>:<message string>`. We can control message option from:

1. +MSG_DETAIL argument to irun, e.g., irun … +MSG_DETAIL=OVM_NONE
2. Use set_reportverbosity_level( ) in test
3. Use ovm_message command in TCL API

### Sequences and virtual sequencers

- OVM sequences are library base classes that allow you to create meaningful ordered scenarios. OVM provides 3 built-in sequences: ovm_random_sequence (the default), ovm_exhaustive_sequence, and ovm_simple_sequence. Use set_config_int("<path to sequencer>", "count", <count>) to configure count for an ovm_random_sequence, and use set_config_string("<path to sequencer>", "default_sequence", "<new sequence>") to configure default sequence inside a test.
- Inside one sequence, we can call `ovm_do` or `ovm_do_with` with the argument instantiated from another sequence class. This is called nested sequences.
- Virtual sequencers can control and reuse other interface sequences (defined in a reusable sequence library) by coordinate data and timing and by layering sequences, which is especially useful in system-level environment. In general, a virtual sequencer (e.g., in system-level) contains references to its subsequencers (e.g., in block-level), but virtual sequencers do not have their own data item and therefore do not execute data items on themselves. Sequencers use `ovm_update_sequence_lib_and_item` macro in the constructor, but virtual sequencers use `ovm_update_sequence_lib` macro.
- We need to create a virtual sequencer to associate to its virtual sequencer. The association is done in the environment by calling set_config_string("<path to sequencer>", "default_sequence", "<virtual sequence>"). A virtual sequence uses `ovm_do_on` or `ovm_do_on_with` to execute sequences on any of the subsequences connected to the current virtual sequencer, and a virtual sequence uses `ovm_do` or `ovm_do_with` to execute other virtual sequences of this sequencer. We also need to connect subsequencers to its virtual sequencer in testbench. Virtual sequencers use grab( ) and ungrab( ) to achieve full control of the underlying drivers for a limited time, and we can use them to handle interrupt. Here is an example:
  - Define an interrupt handler sequence
    ```
    // Upon an interrupt, grab the sequencer, and execute a
    ```

### Factory

- The factory method pattern is one of object-oriented creational design patterns. It deals with the problem of creating objects without specifying the exact class of object that will be created. The factory method design pattern handles this by defining a separate method for creating the objects, whose subclasses can then override to specify the derived type of objects that will be created. All OVM components register themselves with the factory by keeping the type names in a table. Factor returns an instance from create( ). You can use get_type_name( ) to see what you've got. Test can override either by type name or by instance name.