UVM-Connect and TLM-2. 0 primer

Revision 2.3.0 March 2015

Abstract

The UVM Connect library provides TLM1 and TLM2 connectivity between SystemC and SystemVerilog models and components. It also provides a UVM Command API for accessing and controlling UVM simulation from SystemC (or C or C++). This document provides a user guide to the UVM-Connect API package itself as well as a primer on TLM-2.0 usage in general.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

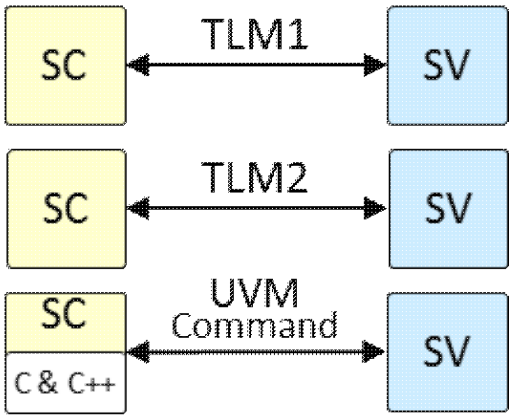
Table of Contents

Table of Contents

1 Introduction to UVM Connect

1.1 Overview

The UVM Connect library provides TLM1 and TLM2 connectivity between SystemC and SystemVerilog models and components. It also provides a UVM Command API for accessing and controlling UVM simulation from SystemC (or C or C++).



1.1.1 Purpose

UVM Connect enables the following use models, all designed to maximize IP reuse.

Abstraction Refinement	Reuse your SC architectural models as reference models in SV verification.
Expansion of VIP Inventory	More off-the-shelf VIP is available when you are no longer confined to VIP written in the language of your testbench.
Leveraging each language	Each language has its strengths. You can leverage SV's powerful constraint solvers and UVM's sequences to provide random stimulus to your SC architectural models. And you can leverage SC's speed and capacity for verification of untimed or loosely timed system-level environments.
Direct access to UVM	The UVM Command API provides a means to wait for UVM phase transitions, raise and drop objections to phase transitions, set and get configuration, issue UVM-formatted reports, set report filters, print UVM topology, set factory overrides by instance and type, and more.

The UVM Connect library makes connecting TLM models in SystemC and UVM in SystemVerilog a relatively straightforward process. However, because UVM Connect is effectively integrating several technologies, you'll need to have basic knowledge of SystemC, SystemVerilog, and the UVM, and TLM standards. Refer to the References section for a partial list of relevant documentation. You may also wish to read the brief TLM Review included in this documentation.

1.1.2 Key Features

This section enumerates some important characteristics of UVM Connect.

Simplicity	Object-based data transfer is accomplished with very little preparation needed by the user.
Optional	The UVM Connect library is provided as a separate, optional package to UVM. You do not need to import the package if your environment does not require cross-language TLM connections or access to the UVM Command API.

Works with Standard UVM	UVM Connect works with the free, open-source Accellera UVM 1.1d and later. With a small modification, UVM Connect can work with previous UVM open-source releases.
Enhances native modeling methodology	UVM Connect does not impose a foreign methodology nor require your models or transactions to inherit from a base class. Your TLM models can fully exploit the features of the language in which they are written.
Supports existing models	Your existing TLM models in both SystemVerilog and SystemC can be reused in a mixed-language context without modification.
Reinforces TLM modeling standards	The UVM Connect reinforces the principles and purpose of the TLM interface standard--enabling independently designed models to communicate without directly referring to each other. Such models become highly reusable. They can be integrated in both native and mixed-language environments without modification. See TLM Review for an introduction to TLM concepts and principles.

1.1.3 Definitions

Definitions for terms used throughout this document.

SV	Short for SystemVerilog, or UVM in SystemVerilog. The context will make clear which. In figures, UVM SV components are shown in shades of blue.
SC	Short for SystemC. In figures, SC components are shown in shades of yellow.
model	Functionality encapsulated by a class. A model is typically a subtypes (derived from) of *sc_module* in SC and *uvm_component* in SV. Although dynamic in nature, models deriving from these classes are quasi-static; they are created during elaboration of the testbench and continue to exist throughout simulation.
component	Synonymous with *model*, above.
hierarchical component	A model that contains one or more models. The hierarchical component is often referred to as the *parent* of the sub-components, which are referred to as its *children*.

See TLM Review for definitions of TLM-related concepts and terms.

1.1.4 Navigating the documentation

How to navigate the documentation.

Click a major heading on the navigation bar at left to expand or collapse the list of items under that heading.

Each page in a major topic or adaptor class is summarized in a box like the one at the top of this page. Each heading or method has a one-line summary description for easy reference. Clicking on the heading or method will take you to its full description.

Clicking on the title of the summary box or full description heading will take you to the actual source file, if available.

Click on *Index* to look up a class or method name whose absolute location is unknown, or you can enter a keyword in the Search box to see a list of matches.

1.1.5 Using with OVM

UVM Connect 2.2 and greater can be compiled to run with OVM 2.1.1 or greater. First, you must set an environment variable, OVM_HOME, to point to a valid OVM release. Then, to compile the OVM and OVMC libraries for 32 and 64 bit Linux, do the following

```
cd $UVMC_HOME/lib
make -f Makefile.<tool> OVM=1 all
```

Where *tool* is the name of the simulator you are using.

Now, try running an example from the OVM-specific *examples_ovm* directory.

```
cd $UVMC_HOME/examples_ovm/connections
make -f Makefile.<tool> sv2sc
```

1.1.6 Tool platform requirements

This section specifies the requirements for compiling and using the UVMC library and included examples.

1.1.7 UVM/ OVM requirements

The latest version requirements

UVM 1.1d or later (earlier versions possible)
OVM 2.1.1 or later
The latest version of UVM can be downloaded from Accellera. (http://www.accellera.org- /activities- /committees- /vip)

OVM can be downloaded from (http://www.verificationacademy.com).

UVM 1.1d adds a simple accessor method to uvm_port_base #(IF) for getting the interface mask of the port that is required by UVMC:

```
376a374,377
>   function int m_get_if_mask();
>     return m_if_mask;
>   endfunction
```

You can get back versions of UVM (1.0p1 or UVM-1.1) to work with UVM Connect by adding this method to the uvm_port_base #(IF) class in *UVM_HOME/src/base/uvm_port_base.svh*. You can make the edit directly in the source file, or you can replace the source file with the one included the *UVM_HOME/compatibility* directory in this kit. No other changes have occurred in this file between UVM 1.0p1 and 1.1d, so it is OK to replace the whole file.

No version of OVM has this accessor method, so port mask compatibility checks are not possible with any OVM release out-of-box. However, like UVM, you can enable port compatibility checks by replacing the file at *OVM_HOME/src/base/ovm_port_base.svh* with the file contained in the *UVMC_HOME/compatibility* directory.

1.1.8 Simulator requirements

The latest simulator requirements

Mentor Questa 10.3 or later recommended / gcc 4.5.0 (Linux)
Synopsys VCS 2014.03-SP1 / gcc 4.7.2 (Linux)
Cadence IUS 14.10 / gcc 4.4.5 (Linux)

UVMC is intended to work with all simulators--it uses standard SV, using only DPI-C behind the scenes. No simulator-specific use models are demonstrated by the examples, but that does not mean those use models are not possible with UVM Connect. For example, most simulators support SV-instantiates-SC and SC-instantiates-SV use models, but these are implemented differently by each vendor. The examples in this kit can be easily converted to employ those use models.

Despite UVM-Connect having been developed on the Mentor Questa simulator, based on end-user feedback, very minor modifications were needed to enable UVM Connect to run on Synopsys' VCS and Cadence's Incisive simulators. The tool-specific makefiles included in this release were well tested and are known to work on the given simulator version, but they may not represent best practice or recommended use models for those simulators. For tool-specific issues and questions, please consult the appropriate vendor.

The UVMC library is intended to be portable across all simulators. If you had to make changes to the UVMC library source to accommodate your simulator, please let Mentor know via Verification Academy (http::/verificationacademy.com), your field representative(s), or Mentor's general support line.

1.1.9 Portability considerations

There are some places in the source code where the coding had to be done differently for the different vendor simulators listed above.

In particular for the "fast packer" converters, techniques for passing dynamic arrays by reference across the language boundary varied among the different vendor simulators. In some cases special extra code had to be added for the VCS and IUS simulators that was not needed for Questa.

To handle such cases you will see use of the following compiler directives for the SystemVerilog source files (*src/connect/sv*),

```
`ifdef VCS   # For Synopsys VCS specific code
`ifdef INCA  # For Cadence IUS specific code (auto-defined by analyzer)
`ifdef VCS_OR_INCA  # For VCS or IUS specific code
```

and the following for SystemC source files (*src/connect/sc*),

```
#ifdef VCS          # For Synopsys VCS specific code
#ifdef NCSC         # For Cadence IUS specific code
#ifdef VCS_OR_NCSC  # For VCS or IUS specific code
```

See comments in affected source code modules for more details of simulator specific concerns - particularly in the TLM GP packer/converter code (*sc/uvmc_converter.svh, sv/uvmc_xl_converer.svh, sc/uvmc_tlm_gp_converter.cpp, sc/uvmc_xl_converter.cpp*).

1.1.10 Platform requirements

The latest platform requirements

Mentor Questa	At present, only Linux 32 and 64-bit platforms are supported. Specific OS and version support is the same as with Questa.
Synopsys VCS	Linux 32/64. Unknown arch/OS/version support. Consult vendor.
Cadence IUS	Linux 32/64. Unknown arch/OS/version support. Consult vendor.

1.1.11 Quickstart 1-2-3 to running examples

1.1.12 Keep it simple

Every attempt was made at keeping required basic steps to running your first example as simple as possible, and consistently so across the 3 supported vendor platforms.

Care was taken to use vendor provided UVM libraries where possible and, in the case of Questa, to even use the pre-built Questa UVM libraries to minimize the number of steps required for a basic example bringup.

More advanced options for compiling and building libraries are also presented later in More details about compiling libraries and running examples but this section will focus on the bare minimum number of steps required to run one of the example simulations.

The steps outlined below are identical for the 3 platforms and are summarized as follows:

- 1. Set approriate ENV vars and customary vendor tool environment
- 2. Build UVMC library
- 3. Run UVMC example

1.1.13 1. ENV setup

First you need to set up the environment for your vendor's simulator in the recommended fashion for that product. If you know how to do this, the additional environment variables you will need to set for UVM and UVMC are shown below.

To run any example, you need compiled UVM and UVMC libraries.

The following environment variables should be set before compiling the UVM and UVMC libraries (if needed), and before running the examples included in this kit.

PATH	Your simulator should be included in your PATH and meet the minimum version requirements stated earlier. If you need to (or plan to) compile the UVM Connect and/or UVM libraries, your path may also need to point to a GCC compiler supported by the simulator. Refer to your simulator documentation for such requirements.
UVM_HOME	The location of the UVM source distribution that meets the minimum version requirements. See UVM/OVM requirements for details. This path is needed for compiling the UVM library, and for locating the *uvm_macros.svh* file when compiling the examples.
UVMC_HOME	The location of the UVMC library source. This is needed when compiling the UVMC library from outside the install directory. It is also needed for locating the UVMC's SC headers and SV's uvmc_macros.svh when compiling the examples.
UVM_LIB	Specifies the location to put the compiled UVM library Default: *$UVMC_HOME/lib/uvmc_lib*. If you're using Questa you can point to one of the pre-compiled UVM_LIB's in the Questa release.
UVMC_LIB	Specifies the location to put the compiled UVMC library Default: *$UVMC_HOME/lib/uvmc_lib*. If you're using Questa you can point to one of the pre-compiled UVMC_LIB's in the Questa release.

If your vendor supplies UVM source code and/or pre-compiled UVM libraries, you can simply set UVM_HOME, UVM_LIB accordingly.

If you have a writable installation of the UVMC kit, you can compile the libraries and examples directly within the UVMC tree. In this case, you only need to specify the UVMC_HOME environment variable and UVMC_LIB will automatically defined to point under there.

Alternatively, you can point UVMC_HOME, UVM_HOME to "read-only" areas and compile either or both libraries to your own areas by setting UVMC_LIB and/or UVM_LIB.

For example, here is the simplest recommended way to set up your ENV for the Questa environment,

```
# Set up PATH and MGC_HOME as is customary for Questa simulator setup

setenv UVM_HOME $MGC_HOME/verilog_src/uvm-1.1d # Source area for UVM macros
setenv UVM_LIB $MGC_HOME/uvm-1.1d              # This is pre-compiled !
setenv UVMC_HOME <path to your UVMC install dir>
setenv UVMC_LIB <local path to your desired UVMC target library>
```

It is possible to override the environment variable settings via Makefile arguments of the same name. This is not recommended because it will be easy to inadvertently compile libraries and examples using different paths.

See the section Mind your ENV for additional suggestions on how to create reusable scripts to set up your environment in a consistent way.

1.1.14 2. Compiling Libraries

Compiled UVM and UVMC libraries are required before you can run the examples. Compilation is done separately from the examples in accordance with standard practice.

Assuming you've set your ENV vars to use pre-existing UVM libriares as recommend above and you've also set your UVMC_HOME to your UVMC installation area and UVMC_LIB to target library area as recommended above, then, to build both 32 and 64 bit libraries simply do the following,

```
make -f $UVMC_HOME/lib/Makefile.<vendor tool> uvmc # Makes 32 and 64 bit libs
```

where *vendor tool* is one of "questa", "vcs", or "ius".

A couple of other options,

```
# Remove all target compiled library directories
make -f $UVMC_HOME/lib/Makefile.<vendor tool> clean

# Print out detailed help information
make -f $UVMC_HOME/lib/Makefile.<vendor tool> help

# Build 64 bit library only
make -f $UVMC_HOME/lib/Makefile.<vendor tool> uvmc64
```

1.1.15 3. Running a UVMC example

All examples can be found in the *$UVMC_HOME/examples* directory.

In each of the examples below assume the command *make* is replaced with *make -f Makefile.<vendor tool>* as explained above. Alternatively you can just change the *Makefile* links you see in each example directory to

point to your vendor's Makefile. By default, the *Makefile* links initially point to *Makefile.questa*.

After following steps above to set up your ENV and compile your UVMC library, it is recommended that you make first a local copy of the examples directory,

```
cp -p -R $UVMC_HOME/examples .
```

Let's now run the *phasing* example test under *examples/commands*,

```
cd examples/commands
make phasing # Runs actual simulation of 'phasing' test.
```

All other examples in the suite follow a similar pattern.

See About the examples for a detailed description of all flavors of examples available. The rest of this document also goes into considerable detail about them for each category.

Other options,

```
make help  # Prints out help message for running tests.
make all   # Runs all tests under particular examples/<category>/ directory.
make clean # Cleans out all generated simulation db files from previous runs.
```

You can also combine targets in one command line,

```
make clean phasing
```

1.1.16 Limitations

1.1.17 SV TLM Limitations

TLM2 features not fully implemented in UVM

- Transport debug interface - tlm_transport_dbg_if
- Direct member interface - tlm_fw_direct_mem_if, tlm_bw_direct_mem_if
- Core initiator and target sockets - UVM provides sockets that provide blocking or non-blocking transport, but not both.
- Quantum keeper
- Payload event queue
- Instance-specific extensions

Should your SC models rely on SV-side implementation of these interfaces, further adaptation may be required to achieve successful interoperability.

There are also several limitations in the current UVM implementation.

- The core sockets in standard TLM2, *tlm_initiator_socket* and *tlm_target_socket*, have no direct counterpart in UVM SV. The standard defines initiator and target sockets that use/implement both the b_transport and nb_transport interfaces. UVM defines sockets that implement either blocking or non-blocking but not both. UVM Connect will still allow connections from SC initiator sockets to SV target UVM sockets, but a run-time fatal error will occur if a blocking call is made to a non-blocking

UVM socket (e.g. uvm_tlm_nb_target_socket) or a non-blocking call is made to a blocking UVM socket (e.g. uvm_tlm_b_target_socket). Refer to Mantis 3682 (http://www.eda.org/svdb/view.php?id=3682)

- The *uvm_tlm_generic_payload* needs several fixes. Refer to (http://www.eda.org/svdb/view.php?id=3983)
- UVM does not fully implement TLM1 non-blocking interfaces. The *ok_to_put*, *ok_to_get*, and *ok_to_peek* methods are defined in the standard to return an event that is triggered once the non-blocking port is able to complete a put, get, or peek operation, respectively. Calls to these interface methods by connected SC-side ports will produce a run-time error and return an event that will never trigger.

1.1.18 Starting SC & SV

Issues associated with starting SystemC and SystemVerilog

SystemC typically elaborates and begins simulation before SystemVerilog has completed elaboration. If the SC side attempts to communicate with SystemVerilog too early, you may get run-time errors or undefined behavior. SC-side ports that are bound to SV-side exports or imps are especially vulnerable to this condition. UVM Connect tries to prevent this from happening in two different ways:

- All UVM Command functions block until SV is ready
- All SC-side calls to TLM ports that are registered for connection across the language boundary will block until its cross-language connection is made.

The implication is that UVMC TLM and Command calls must be made from an SC thread process.

1.1.19 About the examples

The examples included in this kit show how UVMC can be used to integrate IP in a mixed SC and SV environment, without modifying existing IP.

Before attempting to run the examples, be sure to review the Quickstart 1-2-3 to running examples section. Check for support for your platform, confirm that the UVM and UVMC libraries are compiled, and make sure your environment variables are set properly.

1.1.20 Categories

The UVM Connect kit provides examples in 6 major categories--connections, converters, field type support, UVM commands, XLerated connections, config extensions.

Connections	Shows how to establish TLM connections across the language boundary. If you are not using the TLM generic payload transaction type, you will also need to define a converter for your transaction. See Connection Examples
Converters	Provides examples of writing (or generating) transaction converters. See Converter Examples
Field Types	Shows how to pack/unpack each data type that can be declared members (properties, or fields) of your transaction. See Type-Support Examples
UVM Commands	Demonstrates use of the UVM Command API for accessing and controlling UVM simulation from SystemC. For example, you can set configuration, override the factory, issue reports, and control phase progression from outside SV using this API. See UVM Command Examples

XLerated Connections	Similar to connections but shows usage of the "fast packer" type converters that can be used with passing TLM generic payloads (TLM GPs) over UVM TLM-2 and SystemC TLM-2.0 socket connections when support for unlimited payloads and improved performance is desired. See Fast packer converters.
Config extensions	Demonstrates use of configuration extensions to TLM 2.0 generic payloads (TLM GPs) to convey static configuration info or sideband information that can accompany a TLM GP. See Configuration extensions.

1.1.21 More details about compiling libraries and running examples

This section describes more about how to run the examples included in this kit.

1.1.22 Mind your ENV

It is important that you consistently set the values for UVM_HOME and UVMC_HOME for compiling and running examples and for compiling the libraries themselves. The best way to ensure this is to define the environment variables once, perhaps using a shell script or .cshrc file.

```
# Setting required UVM and UVMC environment variables...
source my_uvmc_setup.sh

cd $UVMC_HOME/lib
make clean all

cd ../examples/commands
make all
cd ../converters
make all
...
```

1.1.23 Environment setup template script

If you would like further guidance on a good template script that can be used for environment setups, this section details a template for environment setups that will work for all examples in included in this package and can even be used when building special target libraries, using different vendor simulators. Additionally it can be used as part of an automated procedure to regression test all the package examples (see Running all the examples as a regression test section below).

In each of the example test directories as well as in the $UVMC_HOME/lib/ you'll see a local *Env.script* present. You can run the tests in that individual directory by just manually sourcing the Env.script. This is an alternative technique to setting up the environment on your own as was detailed in the Quickstart 1-2-3 to running examples section above.

If you do choose the Env.script method, you'll notice each Env.script identifies the tools it needs by setting *env_** variables that act as "switches" to identify which tools that test needs then sourcing a master *.toolsrc* script as shown here in a sample *Env.script*,

```
setenv env_gcc  # GCC compiler setup

#Choose one of these (but not all 3):
setenv env_questa  # Mentor Questa setup
#setenv env_vcs     # Synopsys VCS setup
#setenv env_ius     # Cadence IUS setup
```

```
setenv env_uvm  # UVM_HOME setup

setenv env_sysc  # OSCI SystemC
setenv env_vista # Vista SystemC

# $DEMO_ROOT must be set to a directory containing .toolsrc
#   customized to your tool's environment
# Example: setenv DEMO_ROOT $UVMC_HOME/examples/.toolsrc
if( $?DEMO_ROOT ) source $DEMO_ROOT/.toolsrc
```

The Env.script references a “master” .toolsrc env setup file by referencing the env variable *DEMO_ROOT*. So simply set this variable to a directory containing a *.toolsrc* that has been suitably customized for your site.

You will find a well tested template *$UVMC_HOME/examples/.toolsrc* that can be customized to your site settings for the required tool environments mentioned above. Simply search for the pattern, **SITE SPECIFIC** and change the variables enclosed by those blocks to your specific site settings.

1.1.24 Other options for compiling libraries

For building libraries for your specific vendor, here are the common targets which you’ll see printed out from the “make help” command when executing in *$UVMC_HOME/lib/Makefile.<vendor tool>*.

In all cases below assume ‘make <target>’ really means,

```
make -f $UVMC_HOME/lib/Makefile.<vendor tool> <target>
```

For example to print help using ‘questa’ vendor Makefile,

```
make -f $UVMC_HOME/lib/Makefile.questa help
```

You can combine targets to build more than one library.

```
make uvmc      # makes both uvmc32 and uvmc64
make uvmc32    # make UVM Connect (uvmc_pkg) 32b
make uvmc64    # make UVM Connect (uvmc_pkg) 64b

make ovmc      # makes both ovmc32 and ovmc64
make ovmc32    # make UVM Connect for OVM (ovmc_pkg) 32b
make ovmc64    # make UVM Connect for OVM (ovmc_pkg) 64b
```

To build a custom UVM/OVM, override the built-in library distributed with Questa, specify one or more of the following targets **before** any of the uvmc/ovmc targets above. You must define UVM_HOME (or OVM_HOME) when using these targets.

```
make uvm       # makes both uvm32 and uvm64
make uvm32     # make UVM lib (uvm_pkg) 32b (override built-in)
make uvm64     # make UVM lib (uvm_pkg) 64b (override built-in)

make ovm       # makes both ovm32 and ovm64
make ovm32     # make OVM lib (ovm_pkg) 32b (override built-in)
make ovm64     # make OVM lib (ovm_pkg) 64b (override built-in)
```

The recommended usage is to build 32/64-bit libraries for UVM Connect, using a built-in UVM from your vendor,

```
make uvmc
```

Here's an alternate usage example for building 64 bit libraries for both UVM and UVM-Connect,

```
make uvm64 uvmc64 UVM_HOME=<path-to-UVM-source>
```

Presently, the Makefiles are written for compilation on Linux platforms. Future releases may provide make targets for Windows compilation, subject to simulator support for that platform.

1.1.25 Running the examples outside the install tree:

To run the examples outside the UVMC_HOME install tree, all four environment variables must be defined either as environment variables or via the *make* command line. See Quickstart 1-2-3 to running examples for details.

Each example directory relies on a master common Makefile in the *examples/common* directory. Copy the entire examples directory from the install location into a local, writable area, perhaps in your HOME directory or a shared workspace. Then *cd* to any subdirectory and run *make* as before, for example,

```
cp -p -R $UVMC_HOME/examples .
cd examples/commands
make phasing # Runs actual simulation of 'phasing' example test.
```

1.1.26 Running all the examples as a regression test

If you would like to set up a simple regression harness to run the entire suite of UVMC examples that comes with the package please see the section entitled Regression testing below. That section goes into considerable detail on environment setups that will work for all examples in included in this package and can be used to drive automated regression test procedure.

1.1.27 Support for alternative simulator engines

1.1.28 Compiling Standalone SystemC Libraries

In addition to support for native Questa (and VCS and IUS) compiled SV and SystemC libraries, support was also added for standalone libraries that can be used with alterate *SystemC-only* engines, namely OSCI SystemC and Mentor Vista SystemC.

Even Questa can be used with a standalone library for *SystemC-only* use models of UVM-Connect (see Release Notes - UVM Connect about support for SC <-> SC peer UVM-Connect'ions).

You will find special Makefile's for the standalone libraries here,

```
$UVMC_HOME/lib/
  Makefile.uvmc_sysc_standalone_questa
  Makefile.uvmc_sysc_standalone_osci
  Makefile.uvmc_sysc_standalone_vista
```

These each build a library called **uvmc.so** which can be directly linked into the Questa, OSCI SystemC or Vista SystemC kernel programs respectively.

NOTE: For the case of OSCI and Vista this assumes SV-UVM is not even being used. In fact, the SV-UVM infrastructure is completely removed from these libraries. They only support peer SC <-> SC UVM-Connect'ions for these use models.

To build each of these libraries first, make sure you properly set up your normal OSCI SystemC or Vista environments (with appropriate gcc or vista_g++ env setup as well).

For further guidance on a good *.toolsrc* template script that can be used for alternate SystemC simulator environment setups, please see the section entitled Environment setup template script earlier in this chapter.

That section goes into considerable detail on environment setups that can be used when building special target libraries. You'll find a good customizeable *.toolsrc* template under the *$UVMC_HOME/lib* directory.

It is important to make sure the correct $UVMC_BUILD_PLATFORM is set depending on which gcc you're using and 32 vs 64 bit builds (again see *.toolsrc* template for guidance).

Assuming you've made the documented adjustments to your *.toolsrc* template described in the section for native vendor simulator, OSCI SystemC, and/or Vista SystemC environments (with appropriate *gcc/g++* or *vista_g++* env setup as well), you can now follow this procedure to build the standalone libraries,

```
setenv MTI_VCO_MODE 64 # for 64 bit platforms, or '32' for 32 bit platforms

cd $UVMC_HOME/lib/
source Env.script # Which references the pre-customized .toolsrc template

# For Questa SystemC standalone lib ...
gmake -f Makefile.uvmc_sysc_standalone
# For OSCI SystemC standalone lib ...
gmake -f Makefile.uvmc_sysc_standalone_osci
# For Vista SystemC standalone lib ...
gmake -f Makefile.uvmc_sysc_standalone_vista
```

NOTE: You can continue to use the default way of building UVMC libraries for Questa (or VCS or IUS) SystemC+SV-UVM applications. Standalone libraries are *not* needed in this case and the builds of them do not interfere with the "normal" way of building UVM-Connect libraries.

You will see the resulting libraries placed in the following directories,

```
lib/
  questa/
    <platform dir>/
  osci/
    <platform dir>/
  vista/
    <platform dir>/
```

The *<platform dir>/* areas are directory names formed by the setting of $UVMC_BUILD_PLATFORM which has a naming convention that accounts for 32 vs. 64 bit and what GNU *gcc/g++* version is used.

1.1.29 References

A partial list of sources for information on SystemC, SystemVerilog, UVM, and related topics

- [1] Standard SystemC Language Reference Manual; IEEE 1666-2011, March 8, 2011. http://standards.ieee.org- /getieee- /1666- /download- /1666-2011.pdf
- [2] IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language; IEEE 1800-2009; December 11, 2009
- [3] OSCI TLM 2.0 Language Reference Manual, version TLM 2.0.1, Document JA32; copyright Open SystemC Initiative, July 2009 (absorbed into [1] above)
- [4] Universal Verification Methodology (UVM) Accellera; http://www.accellera.org/activities/vip
- [5] Verification Academy - http://verificationacademy.com
- [6] UVM Cookbook - http://verificationacademy.com/uvm-ovm
- [7] UVM World - http://uvmworld.com
- [8] “Are UVM/OVM Macros Evil? A Cost-Benefit Analysis”; DVCon 2011, Feb 2011. http://verificationacademy.com- /uvm-ovm- /MacroCostBenefit

1.1.30 Copyright

```
//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//
```

A partial list of sources for information on SystemC, SystemVerilog, UVM, and related topics
This section provides a basic introduction to TLM ports, exports, interfaces, and sockets as well as basic rules for connecting them.
This section describes more about how to run the examples included in this kit.
The latest version requirements
It is important that you consistently set the values for UVM_HOME and UVMC_HOME for compiling and running examples and for compiling the libraries themselves.
The examples included in this kit show how UVMC can be used to integrate IP in a mixed SC and SV environment, without modifying existing IP.
This section describes how to prepare and run the connection examples including in this kit.
The directory *UVMC_HOME/examples/converters* contains several examples of transaction conversion in both SystemC (SC) and SystemVerilog (SV)
How to run the example demonstrating type support
The *examples/commands* directory contains several examples of using the UVMC Command API from SystemC to query, configure, and control UVM simulation in SystemVerilog.
If you would like to set up a simple regression harness to run the entire suite of UVMC examples that comes with the package please see the section entitled Regression testing below.
These notes provide information about version updates, bugfixes, known issues, changes to supported platforms, etc.

If you would like further guidance on a good template script that can be used for environment setups, this section details a template for environment setups that will work for all examples in included in this package and can even be used when building special target libraries, using different vendor simulators.

2 Release Notes - UVM Connect

These notes provide information about version updates, bugfixes, known issues, changes to supported platforms, etc. Updates and changes made prior to public release are not included.

2.1 UVM Connect v2. 3.0 - March 2015

Notes for release v2.3.0

2.1.1 Feature summary

This *uvmc-2.3.0/* library is a variation of the original *uvmc-2.2/* library that has had the following additional features added:

- Support for “fast packers”
- Support for SC <-> SC peer UVM-Connect’ions
- Support for SystemC standalone libraries
- Support for “configuration extensions”
- Support for order-independent rendezvous

2.1.2 Support for “fast packers”

The uvmc-2.3.0 release adds support for “fast packers” for the specific case of passing TLM generic payloads (class **uvm_tlm_generic_payload**) across UVM-Connect’ed sockets.

These “fast packers” add two features,

- Improved performance
- Support for TLM generic payloads with no fixed limitations on data payload sizes (i.e. unlimited data payloads)

There are two flavors of fast packers,

```
1. class uvmc_xl_converter
2. class uvmc_tlm_gp_converter
```

The two new classes have both been enhanced for better performance than the default packers, and both support unlimited payloads. But there are slightly differing semantics for each of the two.

- The class **uvmc_xl_converter** conforms in the strictest sense to the required semantics of the TLM-2.0 base protocol specifically with respect to modifiability of attributes (see IEEE 1666-2011 section on TLM-2.0 base protocol), and thus does not indiscriminately transfer all fields of the generic payload in both directions across the language boundary. Rather it decides, depending on the mode of the transaction (READ or WRITE), and whether it is being transferred along the forward, backward, or return paths, which fields to transfer and which to leave alone.
- The class **uvmc_tlm_gp_converter** has the same features of unlimited payload size and efficient data payload passing techniques that use “C assist” and “pass by reference” that version #1 above does, but it is unconditionally transferring all fields of the generic payload along all paths without regard to modifiability of attributes which is more semantically compatible with the slower, size limited default packer.

- For the packers themselves see,

```
src/connect/
  sc/uvmc_tlm_gp_converter.*
  sc/uvmc_xl_converter.*
  sv/uvmc_converter.svh
  sv/uvmc_xl_converter.svh
```

- Specifically see sc/uvmc_tlm_gp_converter.h for an explanation of when you would want to use class **uvmc_tlm_gp_converter** vs. class **uvmc_xl_converter**.
- For examples that use them see

```
examples/xlerate.connections/Makefile
```

2.1.3 Support for SC <-> SC peer UVM-Connect'ions

Previously support for SC <-> SC peer UVM-Connect'ions did not exist but was added to allow SystemC applications to create UVM-Connect'ions without knowing apriori whether the opposite endpoint will be in a SystemC model or an SV-UVM model. It also provides a very easy, intuitive way to bind SystemC TLM-2.0 ports and let the overloaded variations of the **uvmc_connect()** function automatically figure out whether they are initiator port or target export bindings. Just pass the port and the ID string and Presto ! UVM-Connect figures out the rest !

2.1.4 Support for SystemC standalone libraries

In addition to support for native Questa (and VCS and IUS) compiled SystemC libraries, support was also added for standalone libraries that can be used with OSCI SystemC and Vista SystemC.

You will find special Makefile's for the standalone libraries here,

```
lib/
  Makefile.uvmc_sysc_standalone_questa
  Makefile.uvmc_sysc_standalone_osci
  Makefile.uvmc_sysc_standalone_vista
```

These each build a library called **uvmc.so** which can be directly linked into the Questa, OSCI SystemC or Vista SystemC kernel program respectively.

NOTE: For the case of OSCI and Vista this assumes SV-UVM is not even being used. In fact, the SV-UVM infrastructure is completely removed from these libraries. They only support peer SC <-> SC UVM-Connect'ions for these use models.

See Compiling Standalone SystemC Libraries in the main *UVM Connect->Introduction* section for more info on how to build these standalone libraries.

2.1.5 Support for "configuration extensions"

Configuration extensions are ignorable extensions (in the sense of TLM-2.0 generic payloads) that can be used to pass configurations which accompany generic payloads that travel from TLM-2.0 initiators to targets.

The UVMC config extension base class **uvmc_xl_config** contains a simple abstraction of a set configuration registers that can act as shadows of the associated configuration register set one might find in the target

model.

There are two types of configuration extensions that are handled by **class uvmc_xl_config**,

1. Static configuration register

- Static configurations are sent as separate dedicated transactions to update configuration register sets on the target side of the connection.
- Static configs can be used for configuring things that don't change often such as UART baud rate, AXI randomized wait state bounds and cross channel latencies.

2. Sideband configuration register

- Sideband confgurations are unconditionally sent with each and every generic payload transaction along the forward path to the target.
- These should be used for things that typically change as frequently as every transaction such as tid's for AXI transactions, tags for Wishbone transactions, etc.

NOTE: The **class uvmc_xl_config** TLM GP extension is designed to be used **only** with TLM GPs passed to the **class uvmc_xl_converter** fast packer described above. You can attach them to TLM GPs that use other packers but the extension itself may not accompany the TLM GP across the TLM channel in that case (certainly not for UVMC default converters or **class uvmc_tlm_gp_converter** fast packers).

For the config extensions themselves see,

```
src/connect/
  sc/uvmc_xl_config.*
  sv/uvmc_xl_config.svh
```

2.1.6 Support for order-independent rendezvous

The uvmc-2.3.0 release has been enhanced to allow for *order-independent rendezvous* of TLM port connections. There is now a more relaxed dependency on the ordering between when an SV-side peer uvmc_connect()'s its port and when the SC-side peer does so. Same applies for SystemC peer-to-peer UVM-Connect'ions.

This allows more flexibly for "late bindings" of UVM-Connect'ions. Only requirement is that no transaction communication is done on any cross-language port that has not been bound. An error will occur if any such attempts are prematurely made before all peers have connected.

2.1.7 UVM Connect v2. 2 - August 2012

Notes for release v2.2.

- OVM Support. You can now compile UVM Connect to work with OVM 2.1.1 or greater. Compile libraries with OVM=1

```
cd $UVMC_HOME/lib
make -f Makefile.<tool> OVM=1 all
cd $UVMC_HOME/examples_ovm/connections
make -f Makefile.<tool> sv2sc
```

See Using with OVM for details.

- Added support for two other vendors' simulators (UVM only).
- Added ability to set stack size for SC background processes used to make blocking calls on behalf of SV initiators.
- Refactored internal implementation for efficiency
- Improved some messages.
- Removed registration of both lookup string and port hierarchical name. Only one will ever be used. So, if lookup string provided, that is what is used to match against other ports. Otherwise, the port's full name is registered as the lookup string.
- Removed inclusion of Questa-specific libraries to reduce size of distribution. Compiling UVM Connect is quick and straight-forward no matter what simulator you are using. See <Compiling Libraries> for how to compile the UVM (or OVM) and UVM Connect libraries.

2.1.8 UVM Connect v2. 1.4 - February 2012

Notes for release v2.1.4.

Key additions to this release include

- Improved TLM2 support
- More comprehensive User Guide with supporting examples, all documented
- Support for hierarchical connections, i.e. wrapping foreign models and promoting their TLM connections to native TLM ports using UVM Connect.
- Additional examples, reorganized. All examples are found in $UVMC_HOME/examples. See the Overview page in the online documentation for information on running the examples included in this kit.
- HTML documentation added.

While the kit is intended to work with all three simulators, correct operation on other simulators has not been verified.

2.1.8.1 Version requirements

UVM 1.1a see Overview for instructions on enabling earlier versions
Questa 10.1 see Overview for minor restrictions for use with 10.0c or later.

```
//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//
```

In addition to support for native Questa (and VCS and IUS) compiled SV and SystemC libraries, support was also added for standalone libraries that can be used with alterate *SystemC-only* engines, namely OSCI SystemC and Mentor Vista SystemC.
UVM Connect 2.2 and greater can be compiled to run with OVM 2.1.1 or greater.
To communicate, verification components must agree on the data they are exchanging and the interface used to exchange that data.

3 TLM Review

This section provides a basic introduction to TLM ports, exports, interfaces, and sockets as well as basic rules for connecting them. See the References section for more in-depth materials, including the IEEE 1666-2011, the SystemC LRM that defines the TLM standard.

3.1 Terms

Definitions of terms used throughout this document.

SV	SystemVerilog, or UVM in SystemVerilog. The context will make clear which. In figures, UVM SV components are shown in blue.
SC	SystemC. In figures, SC components are shown in yellow.
model	Functionality encapsulated by a class. A model is typically a subtypes (derived from) of *sc_module* in SC and *uvm_component* in SV. Although dynamic in nature, models deriving from these classes are quasi-static; they are created during elaboration of the testbench and continue to exist throughout simulation.
component	Synonymous with *model*, above.
hierarchical component	A model contains one or more models.
TLM	Transaction-Level Model. TLM models exchange information as objects, not the discrete signals or wires you see in RTL. Transactions abstract away the low-level RTL details in exchange for speed and easier maintenance. To be reusable, TLM models should communicate via standard TLM1 or TLM2 interfaces.
interface	A class that defines one or more method prototypes but does not implement them. Classes inheriting from an interface must implement the methods defined in the interface. TLM1 and TLM2 define several interfaces for transaction-level communication. Models that implement an interface are referred to as *targets*.
imp	SV only. Like an interface, except instead of providing an interface implementation through inheritance, the *imp* is an object that provides an interface through delegation. It is a SV workaround to lack of multiple inheritance.
export	An object that conveys (exports) an interface implementation from the child to the parent level. Exports can also be connected to other exports for purposes of promoting the interface implementation as high in the model hierarchy as needed.
port	An object through which interface calls are made. A port is connected to the interface implementation via a combination of parent port-export-imp connections. Thus, calls to a port in an initiator component end up calling the method implementations in the target component, with neither the initiator nor target knowing about each other.

Who initiates requests and who services them dictates the control flow relationship between models.

initiator	A component that initiates transaction requests. Initiators typically contain at least one port or initiator_socket.
target	A component to which transaction requests are sent. Targets typically implement an interface (SC) or contain at an initiator_socket or imp.

Who creates the transactions and who processes them dictate the data flow relationship between models.

producer	A component that produces transactions.
consumer	A component that consumes or executes transactions.

There are thus four combinations of control and data flow possible for any given TLM connection.

- *Initiator-producers* create transactions and send them out TLM ports or sockets.
- *Target-consumers* receive transactions from initiator-producers via TLM exports, interfaces, imps, or sockets. *Initiator-consumers* request transactions from target-producers via TLM ports or sockets.
- *Initiator-consumers* request transactions from connected *target-producers*. Although it does not use a standard TLM interface, the UVM driver is an example of a *initiator-consumer*.
- *Target-producers* create transactions upon request from *initiator-consumers*. Although not a component using standard TLM, the UVM sequencer is an example of a *target-producer*, accepting requests for transactions from the driver.

3.1.1 Context Independence

The purpose of TLM is to allow components to be self-contained, independent of the myriad ways they might be connected in a testbench. Likewise, connections should be not depend on the components' internal implementation details. The TLM standard defines the set of common interfaces and semantics required to make successful connections to independently developed components.

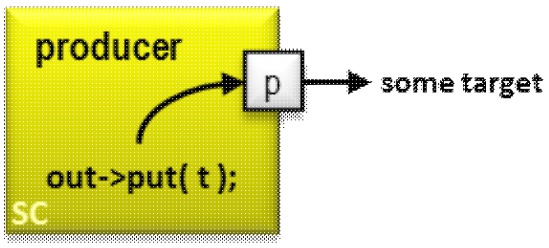
Provided the IP you intend to integrate has properly applied the principles of context independence and TLM connectivity, mixed-language interoperability with UVMC can be achieved with no modifications to the IP on either side of the language boundary.

The examples included in this kit reinforce these concepts. All examples are fully documented with diagrams, explanations, and code in hyperlinked HTML. Together they show you step-by-step how to integrate existing IP in a mixed SystemC / SystemVerilog environment.

3.1.2 Ports

Ports are used to call interface methods implemented elsewhere.

- Ports are the "starting point" for communication by initiators
- The testbench developer (integrator) connects ports external to the owner of th eport.
- Ports are depicted as square in diagrams



3.1.2.1 SC Example

```
class producer : public sc_module
{
  sc_port<tlm_blocking_put_if<packet> > out;  <--

  producer(sc_module_name nm) : out("out"){
     SC_THREAD(run);
  }

  void run() {
     packet t;
```

```
      ...initialize/randomize packet...
      out->put(t); <--
    }

};
```

3.1.2.2 SV Example

```
class producer extends uvm_component
{
  tlm_blocking_put_port #(packet) out; <--

  `uvm_component_utils(producer)

  function new (string name, uvm_component parent=null);
    super.new(name,parent);
    out = new("out",  this);
  endfunction

  virtual task run_phase (uvm_phase phase);
    packet t = packet::type_id::create("tr",this);
    t.randomize();
    `uvm_info("PRODUCER/PKT/SEND", t.sprint(),UVM_MEDIUM)
    out.put(t);   <--
  endtask
};
```

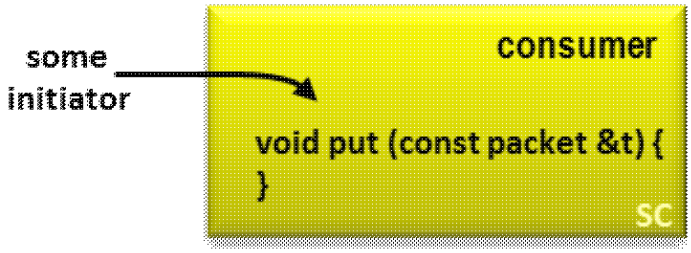
3.1.3 Interfaces & Imps

An *interface* in SC and an *imp* in SV UVM are used to expose to the outside world an implementation of a standard TLM interface, which is a group of methods with predefined signatures and semantics. The interface is typically a small subset of a component's overall API. Thus, the TLM interface and imp minimize the coupling between components by exposing only the standard portion of its API.

- Imps/interfaces are "end points" in a network of port/export/interface connections
- Inteface methods are called via ports bound to the interface/imp, NOT directly
- Interfaces/imps are depicted as a circle in diagrams

3.1.3.1 SC Example

In SC, target components inherit the interface & implement the interface's methods

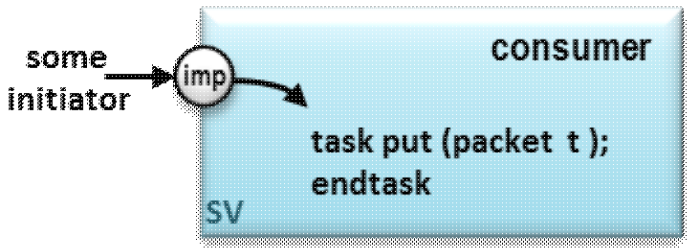



```
class consumer : public sc_module,
                 tlm_blocking_put_if<packet> <-- interface inherited
{
   public:
   consumer(sc_module_name nm) { }
```

```
  virtual void put(const packet &t) {    <-- implementation
    cout << "Got packet: " << t << endl;
    wait(10,SC_NS);
  }
};
```

3.1.3.2 SV Example

In SV, target components provide an "imp" object for connecting to the outside world. The *imp* delegates to the interface implementations provided in the component.




```
class consumer extends uvm_component;

  uvm_blocking_put_imp #(packet,consumer) in;  <-- imp object

  `uvm_component_utils(consumer)

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      in = new("in",  this);
   endfunction

   virtual task put (packet t);   <-- implementation
     `uvm_info("CONSUMER/PKT/RECV",
               t.sprint(),UVM_MEDIUM)
     #10ns;
   endtask

endclass
```

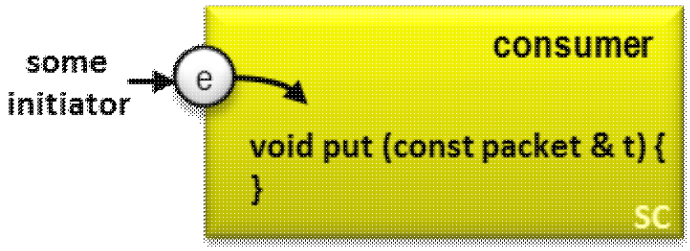
3.1.4 Exports

Exports promote an interface (or imp) implementation from a child to its parent.

- Promotes (exports) interface implementations from a child (or self) up a level
- Internally bound to child export, imp / interface in c'tor
- Externally connected to port or parent export, but not required. (If no connection, no activity)
- Exports are depicted as circle in diagrams

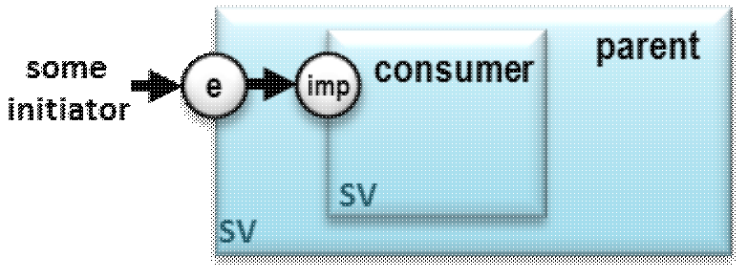
3.1.4.1 SC Example

Often, an SC component that implements an interface may provide that interface via an explicit export object, much like the *imp* provides an interface in SV. In this case, the SC Target binds its own interface implementation to the export. Then, a port can be connected directly to the export, e.g. *prod.out.bind(cons.in);*




```
class consumer : public sc_module,
                 tlm_blocking_put_if<packet>
{
  public:

  sc_export<tlm_blocking_put_if<packet> > in;  <-- export

  consumer(sc_module_name nm) : in("in") {
    in(*this);                  <-- promote own intf impl
  }

 virtual void put(const packet &t) {
    cout << "Got packet: " << t << endl;
    wait(10,SC_NS);
  }
};
```

3.1.4.2 SV Example




```
class parent extends uvm_component;

  uvm_blocking_put_export #(packet) in;

  consumer cons;

  `uvm_component_utils(consumer)

  function new(string name, uvm_component parent=null);
     super.new(name,parent);
     in = new("in",  this);
  endfunction

  function void build_phase(uvm_phase phase);
    cons=consumer::type_id::create("const",this);
  endfunction

  function void connect_phase(uvm_phase phase);
    in.connect(consumer.in);
  endfunction
```

```
endclass
```

3.1.5 Analysis

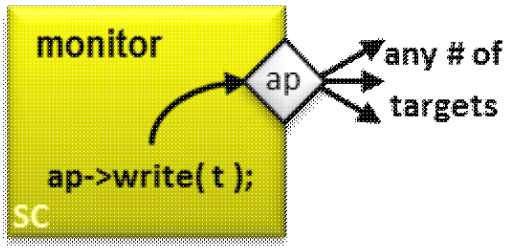
Analysis ports, exports, and imps are used to monitor transaction flow and help you debug your designs. Components emitting transactions out an analysis port are not necessarily the producers of those transactions. In all cases, components receiving transactions from an analysis connection do not execute them or modify them in any way.

3.1.5.1 Analysis Ports

Analysis ports are a special kind of TLM port. They publish transactions to any number of listeners, including zero.

- Broadcasts ("publishes") to all connected targets ("subscribers")
- Transactions are read-only. Used for debug, scoreboards, etc.
- Usually does not require connection (SC_ZERO_OR_MORE_BOUND)
- Depicted as diamond in diagrams

3.1.5.2 SC Example



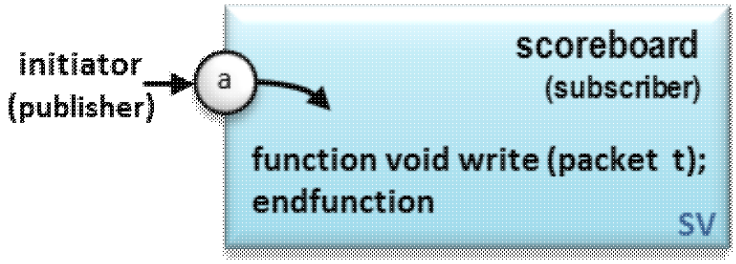

```
class monitor: public sc_module
{
  sc_port<tlm_analysis_if<packet>,
          0, SC_ZERO_OR_MORE_BOUND> > ap;

  monitor(sc_module_name nm) : ap("ap"){
     SC_THREAD(run);
  }
  void run() {
     packet t;
     ...gather packet off bus...
     ap->write( t );  <--
  }
};
```

3.1.5.3 Analysis Exports & Imps

Analysis exports and imps are subscribers to a TLM analysis port.

- Receives streams of transactions from connected publisher (e.g. monitor)
- Transactions are read-only, i.e. for debugging, scoreboarding, etc.
- Like all exports and imps, does not typically *require* a connection
- Depicted as circle in diagrams

3.1.5.4 SV Example



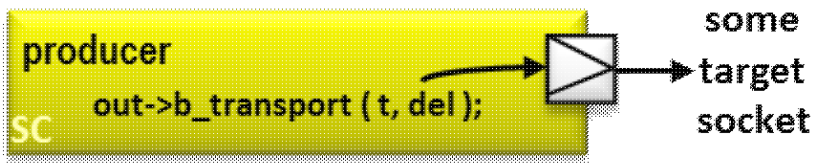

```
class scoreboard extends uvm_component;

   uvm_analysis_imp #(packet,scoreboard) actual_in;  <-- imp

   `uvm_component_utils(scoreboard)

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      actual_in = new("actual_in", this);
      ...
   endfunction

   virtual function void write(packet t); <-- called by actual_in
     packet exp;
     `uvm_info("SB/PKT/RECV",t.sprint(),UVM_MEDIUM)
     if (!expect_fifo.try_get(exp)) ...error
     if (!t.compare(exp)) ...error
   endfunction
endclass
```

3.1.6 TLM Initiator Socket

Sockets are a convenient way to make TLM2 connections; in fact, most TLM2 connections are made with sockets, not individual interface connections.

- Can do blocking or non-blocking transport (usually one or the other)
- Default type is *tlm_generic_payload* with base protocol semantics
- Initiator must implement backward interface, unless a simple initiator socket is used (in tlm_utils namespace)
- When driving an SV target, the *DMI* and *debug* interface calls will be ignored, as they are not implemented in SV.
- When driving an SC target from SV, the *DMI* and *debug* interfaces will not be called, so your SC models should not rely on them being called.
- Depicted as square with outward facing arrow

3.1.6.1 SC Example



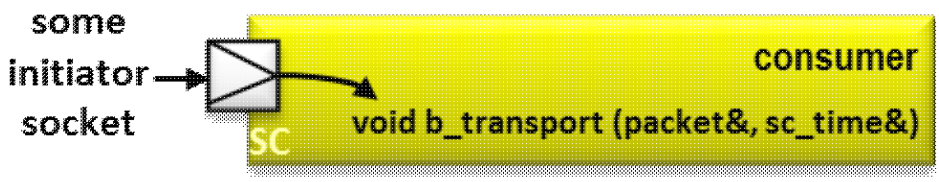
```
struct producer: public sc_module,
                 public tlm_bw_transport_if< >
{
  tlm::tlm_initiator_socket< > out; // default: tlm_gp

  producer (sc_module_name nm) : out("out") {
    out(*this); // bind bw intf to self
    SC_THREAD(fw_proc);
  }

  // FORWARD PATH

  void fw_proc() {
    // produce tlm gp trans, then emit using...
    out->b_transport(t,del);
       *or*
    out->nb_transport_fw(t,ph,del);
  }

  // BACKWARD PATH

  virtual tlm_sync_enum nb_transport_bw(...) {
    ...coordinate with fw path, per protocol
  }

  virtual void invalidate_direct_mem_ptr(...) {
    // Dummy implementation
  }
};
```

3.1.7 TLM Target Socket

A component having a target socket receives transaction on its forward interface and, if non-blocking, sends responses to the initiator via the backward path.

Some other key aspects of target sockets include

- They implement both blocking or non-blocking transport interface, although the connected initiator typically uses one or the other.
- The default transaction type is the *tlm_generic_payload* executed with *TLM base protocol* semantics
- The target model must implement all of forward interface unless the simple target socket is used, in which case only those methods that are registered need to be implemented.
- Because UVM SV does not support the direct memory and debug interfaces, UVMC stubs these out. Attempts to use these interfaces by SV initiators will be ignored.
- Sockets are depicted in diagrams as a square with inward facing arrow.



3.1.7.1 SC example

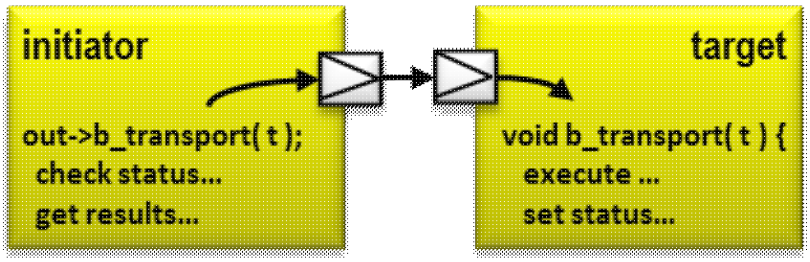
```
struct consumer: public sc_module,
                 public tlm_fw_transport_if< > {

  tlm::tlm_target_socket< > in;

  consumer(sc_module_name nm) : in("in") {
    in.bind(*this);     SC_THREAD(bw_proc);
  }

  // FORWARD PATH

  void b_transport( packet& trans,sc_time& t ) {
    // fully execute request, modify args, return
  }

  tlm_sync_enum nb_transport_fw(...) {
    // per protocol, update args as allowed,return
  }

  bool get_direct_mem_ptr() { return FALSE; }

  unsigned int transport_dbg() { return 0; }


  // BACKWARD PATH

  void bw_proc() {
     ...coordinate with fw transport per protocol
     in->nb_transport_bw(trans,ph,delay);
  }
};
```

3.1.8 Socket Connections

A socket is used to connect a forward and backward path between an initiator and target using a single *connect* or *bind* call. Although sockets support both blocking and non-blocking semantics, typically only one of them is in play for any given connection.

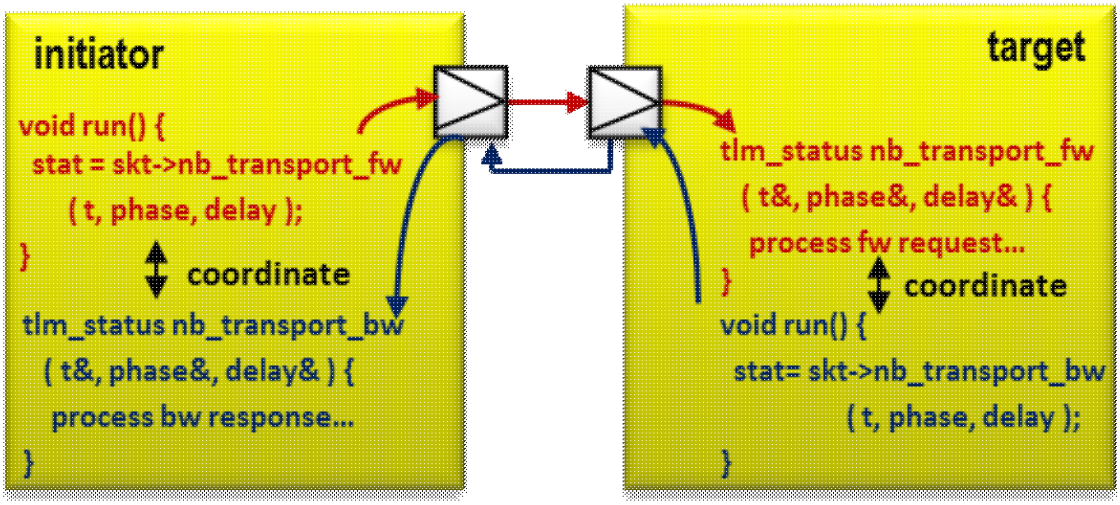
3.1.8.1 Blocking Transport

- Initiator indirectly calls *b_transport* in Target
- Initiator must not modify transaction; transaction contents invalid until *b_transport* returns
- When *b_transport* returns, transaction is complete with status/results
- Transaction can be reused in next *b_transport* call



3.1.8.2 Non-blocking Transport using Base Protocol

- Initiator starts request by calling *nb_transport_fw* in Target. Target returns with updated arguments.
- Target can call *nb_transport_bw* in Initiator at phase transitions. To provide Initiator updates; Initiator may respond via fw interface.
- Transaction contents, phase, & delay can change. Only certain fields in certain phases, according to base protocol rules.
- Transport calls continue back and forth until either returns transaction complete status. For efficiency, the same transaction handle is used throughout its execution.

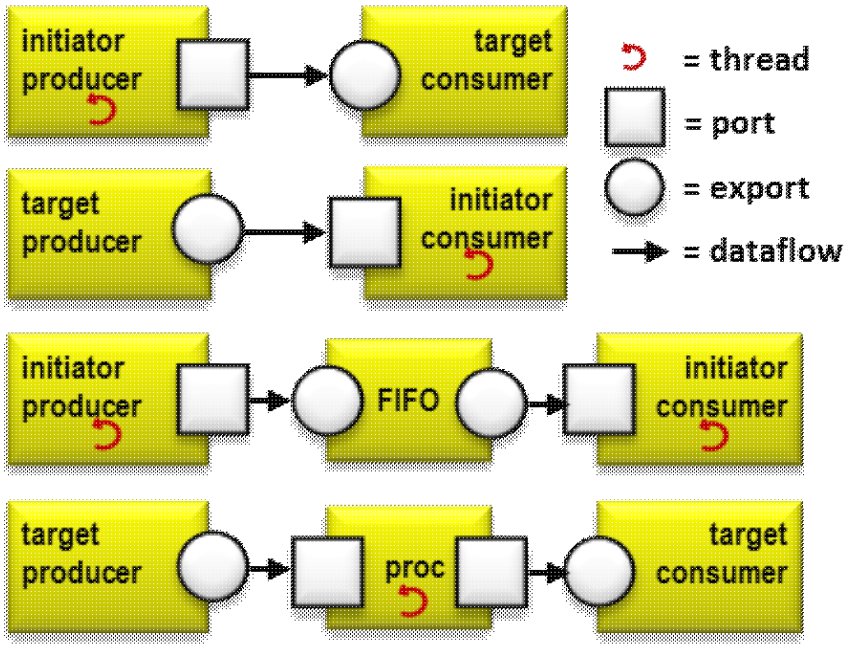


3.1.9 Legal TLM Connections

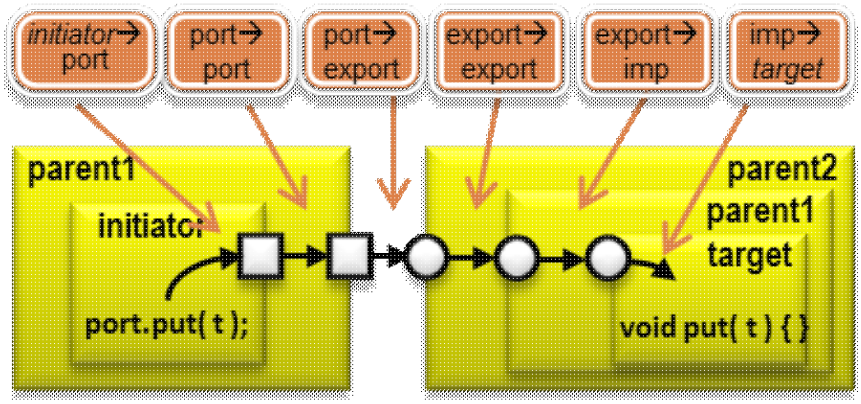
TLM port connections are like Verilog module port connections except you're connecting interfaces not wires. Control and data flow through each TLM connection can be in opposite directions.

A successful connection requires a given pair of ports to agree on the interface and transaction types as well as the direction of control and data flow.

Ports	Ports can connect/bind to parent ports, sibling exports, and sibling interfaces/imps. *prod.port.bind(port|export|intf|imp)*
Exports	Exports can connect/bind to child exports and child interfaces/imps. *export.bind(export|imp|interface)*
Interfaces (SC)	Interfaces in Systemc are pure virtual classes that are inherited by target. They are never on the left-hand side of a bind or connect call.
Imps (SV)	Imps in SV are implicitly bound to the parent in the parent's constructor. *imp = new("name", this)*, where *this* is the parent. They are never on the left-hand side of a bind or connect call.



A connection is hierarchical when a port connects to a parent port or an export connects to a child export, interface, or imp. The following figure shows the legal port-export-imp-interface connections. Passthrough TLM sockets in SV UVM are used to make hierarchical socket connections.



3.1.10 TLM Generic Payload

TLM 2.0 defines a canonical transaction type, *tlm_generic_payload*, or *tlm_gp* for short. TLM 2.0 also defines a base protocol for execution of the generic payload over standard initiator and target sockets.

When used together--*tlm_gp* with the *base protocol*--interoperability potential is at its highest.

The TLM GP defines the following major fields.

command	READ, WRITE, or IGNORE
address	Base address
data	Data buffer. An array of bytes
data_length	Number of valid bytes in data buffer
response_status	OK, INCOMPLETE, GENERIC_ERROR, ADDRESS_ERROR, BURST_ERROR, etc.
byte_enable	Byte-enable data buffer

byte_enable_lengthNumber of valid byte-enables buffer
Between SystemC, UVM, and the UVMC libraries, the tlm generic payload transaction definitions and converters come pre-defined for you. You do not need to define converters for *tlm_gp*. You can just connect and go!

3.1.11 TLM1 combination interfaces

TLM ports *require* a connection to an implementation of its interface type, while TLM exports, interfaces, and imps *provide* the implementation. As long as the provider provides *at least* the required interface, the connection is allowed.

In most connections, you will be connecting TLM ports that are typed to matching interfaces, e.g. a *uvm_tlm_blocking_put_port #(my_trans)* would be connected to a *uvm_tlm_blocking_put_imp #(my_trans)*, or in SC, an *sc_port< tlm_blocking_put_if<my_trans> >* would be connected to a component implementing (inheriting) *tlm_blocking_put_if<my_trans>*.

To increase connection options for integrators, a VIP designer may opt to provide both the blocking and non-blocking interfaces, e.g. provide a *uvm_tlm_put_imp #(my_tras)*. Integrators may connect to this any of *uvm_tlm_blocking_put_port #(my_trans)*, *uvm_tlm_nonblocking_put_port #(my_trans)*, or of course *uvm_tlm_put_port #(my_trans)*.

As you will see in the chapter on UVMC Connections, the same options when making connections that cross the language boundary.

3.1.12 UVMC TLM Connections

Getting your SystemC TLM models and SystemVerilog UVM components talking to each other breaks down to two steps

- Define converters, if necessary
- Make connections

If you're not using the TLM generic payload, you must define compatible transaction classes in SV and SC and the converters to go between them. If the transactions and the components that use them are pre-existing, this task entails writing a custom converter class that not only packs and unpacks but adapts to the differences in member types, member number, and declaration order between the two classes.

If a transaction type pre-exists in one language but not the other, you would need to define the missing transaction type first, then define the converters to go between it and the original transaction. Try to define the class to match the existing definition as closely as possible.

In UVM, your transaction should extend uvm_sequence_item. It must implement the *do_pack* and *do_unpack* methods, or it must use the *`uvm_field* macros (not recommended). The number, order, and manner of unpacking must be compatible with that for unpacking. Typically, one is the exact reverse of the other.

```
//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
```

```
//   the License at                                                 //
//                                                                  //
//       http://www.apache.org/licenses/LICENSE-2.0                 //
//                                                                  //
//   Unless required by applicable law or agreed to in              //
//   writing, software distributed under the License is             //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR          //
//   CONDITIONS OF ANY KIND, either express or implied.  See        //
//   the License for the specific language governing                //
//   permissions and limitations under the License.                 //
//------------------------------------------------------------------//
```

A partial list of sources for information on SystemC, SystemVerilog, UVM, and related topics
This chapter shows how to make TLM connections between SystemC and SystemVerilog UVM components.

4 Regression testing

4.1 Regression testing your UVMC examples

The following procedures show how to automatically regression test all the examples included in this package with a simple scheme that deploys recursive use of Makefiles applied from the root of the example tree. The same procedure can regression test all 3 vendor platforms, Questa, VCS, IUS.

4.1.1 Regression tree structure

The *regression root* defines a *tree of tests*.

Starting with the root, the tests are organized as a hierarchical structure of *tree branch node directories* and *leaf test directories*.

Each leaf test directory of the tree defines a specific unit test area where a given test or set of related tests sharing the same source code files are run.

And each branch node of the tree, including the regression root node itself, leads to other branch nodes, or leaf tests, or both.

Specifically for the UVM-Connect examples the *regression root* starts out at **$UVMC_HOME/examples/** and there are no branch nodes other than the regression root itself. Below the root node, there are 5 *leaf test directories*,

```
commands/
connections/
converters/
field_types/
xlerate.connections/
config_exts/
```

4.1.2 Running regression trees recursively

The entire regression tree structure is *fractal* in nature. Any branch node reachable from the *regression root node* has the same regression run semantics as the root itself. And every node or leaf test has a Makefile (for Questa, VCS, IUS) that completely self documents the procedures recursing to lower nodes or running the given tests.

And the Makefile format used here is intentionally kept very “bare-bones basic simple” by avoiding use of some of the fancier gmake constructs. Again the intention is to keep things simple, intuitive, readable, maintainable.

Each intermediate tree branch node contains a *Makefile.<tool>* (where *<tool>* can be *questa*, *vcs*, or *ius*) which documents the child branch nodes and/or leaf test nodes reachable from that node by defining a **$(DIRS)** macro. For example in the **$UVMC_HOME/examples/** regression tree root node, the Makefile looks like this,

```
#------------------------------------
DIRS = \
    commands \
```

```
    connections \
    converters \
    field_types \
    xlerate.connections \
    config_exts

BITS ?= 64

MAKEFILE ?= Makefile

all compile build sim check clean:
    #
    for i in $(DIRS); do \
        ../../test_drive.csh $$i $(MAKE) -f $(MAKEFILE) $@; \
    done
```

All branch node Makefile's have exactly the same look as that shown above.

Each tool specific Makefile drives the main branch node Makefile as you see here for example in *Makefile.questa*,

```
all compile build sim check clean:
    $(MAKE) MAKEFILE=Makefile.questa $@
```

So, again being fractal in nature, each tree branch node in effect defines its own miniature regression root which can be run on the spot and will recurse to all child nodes and eventually leaf tests. Simply execute the Makefile at that particular branch node and that will happen.

Notice the reference to **test_drive.csh**. This is a generic drive script invokable from C-shell (csh) script that you will see placed at each regression root node that provides a structure for setting up the required environment of each leaf test and provides an option for generating a grep'pable PASSED/FAILED report (see topic below).

The **test_drive.csh** simply has the following in it,

```
#!/bin/csh -f
cd $1;
grep -q DIRS Ma*
if ( $status == "0" ) then
    echo "=+= ----------- `pwd`"
    shift
    $*
else
    source Env.script
    shift
    echo "Test Started: `date`"
    $*
    if ( $status == "0" ) then
        echo "=+= Test PASSED `pwd`"
    else
        echo "=+= Test FAILED `pwd`"
    endif
    echo "Test Ended: `date`"
endif
```

so not much to it really. It was done that way on purpose ! The intent was to avoid an "empire of scripts" for running regressions by keeping things fractal, atomic, simple throughout.

So the **test_drive.csh** driver is what recurses through the regression tree all the way to the leaf test nodes.

No complex environment setup is required to run the Makefile's at each intermmediate tree branch node. This is because only leaf tests below the branch nodes are responsible for *self specifying* their own specific environment setup requirements via a C-shell (csh) script called **Env.script** explained below.

The only exception to the rule of "no environment setup" is that before running any of the branch node Makefile's, you need to define the following ENV var:

```
$DEMO_ROOT
```

- Be sure **$DEMO_ROOT** is set to point to a directory containing a proper **.toolsrc** file which is a master, *site-specific* tool setup script that is referenced in each Env.script. A sample .toolsrc is included in the $UVMC_HOME/lib/ area that you can customize for your particular site settings. For example,

  ```
  setenv DEMO_ROOT $UVMC_HOME/lib
  ```

- OK, now you can source your *Env.script*:

  ```
  source Env.script
  ```

So the **test_drive.csh** script simply sees the local Makefile and executes it. If it is a leaf test node, it will automatically source the **Env.script** required for that leaf test assuming the 2 basic ENV vars above have been properly set a-priori.

4.1.3 Local Env. script's

Note that the above assumes there is a local *Env.script* present for each leaf test directory. You can run the tests in that individual directory by just manually sourcing the Env.script.

See Environment setup template script in the intro chapter for details of how this setup script uses a *.toolsrc* master environment setup template.

As described above the recursive Makefile driven regression test driver deploys this Env.script to allow each test to self specify its environment but if you set the environment in your own way, you can just run the Makefile's directly as was described in the <Running a UVMC example> section.

4.1.4 Running leaf tests

You will see that each of the leaf tests are self contained with their own *Makefile.<tool>* variations (where *<tool>* can be *questa*, *vcs*, or *ius*).

Each leaf test directory also has its own **Env.script**. That is the file that must be sourced from a **PLAIN VANILLA** xterm csh and this is automatically done by the **test_driver.csh** drive script mentioned above.

Assuming you follow the same procedures described above prior to sourcing **Env.script**, you can easily "manually" run any leaf test as well.

Among the different leaf test directories you will see 3 types of Makefile's distinguished with different suffices. They have the following meanings,

```
Makefile.questa   # Mentor Graphics Questa simulator
Makefile.vcs      # Synopsys VCS simulator
Makefile.ius      # Cadence IUS simulator
```

- For example to run the any of the leaf tests in Questa mode,

```
setenv DEMO_ROOT <path to directory containing .toolsrc>
source Env.script
make -f Makefile.questa BITS=64  # Note: BITS defaults to 32
```

- Each leaf test Makefile has 5 standard targets that are used consistently throughout the entire regression tree structure,

```
all: compile build sim check clean # i.e. All of the 5 targets below.

compile:  # Analyze, synthesize HDL side
build:    # Build HVL side
sim:      # Run simulation
check:    # Check results of simulation
clean:    # Clean everything up
```

- The rules for PASS/FAIL are quite simple: if the Makefile fails the test fails, if the Makefile passes the test passes. I.e. if the Makefile can execute all 5 targets cleanly including the **check:** target without failing, then the test passes. That's it ! Again, this operation is consistent throughout all of the Makefiles you see in the entire regression tree structure.

4.1.5 Generating PASSED/ FAILED reports

Using the simple **grep** command you can easily generate a comprehensive PASSED/FAILED report.

To do this, simply follow the branch node procedures described above but just redirect all output to a log file as follows:

```
cd <any regression tree root node or branch node>
gmake -f Makefile.<tool> |& tee gmake.log
```

This will generate a full regression test report in 'gmake.log'.

To generate a nice PASSED/FAILED report, simply grep for the pattern "=+= Test" as follows:

```
grep "=+= Test" gmake.log
```

and you will get a report that looks something like this,

```
=+= Test PASSED /.../examples/commands
=+= Test PASSED /.../examples/connections
=+= Test PASSED /.../examples/converters
=+= Test PASSED /.../examples/field_types
=+= Test PASSED /.../examples/xlerate.connections
=+= Test PASSED /.../examples/config_exts
```

If you would like further guidance on a good template script that can be used for environment setups, this section details a template for environment setups that will work for all examples in included in this package and can even be used when building special target libraries, using different vendor simulators.

5 UVMC Connections

This chapter shows how to make TLM connections between SystemC and SystemVerilog UVM components.

5.1 Overview

To communicate, verification components must agree on the data they are exchanging and the interface used to exchange that data. This chapter covers how to make connections between components using standard TLM interfaces.

This chapter focuses on connections. The code to make the connections look the same regardless of the types of the TLM interfaces being connected. We do not need to show the actual types of the ports, exports, or sockets used by the models we are connecting. The only requirement is that the port types be compatible. If they are not, the C++ or SystemVerilog compiler or elaborator will let us know.

All the examples in this section use the TLM2 generic transaction type, *tlm_generic_payload* (TLM GP), for which transaction type and converters are pre-defined for you. Greater reuse and interoperability is possible for models that use the TLM GP and follow the TLM base protocol, both of which are defined in IEEE 1666-2011.

5.1.1 The Connect Function

The *connect* and *connect_hier* functions are used to register any type of TLM port, export, interface, imp, or socket for connection across the language boundary.

5.1.2 Syntax

The calling syntax for the *connect* function.

5.1.2.1 SV

```
TLM2
 uvmc_tlm #(T, CVRT)::connect (port, lookup);
 uvmc_tlm #(T, CVRT)::connect_hier (port, lookup);

TLM1:
 uvmc_tlm1 #(T, CVRT)::connect (port, lookup);
 uvmc_tlm1 #(REQ,RSP,CVRT_REQ,CVRT_RSP)::connect (port,lookup);

 uvmc_tlm1 #(T, CVRT)::connect_hier (port, lookup); TLM1 uni
 uvmc_tlm1 #(REQ,RSP,CVRT_REQ,CVRT_RSP)::connect_hier (port,lookup);
```

5.1.2.2 SC

```
TLM1 & TLM2:
 uvmc_connect (port, lookup);
 uvmc_connect <CVRT> (port, lookup);

 uvmc_connect_hier (port, lookup);
 uvmc_connect_hier <CVRT> (port, lookup);
```

5.1.3 Parameters

A description of the type parameters to *connect*.

The *connect* function is a static function accessed via a class with type parameters for *T*, the transaction type, and optional *CVRT*, a custom converter type. References to *port* in the following descriptions refer to all port types, i.e. ports, exports, sockets, etc., unless otherwise noted.

T	Specifies the transaction type for all TLM2 ports and unidirectional TLM1 ports. Required parameter for SV connections.
REQ, RSP	Specifies the request and response transaction types for bidirectional TLM1 ports. The default RSP type is the REQ type, so RSP must be specified only if different than REQ. Required parameter for SV connections.
CVRT, CVRT_REQ, CVRT_RSP	The converter policy class to use for this connection (optional). In SV, you do not need to specify a converter for transaction types that extend *uvm_object* and implement the *do_pack* and *do_unpack* methods (or use the *`uvm_field* macros). In SC, you do not need to specify a converter for transaction types that implement *do_pack* and *do_unpack* or for which you have defined a template specialization of *uvmc_converter<T>*. See Converters for how to define and use converter classes. Default: *uvmc_converter #(X)*, with X = [T, REQ, or RSP]

On the SC-side, you do not typically need to specify any type parameters. The transaction type is deduced by the C++ compiler based on the port provided, and converters are almost always a specialization of the default converter. The compiler choses any specialization over the default implementation automatically.

5.1.4 Arguments

A description of the arguments to *connect*.

port	The port, export, imp, interface, or socket instance to be connected. The port's hierarchical name will be registered as a lookup string for matching against other port registrations within both SV and SC. A string match between two registered ports results in those ports being connected.
lookup	An optional, additional lookup string to register for this port. Every UVMC connection must involve at least one usage of this optional string, as all ports have unique hierarchical names. Default: ""

5.1.4.1 Port Argument

The connect function's *port* argument is a handle to a TLM1 or TLM2 port, export, interface, imp, or socket. During elaboration, the matched port must agree on interface (e.g. put, get, peek), direction (e.g. port or export/imp), and transaction type, else the connection will fail.

For example, consider the following SV port

```
uvm_tlm_blocking_put_port #(trans)
```

This port can be connected via *connect* to instances of the following SC port types, assuming the appropriate converters exist.

```
tlm_blocking_put_if<trans>
sc_export< tlm_blocking_put_if<trans> >

tlm_put_if<trans>
sc_export< tlm_put_if<trans> >
```

The same SV port can be connected to the following SC port instances using the *connect_hier* function.

```
sc_port< tlm_blocking_put_if<trans> >
sc_port< tlm_put_if<trans> >
```

As you can see, the blocking put port in SV has several compatible connection options in SC. That's because the blocking put port requires a connection to something that provides a blocking put interface, a requirement satisfied by all the above SC-side exports and interfaces.

- The *tlm_blocking_put_if<trans>* meets this requirement exactly
- The *tlm_put_if<trans>* interface provides an implementation of both the blocking put and non-blocking put interface, so it meets the blocking interface requirement
- Each export is ultimately connected to implementations of the *tlm_blocking_put_if* or *tlm_put_if* interfaces

Derivatives of the above export and interface types are also valid connections to our blocking put port.

5.1.4.2 Lookup Argument

Lookup strings are global across both SC and SV. A lookup string can be anything you wish as long as it is unique to other UVMC connections. Just before UVM's *end_of_elaboration* phase, UVM Connect will establish the actual cross-language connection.

It is recommended that you apply a naming convention that assures the lookup strings will be unique yet do not embed hierarchical paths.

While most connections will be made by matching *lookup* strings, UVMC also captures each port's hierarchical name in each connect call. This hierarchical name can be used for matching as well.

Connecting by matching an SV port's hierarchical name

```
SV:
uvmc_tlm #()::connect(prod.out);

SC:
uvmc_connect(cons.in, "prod.out");
```

The connect call in SV omits the 2nd argument. Therefore, UVMC only registers the hierarchical name, *"prod.out"* , to represent the producer's port. The connect call on the SC side supplies a 2nd argument. Thus, UVMC registers the names *"cons.in"* and *"prod.out"* to represent the consumer's export. During elaboration, UVM Connect will match the string *"prod.out"* and make the connection between the SV producer and the SC consumer.

This approach is not recommended because you end up coupling your code to component hierarchy. If one side's hierarchy changes, your UVMC connections will need to updated. Using the global, arbitrary lookup string, while not ideal, provides better protection from hierarchy changes.

If you prefer to use hierarchical names, you will have to specify at least one hierarchical name as the lookup string.

To avoid affecting the *connect* code when paths or lookup strings need to change, consider storing the paths and lookup strings in a separate file for reading/parsing rather than hard-coding them in your code.

5.1.5 Usage

The *connect* function registers a port for UVMC connection. During elaboration, the port's hierarchical name and optional lookup name will be used to match with lookup strings of other registered ports. During operation, transactions are converted using the default converter, or using the converter type you specified in the *connect* call.

Most ports require they be connected. Registering with UVMC *connect* satisfies this requirement. However, any UVMC connection that does not end up with a match will produce a fatal error.

While native connections require a single call to *connect* (SV) or *bind* (SC), a UVMC connection requires *two* connect calls, one each for the initiator and target, each of which can be in either SC or SV.

5.1.6 SV Connections

For SV, the connect function is a static member function of a class that is parameterized to the transaction type and optional converter. The transaction type is a required parameter, whereas the converter is only required if your transaction does not implement *do_pack* and *do_unpack* (or use the *`uvm_field* macros).

5.1.6.1 Point-to-point TLM2 or analysis connection

```
uvmc_tlm #(trans)::connect(port_handle, "lookup");
```

5.1.6.2 Hierarchical TLM2 connection

```
uvmc_tlm1 #(trans)::connect_hier(port_handle, "lookup");
```

5.1.6.3 Point-to-point unidirectional TLM1 connection

```
uvmc_tlm1 #(trans)::connect(port_handle, "lookup");
```

5.1.6.4 Point-to-point bidirectional TLM1 connection

```
uvmc_tlm1 #(request, response)::connect(port_handle, "lookup");
```

5.1.6.5 Hierarchical TLM1 connection

```
uvmc_tlm1 #(trans)::connect_hier(port_handle, "lookup");
```

5.1.6.6 Notes

- These calls to connect do not specify a converter class explicitly. Therefore, the default converter will be used. See Default Converters for details.
- Connections to analysis ports and exports is made using *uvmc_tlm #(trans)::connect*, not *uvmc_tlm1 #(trans)::connect*.
- For TLM2 connections, if the *trans* type is not specified, the default *uvm_tlm_generic_payload* is used.
- You must specify the *trans* type when making TLM1 connections, as there is no default transaction type for TLM1.

5.1.7 SC Connections

UVMC connections in SystemC are made by registering any TLM port, export, interface, or socket using the *uvmc_connect* function. When calling this function, you pass in a reference to the TLM instance and an optional lookup string. During elaboration, UVMC will connect the ports whose registered lookup strings match. An error will occur if the ports are incompatible or a registered port has no match.

All UVMC TLM connections in SystemC are made with two kinds of connect calls.

5.1.7.1 Point-to-point connection (TLM2 and TLM1)

```
uvmc_connect(port_ref, "lookup");
```

5.1.7.2 Hierarchical connection (TLM2 and TLM1)

```
uvmc_connect_hier(port_ref, "lookup");
```

5.1.7.3 Notes

- SC-side connects calls typically do not specify a converter type explicitly. In most cases, you will have defined a template specialization of the default converter, which the compiler choses automatically for you. See Converters for details on defining transaction converters.
- You are not required to specify the port, interface, or transaction types because the C++ compiler will infer them by the port reference you provide to *uvmc_connect*.

5.1.8 Notes

5.1.9 One UVMC Connection per Port

A UVMC connect call can be made only once for each port, export, imp, and socket instance, but this restriction does not limit your connectivity options. For example, an SV-side *uvm_tlm_analysis_port<T>* may drive any number of SC-side analysis imps or exports. The *uvmc_connect* call on the SC side returns a reference to the proxy port that will drive the specified SC-side analysis export/interface. You may subsequently bind this proxy port to any number of other SC-side exports/interfaces.

5.1.10 SC connections without sc_main

All SC examples in this kit all define the standard *sc_main* entry point to instantiate the SC-side testbench and start SystemC. Your simulator may also support exportation of SC model definitions for direct instantiation in SystemVerilog.

The following demonstrates how to create a UVMC connection by defining and exporting a top-level *sc_module* to SystemVerilog. The SC module must be compiled and exported to a library before attempting to compile and link the SystemVerilog code.

5.1.10.1 SC side

In SystemC, you define the sc_module, then invoke the *SC_EXPORT_MODULE* macro to export it.

```
sc_top.h:
```

```
class sc_top : public sc_module
{
   public:
   target trgt;
   SC_CTOR(sc_top) : trgt("trgt")
   {
     uvmc_connect(trgt.target_socket,"foo");
   }
};

sc_top.cpp:

#include "sc_top.h"
SC_MODULE_EXPORT(sc_top)
```

5.1.10.2 SV side

In SystemVerilog, you define the top-level SV module to instantiate the SC module as if it were an SV module. The location of the compiled and exported SC module must be visible to your SV compiler.

```
class sv_env extends uvm_env;
   initiator init;
   ...
   function void connect_phase(uvm_phase phase);
     uvmc_tlm#(payload)::connect(init.socket, "foo");
   endfunction
endclass

...

module sv_top;
  // Instantiate export SC module
  sc_top sc_top_inst();
  initial begin
     sv_env env;
     env = new("env");
     run_test("test");
  end
endmodule
```

The above example shows the UVMC connection being registered in the *connect_phase* of the *sv_env* class.

5.1.11 SystemC Starting before UVM is Ready

SystemC may finish elaboration before SystemVerilog, in which case its models may start to emit transactions out its UVMC ports connections before UVM is ready to receive them. UVM Connect blocks all communication from SystemC until UVM has reached its run_phase. This means that all TLM port communication in SystemC must occur from an SystemC thread via *SC_THREAD* or *sc_spawn*.

To allow port registration to occur up through UVM's *connect_phase*, UVM's ILLCRT check is disabled to allow post-build UVMC port binding.

5.1.12 Timescales

UVM Connect's default time precision for conveying delay times in the TLM2 interfaces is 1 picosecond.

Use Questa's *-t* argument in vsim to force the time scale in both SC and SV to be the same. Refer to the documentation for how to do this in other simulators.

5.1.13 Connection Examples

This section describes how to prepare and run the connection examples including in this kit.

5.1.14 Setup

See <Getting Started> for setup requirements before running the examples. Specifically, you will need to have precompiled the UVM and UVMC libraries and set environment variables pointing to them.

5.1.15 Running

5.1.15.1 Use *make help* to view the menu of available examples

```
> make help
```

You'll get a menu similar to the following

```
 ------------------------------------------------------------------
|                 UVMC EXAMPLES - CONNECTIONS                      |
 ------------------------------------------------------------------
|                                                                  |
| Usage:                                                           |
|                                                                  |
|   make [UVM_HOME=path] [UVMC_HOME=path] <example>                |
|                                                                  |
| where <example> is one or more of:                               |
|                                                                  |
|   sv2sc        : SV producer --> SC consumer                     |
|                  Connection is made via UVMC                     |
|                                                                  |
|   sc2sv        : SC producer --> SV consumer                     |
|                  Connection is made via UVMC                     |
|                                                                  |
|   sv2sc2sv     : SV producer --> SC consumer                     |
|                  Producer and consumer send transactions to      |
|                  scoreboard for comparison                       |
|                  Connections are made via UVMC                   |
|                                                                  |
|   sc_wraps_sv  : SC producer --> SC consumer                     |
|                  Defines SC wrapper around SV model, uses        |
|                  UVMC connections inside the the wrapper to      |
|                  integrate the SV component. The wrapper         |
|                  appears as a native SC component.               |
|                  Consider integration of RTL models in SC.       |
|                                                                  |
|   sv2sv_native : SV producer --> SV consumer                     |
|                  Connection is made via standard UVM in SV       |
|                                                                  |
|   sc2sc_native : SC producer --> SC consumer                     |
|                  Connection is made via standard IEEE TLM in SC  |
|                                                                  |
|   sv2sv_uvmc   : SV producer --> SV consumer                     |
|                  Connection is made via UVMC. Semantically       |
|                  equivalent to sv2sv_native                      |
```

```
|                                                                  |
| UVM_HOME and UVMC_HOME specify the location of the source        |
| headers and macro definitions needed by the examples. You must   |
| specify their locations via UVM_HOME and UVMC_HOME environment   |
| variables or make command line options. Command line options     |
| override any envrionment variable settings.                      |
|                                                                  |
| The UVM and UVMC libraries must be compiled prior to running     |
| any example. If the libraries are not at their default location  |
| (UVMC_HOME/lib) then you must specify their location via the     |
| UVM_LIB and/or UVMC_LIB environment variables or make command    |
| line options. Make command line options take precedence.         |
|                                                                  |
| Other options:                                                   |
|                                                                  |
|   all   : Run all examples                                       |
|   clean : Remove simulation files and directories                |
|   help  : Print this help information                            |
|                                                                  |
|                                                                  |
 ------------------------------------------------------------------
```

To run just one example

```
> make sv2sc
```

This compiles and runs the *sv2sc* example, which demonstrates an SV producer sending TLM generic payload transactions to an SC consumer via TLM sockets.

The UVM source location is defined by the *UVM_HOME* environment variable, and the UVM and UVMC compiled libraries are searched at their default location, *../../lib/uvmc_lib*.

To run all examples

```
> make all
```

The *clean* target deletes all the simulation files produced from previous runs.

```
> make clean
```

You can combine targets in one command line

```
> make clean sc_wraps_sv
```

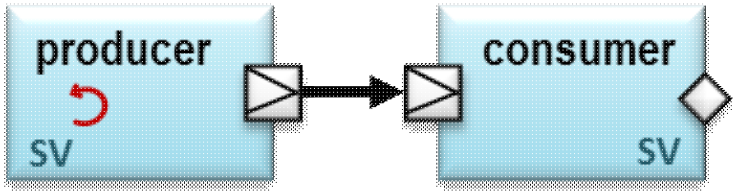
The following runs the ‘sc2sv’ example, providing the path to the UVM source and compiled library on the *make* command line.

```
> make UVM_HOME=<path> UVM_LIB=<path> sc2sv

//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
```

This chapter shows how to write converters for your transactions.
UVM Connect defines default converters in both SV and SC

6 UVMC Connection Example - Native SV to SV

This example reviews how to make a local, native TLM connections between two UVM components in pure SystemVerilog testbench. UVMC is not used. The UVMC Connection Example - UVMC-based SV to SV uses UVMC to make the same local SV connection.



The *sv_main* top-level module below creates and starts the SV portion of this example. It does the following:

- Creates an instance of a *producer* component
- Connects the producer's *out* port to the consumer's *in* port using the native UVM TLM connection.
- Calls *run_test* to start UVM simulation

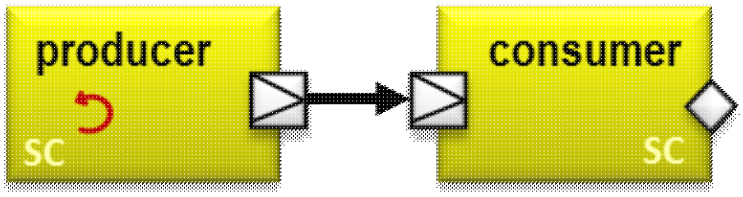
TLM connections are normally made in the *connect_phase* callback of a UVM component. This example does not show that for sake of highlighting the connect functionality.

```
import uvm_pkg::*;

`include "producer.sv"
`include "consumer.sv"

module sv_main;

  producer prod = new("prod");
  consumer cons = new("cons");

  initial begin
    prod.out.connect(cons.in);
    run_test();
  end

endmodule
```

This example shows that you can use UVMC to establish TLM connections between any two compatible components, even if they both reside in SV.

7 UVMC Connection Example - Native SC to SC

This example serves as a review for how to make 'native' TLM connections between two SystemC components (does not use UVMC).



In SystemC, a *port* is connected to an *export* or an *interface* using the port's *bind* function. An sc_module's port can also be connected to a port in a parent module, which effectively promotes the port up one level of hierarchy.

In this particular example, *sc_main* does the following

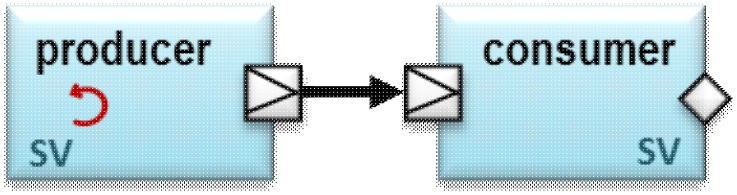
- Instantiates *producer* and *consumer* sc_modules
- Binds the producer's *out* port to the consumer's *in* export
- Calls *sc_start* to start the SystemC portion of our testbench.

The *bind* call looks the same for all port-to-export/interface connections, regardless of the port types and transaction types. The C++ compiler will let you know if you've attempted an incompatible connection.

```
#include <systemc.h>
using namespace sc_core;

#include "consumer.h"
#include "producer.h"

int sc_main(int argc, char* argv[])
{
  producer prod("prod");
  consumer cons("cons");

  prod.out.bind(cons.in);

  sc_start(-1);
  return 0;
}
```

8 UVMC Connection Example - UVMC-based SV to SV

This example shows that you can use UVMC to establish TLM connections between any two compatible components, even if they both reside in SV. This further demonstrates the principle of designing components independent of their context, i.e. how they are connected.



Connecting SV components via UVM Connect has the same overall effect as making a direct, native connection. UVM Connect recognizes that the two components both reside in SV and forwards the transaction to the connected SV consumer, avoiding the unnecessary overhead of converting to bits and back.

This code in this example is very similar to native connections. Compare this example with the UVMC Connection Example - Native SV to SV, which makes the same connection without UVMC. You might also compare this example with the UVMC Connection Example - SV to SC, SV side and UVMC Connection Example - SV to SC, SC side to see how to construct a similar testbench where the consumer is implemented in SystemC.

The *sv_main* top-level module below creates and starts the SV portion of this example. It does the following:

- Creates an instance of a *producer* component
- Registers the producer's *out* port with UVMC using the string "sv2sv". We don't specify the transaction type as a parameter to *uvmc_tlm*, so the default *uvm_tlm_generic_payload* is chosen.
- Registers the consumer's *in* export with UVMC using the same string, "sv2sv". During elaboration, UVMC will connect these ports because their lookup strings match.
- Calls *run_test* to start UVM simulation

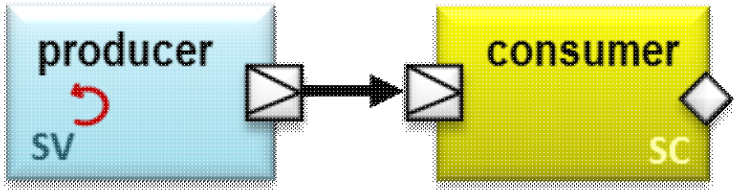
TLM connections are normally made in the *connect_phase* callback of a UVM component. This example does not show that for sake of highlighting the UVMC connect functionality.

```
import uvm_pkg::*;
import uvmc_pkg::*;

`include "producer2.sv"
`include "consumer2.sv"

module sv_main;

  producer prod = new("prod");
  consumer cons = new("cons");

  initial begin
    uvmc_tlm #()::connect(prod.out, "sv2sv");
    uvmc_tlm #()::connect(cons.in,  "sv2sv");
    run_test();
  end

endmodule
```

This example reviews how to make a local, native TLM connections between two UVM components in pure SystemVerilog testbench.
This example shows an SV producer driving an SC consumer via a TLM connection made with UVMC.
This example shows an SV producer driving an SC consumer via a TLM connection made with UVMC.

9 UVMC Connection Example - SV to SC, SV side

This example shows an SV producer driving an SC consumer via a TLM connection made with UVMC. See UVMC Connection Example - SV to SC, SC side to see the SC portion of the example.



The *sv_main* top-level module below creates and starts the SV portion of this example. It does the following:

- Creates an instance of a *producer* component
- Registers the producer's *out* socket with UVMC using the string "foo". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the consumer's *in* port on the SC side. We do not specify the transaction type, so the default *uvm_tlm_generic_payload* is used.
- Calls *run_test* to start UVM simulation

TLM connections would normally be made in the *connect_phase* callback of a UVM component. This example does not show that for sake of highlighting the UVMC connect functionality.

../../../uvmc/examples/connections/sv2sc.sv

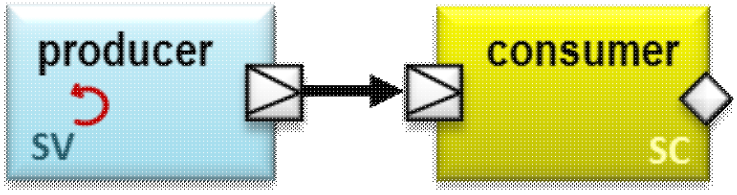
```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

import uvm_pkg::*;
import uvmc_pkg::*;

`include "producer.sv"

module sv_main;

  producer prod = new("prod");
```

```
  initial begin
    uvmc_tlm #()::connect(prod.out, "foo");
    run_test();
  end

endmodule
```

This example shows an SV producer driving an SC consumer via a TLM connection made with UVMC.

10 UVMC Connection Example - SV to SC, SC side

This example shows an SV producer driving an SC consumer via a TLM connection made with UVMC. See UVMC Connection Example - SV to SC, SV side to see the SV portion of the example.



The *sc_main* function below creates and starts the SC portion of this example. It does the following:

- Instantiates a basic *consumer*
- Registers the consumer's *in* port with UVMC using the lookup string "foo". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the producer's *out* port on the SV side.
- Calls *sc_start* to start SystemC

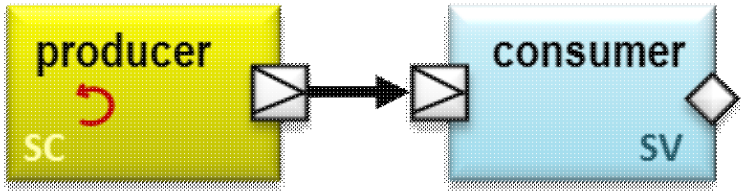
../../../uvmc/examples/connections/sv2sc.cpp

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

#include "uvmc.h"
using namespace uvmc;

#include "consumer.h"

int sc_main(int argc, char* argv[])
{
  consumer cons("cons");
  uvmc_connect(cons.in,"foo");
  sc_start(-1);
  return 0;
}
```

This example shows an SV producer driving an SC consumer via a TLM connection made with UVMC.

11 UVMC Connection Example - SC to SV, SC side

This example shows a SC producer driving a SV consumer via a TLM connection made with UVMC, including how to derive a SC producer subtype that can control UVM phasing using the UVMC Command API. See UVMC Connection Example - SC to SV, SV side to see the SV portion of the example.



In a pure SC simulation, synchronization between a producer and consumer occurs exclusively through the protocol prescribed by the TLM standard. In mixed SC-SV simulation, SC usually elaborates and starts its threads before SV has finished elaborating. To prevent run-time errors, UVMC will block any cross-language access until SV is ready. This means calls to interface methods via SC ports or sockets connected to SV must be made from within SC threads (via the SC_THREAD macro or sc_spawn).

Blocking until SV is ready technically violates TLM non-blocking semantics. It was deemed more useful to hold back activity from all cross-language calls rather than reject all non-blocking calls until SV was ready. Future releases may provide an option to return immediately with 0 status from non-blocking calls.

When a UVM testbench is sitting on the SV side, you must also consider UVM's phasing semantics, which says that a phase will end if there are no objections raised to its ending. When, as in this example, there is a SC-side participant to UVM phase control, an objection will need to be raised from the SC side for the phase(s) in which the SC side actively participates. The producer, in other words, must use the UVMC Command API to raise and drop the objection that corresponds to the phase in which it is generating stimulus. If it does not, then UVM (SV) will end that phase and likely end simulation before the SC producer has had a chance to emit the first transaction.

To preserve reuse, it is recommended that the UVM Command API usage be relegated to a subtype of the native SC producer. This way, you do not couple the producer's primary functionality (producing transactions) with cross-language synchronization issues and UVMC.

11.1 UVM-aware SC producer

This example defines the *producer_uvm* class, which derives from our generic SC *producer*. In it, we spawn a dynamic *objector* thread that calls *uvmc_raise_objection* to UVM's run phase. This keeps simulation alive on the SV side while the base *producer* in SC generates stimulus.

When the base *producer* is finished generating stimulus, it will notify a *done* sc_event. The *objector* thread in this *producer_uvm* wakes up on that event notification and drops its objection using *uvmc_drop_objection*. This allows UVM simulation to proceed to the next phase and eventually complete simulation.

```
#include "uvmc.h"
using namespace uvmc;

#include "producer.h"

class producer_uvm : public producer {
```

```
  public:

  producer_uvm(sc_module_name nm) : producer(nm) {
    SC_THREAD(objector);
  }

  SC_HAS_PROCESS(producer_uvm);

  void objector() {
    uvmc_raise_objection("run");
    wait(done);
    uvmc_drop_objection("run");
  }

};
```

11.1.1 sc_main

The *sc_main* function below creates and starts the SC portion of this example. It does the following:

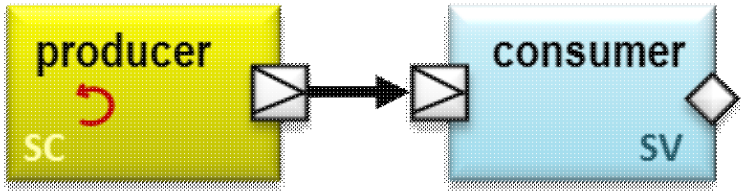
- Instantiates a basic *producer*
- Registers the producer's *in* port with UVMC using the lookup string "42". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with a consumer's *in* port on the SV side.
- Calls *sc_start* to start SystemC

```
int sc_main(int argc, char* argv[])
{
  producer_uvm prod("producer");
  uvmc_connect(prod.out,"42");
  sc_start(-1);
  return 0;
}
```

12 UVMC Connection Example - SC to SV, SV side

12.1 Description

This example shows an SC producer driving an SV consumer via a TLM connection made with UVMC. See UVMC Connection Example - SC to SV, SC side to see the SC portion of the example.



The *sv_main* top-level module below creates and starts the SV portion of this example. It does the following:

- Creates an instance of a *consumer* component
- Registers the consumer's *in* target socket with UVMC using the arbitrary string, "42". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with a producer's *in* port on the SC side.
- Calls *run_test* to start UVM simulation

TLM connections would normally be made in the *connect_phase* callback of a UVM component. This example does not show that for sake of highlighting the UVMC connect functionality.

../../../uvmc/examples/connections/sc2sv.sv

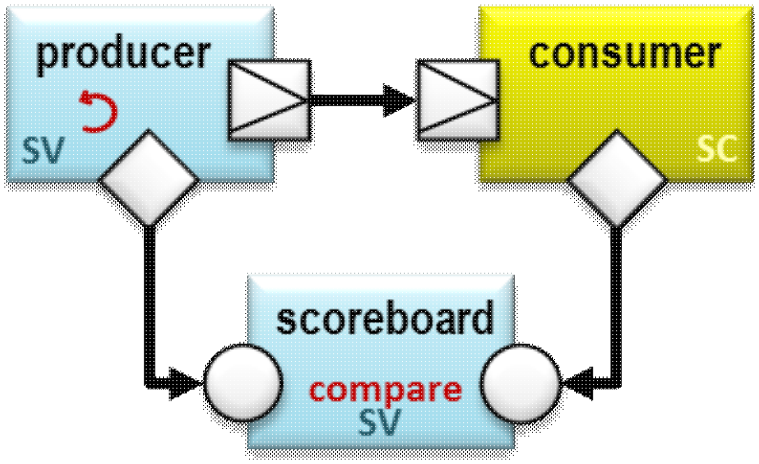
```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

import uvm_pkg::*;
import uvmc_pkg::*;

`include "consumer.sv"

module sv_main;

  consumer cons = new("cons");
```

```
  initial begin
    uvmc_tlm #()::connect(cons.in, "42");
    run_test();
  end

endmodule
```

This example shows a SC producer driving a SV consumer via a TLM connection made with UVMC, including how to derive a SC producer subtype that can control UVM phasing using the UVMC Command API.

13 UVMC Connection Example - Basic Testbench, SV side

This example shows a SV producer driving an SC consumer via a TLM2 UVMC connection, and an SC consumer sending transactions to a SV scoreboard via a TLM1 analysis connection. It also makes a local UVM analysis connection between the producer and scoreboard. See UVMC Connection Example - Basic Testbench, SC side for the SC portion of this example.



The *sv_main* top-level module below creates and starts the SV portion of this example. It does the following:

- Creates an instance of *producer* and *scoreboard* components
- Registers the producer's *out* socket with UVMC using the string "stimulus". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the consumer's *in* port on the SC side.
- Registers the scoredboard's *actual_in* analysis export with UVMC using the string "analysis". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the consumers's *ap* analysis port on the SC side.
- Connects the producer's analysis port, *ap*, to the scoreboard's *expect_in* analysis export.
- Calls *run_test* to start UVM simulation

These connections would normally be made in the *connect* method of a UVM component. This example does not show that for sake of brevity and highlighting the UVMC connect calls.

../../../uvmc/examples/connections/sv2sc2sv.sv

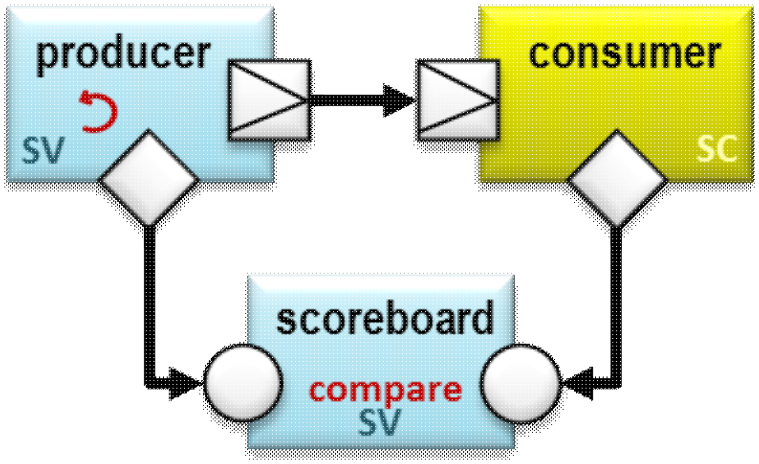
```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
```

```
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

import uvm_pkg::*;
import uvmc_pkg::*;

`include "producer.sv"
`include "scoreboard.sv"

module sv_main;

  producer prod = new("prod");
  scoreboard sb = new("sb");

  initial begin

    // normal SV-only connection
    prod.ap.connect(sb.expect_in);

    // TLM2 connection
    uvmc_tlm #()::connect(prod.out, "stimulus");

    // TLM1 connection
    uvmc_tlm1 #(uvm_tlm_generic_payload)::connect(sb.actual_in, "analysis");

    run_test();
  end

endmodule
```

This example shows a SV producer driving an SC consumer via a TLM2 UVMC connection, and an SC consumer sending transactions to a SV scoreboard via a TLM1 analysis connection.

14 UVMC Connection Example - Basic Testbench, SC side

This example shows a SV producer driving an SC consumer via a TLM2 UVMC connection, and an SC consumer sending transactions to a SV scoreboard via a TLM1 analysis connection. It also makes a local UVM analysis connection between the producer and scoreboard. See UVMC Connection Example - Basic Testbench, SV side for the SV portion of this example.



The *sc_main* function below creates and starts the SC portion of this example. It does the following:

- Instantiates a basic *consumer*
- Registers the consumer's *in* port with UVMC using the lookup string "stimulus". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the producer's *out* port on the SV side.
- Registers the consumer's *ap* port with UVMC using the lookup string "analysis". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the scoreboard's *actual_in* analysis export on the SV side.
- Calls *sc_start* to start SystemC

../../../uvmc/examples/connections/sv2sc2sv.cpp

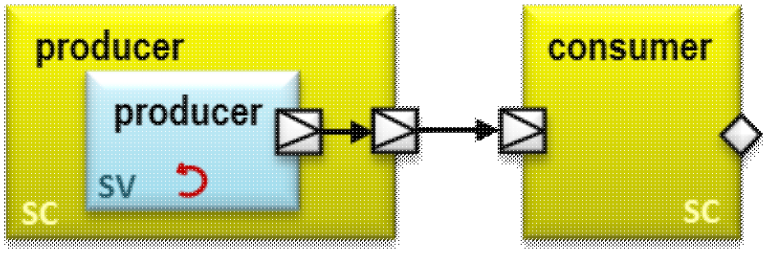
```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
```

```
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

#include "uvmc.h"
using namespace uvmc;

#include "consumer.h"

int sc_main(int argc, char* argv[])
{
  consumer cons("cons");
  uvmc_connect(cons.in,"stimulus");
  uvmc_connect(cons.ap,"analysis");
  sc_start(-1);
  return 0;
}
```

This example shows a SV producer driving an SC consumer via a TLM2 UVMC connection, and an SC consumer sending transactions to a SV scoreboard via a TLM1 analysis connection.

15 UVMC Connection Example - Hierarchical Connection, SC side

This example illustrates how to make hierarchical UVMC connections, i.e. how to promote a *port*, *export*, *imp*, or *socket* from a child component to a parent that resides in the other language. In effect, we are using a component written in SV as the implementation for a component in SC. See UVMC Connection Example - Hierarchical Connection, SV side to see the SC portion of the example.



By hiding the SV implementation, we create what appears to be a pure SC-based testbench, just like the UVMC Connection Example - Native SC to SC. However, the SC producer is implemented in SV and uses UVMC to make a behind-the-scenes connection.

This example illustates good programming principles by exposing only standard TLM interfaces to the user. The implementation of those interfaces is hidden and therefore can change (or be implemented in another language) without affecting end user code.

15.1 producer

Our *producer* module is merely a wrapper around an SV-side implementation, but users of this producer will not be aware of that. The producer does not actually instantiate a SV component. It simply promotes the socket in the SV producer to a corresponding socket in the SC producer. From the outside, the SC producer appears as an ordinary SC component with a TLM2 socket as its public interface.

```
#include "uvmc.h"
using namespace uvmc;

class producer: public sc_module
{
  public:
  tlm_initiator_socket<32> out;

  SC_CTOR(producer) : out("out") {
    uvmc_connect_hier(out,"sv_out");
  }
};
```

15.1.1 sc_main

The *sv_main* top-level module below creates and starts the SV portion of this example. It instantiates our "pure" SC producer and consumer, binds their sockets to complete the local connection, then starts SC simulation.

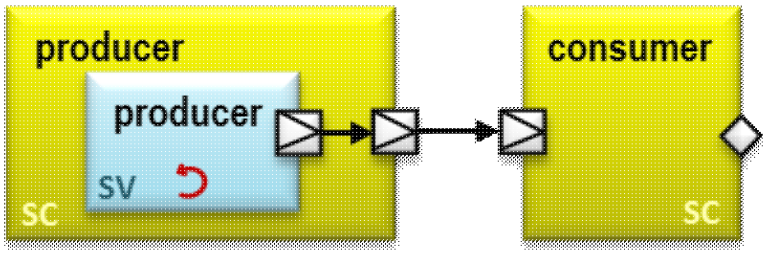
Notice that the code is identical to that used in the UVMC Connection Example - Native SC to SC example. From the SC user's perspective, there is no difference between the two testbenches. We've hidden the UVMC implementation details from the user.

```
#include "consumer.h"

int sc_main(int argc, char* argv[])
{
  producer prod("prod");
  consumer cons("cons");

  prod.out.bind(cons.in);

  sc_start(-1);
  return 0;
};
```

This example illustrates how to make hierarchical connections, i.e.
This example serves as a review for how to make 'native' TLM connections between two SystemC components (does not use UVMC).

16 UVMC Connection Example - Hierarchical Connection, SV side

This example illustrates how to make hierarchical connections, i.e. promoting a *port*, *export*, *imp*, or *socket* from a child component to its parent. See UVMC Connection Example - Hierarchical Connection, SC side to see the SC portion of the example.



In this case, we are creating a SV producer that will serve as the implementation of a producer in SC. SC users won't be aware of this because all they see is a standard SC producer. This is in keeping with the principle of encapsulation and designing to interfaces. The implementation of something can change as long as the interface and semantic doesn't change. Standard TLM interfaces enable the application of this principle for design and verification components alike.

The *sv_main* top-level module below creates and starts the SV portion of this example. It does the following:

- Creates an instance of a *producer* component
- Registers the producer's *out* socket with UVMC using the string "sv_out". During elaboration, UVMC will connect this port with a port registered with the same lookup string. In this example, the match will occur with the producer's *out* port on the SC side.
- Calls *run_test* to start UVM simulation

../../../uvmc/examples/connections/sc_wraps_sv.sv

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

`include "uvm_macros.svh"
```

```
import uvm_pkg::*;

`include "producer.sv"

module sv_main;

  import uvmc_pkg::*;

  producer prod = new("prod");

  initial begin
    uvmc_tlm #(uvm_tlm_generic_payload)::connect(prod.out,"sv_out");
    run_test();
  end

endmodule
```

This example illustrates how to make hierarchical UVMC connections, i.e.

17 UVMC Connection Common Code - SV Producer

17.1 Description

A simple SV producer TLM model that generates a configurable number of *uvm_tlm_generic_payload* transactions. The model uses the TLM2 blocking interface, whose semantic guarantees the transaction is fully completed upon return from the b_transport call. Thus, we can reuse the transaction each iterationi and need only allocate the transaction once.

This example uses the *uvm_tlm_b_initiator_socket*, which only uses the *b_transport* TLM interface. It's default transaction type is the *uvm_tlm_generic_payload*, which is what this example uses.

Normally, a monitor, not the producer, emits observed transactions through an analysis port. We use the analysis port here only to illustrate external connectivity.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM port and socket to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might be driving its *in* socket or who might be listening on its *ap* analysis port.

../../../../uvmc/examples/connections/common/producer.sv

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

import uvm_pkg::*;
`include "uvm_macros.svh"

class producer extends uvm_component;

   uvm_tlm_b_initiator_socket #() out;
   uvm_analysis_port #(uvm_tlm_gp) ap;

   `uvm_component_utils(producer)

   function new(string name, uvm_component parent=null);
```

```
      super.new(name,parent);
      out = new("out", this);
      ap = new("ap", this);
   endfunction

   task run_phase (uvm_phase phase);

      // Allocate GP once
      uvm_tlm_gp gp = new;
      uvm_tlm_time delay = new("del",1e-12);
      int num_trans = 2;

      // Keep the "run" phase from ending
      phase.raise_objection(this);

      // Get number of transactions desired (default=2)
      void'(uvm_config_db #(uvm_bitstream_t)::get(this,"","num_trans",num_trans));

      // Iterate N times, randomizing transaction, setting delay
      for (int i = 0; i <num_trans; i++) begin

        delay.set_abstime(10,1e-9);
        assert(gp.randomize() with { gp.m_byte_enable_length == 0;
                                     gp.m_length inside {[1:8]};
                                     gp.m_data.size() == m_length; } );
        `uvm_info("PRODUCER/PKT/SEND",{"\n",gp.sprint()},UVM_MEDIUM)

        out.b_transport(gp,delay);
        ap.write(gp);
      end
      #100;
      `uvm_info("PRODUCER/END_TEST",
                "Dropping objection to ending the test",UVM_LOW)
      phase.drop_objection(this);
   endtask

endclass
```

18 UVMC Connection Common Code - SC Producer

18.1 Description

A generic producer that creates *tlm_generic_payload* transactions and sends them out its *out* socket and *ap* analysis ports.

This example uses the *simple_initiator_socket*, a derivative of the TLM core class, *tlm_initiator_socket*. Unlike the *tlm_initiator_socket*, the simple socket does not require the module to inherit and implement the initiator socket interface methods. Instead, you only need to register the interfaces you actually implement, none in this example. This is what makes these sockets simple, flexible, and convenient.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM port and socket to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might be driving its *in* socket or who might be listening on its *ap* analysis port.

../../../../uvmc/examples/connections/common/producer.h

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR    //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

#ifndef PRODUCER_H
#define PRODUCER_H

#include
```

19 UVMC Connection Common Code - SV Consumer

19.1 Description

A simple SV consumer TLM model that prints received transactions (of type *uvm_tlm_generic_payload* and sends them out its *ap* analysis port.

The *in* socket exports the *tlm_blocking_transport_if* interface implemented by this consumer.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM port and imp to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might be driving its *in* socket or who might be listening on its *ap* analysis port.

../../../../uvmc/examples/connections/common/consumer.sv

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

import uvm_pkg::*;
`include "uvm_macros.svh"


class consumer extends uvm_component;

   uvm_tlm_b_target_socket #(consumer) in;
   uvm_analysis_port #(uvm_tlm_generic_payload) ap;

   `uvm_component_utils(consumer)

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      in = new("in",  this);
      ap = new("ap", this);
   endfunction

   // task called via 'in' socket
```

```
   virtual task b_transport (uvm_tlm_gp t, uvm_tlm_time delay);
     `uvm_info("CONSUMER/PKT/RECV",{"\n",t.sprint()},UVM_MEDIUM)
     #(delay.get_realtime(1ns,1e-9));
     delay.reset();
     ap.write(t);
   endtask

endclass
```

20 UVMC Connection Common Code - SC Consumer

20.1 Description

A simple SC consumer TLM model that prints received transactions (of type *tlm_generic_payload* and sends them out its *ap* analysis port.

This example uses the *simple_target_socket*, a derivative of the TLM core class, *tlm_target_socket*. Unlike the *tlm_target_socket*, the simple socket does not require the module to inherit and implement all four target socket interface methods. Instead, you only need to register the interfaces you actually implement, *b_transport* in this case. This is what makes these sockets simple, flexible, and convenient.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM port and socket to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might be driving its *in* socket or who might be listening on its *ap* analysis port.

../../../../uvmc/examples/connections/common/consumer.h

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

#include
```

21 UVMC Connection Common Code - SV Scoreboard

21.1 Description

A simple SV scoreboard TLM model that collects expected transactions from its *expect_in* analysis imp and compares them with actual transactions received from its *actual_in* analysis imp.

The model makes use of the *uvm_analysis_imp_decl* macros to allow the scoreboard to directly implement more than one analysis interface (the expected and the actual).

The *write_expect* implementation clones the incoming expect transaction because the producer may reuse the transaction object in subsequent iterations. It then prints the transaction before storing it in the internal queue.

The *write_actual* implementation also makes a clone of the incoming actual transaction. We do not do on-the-fly comparison because the corresponding expect transaction may not have arrived yet, and we can't hold onto the handle because the underlying object may be changed or reused before we've had a chance to compare it.

This approach to scoreboard design affords several benefits over use of *tlm_fifo #(T)* on the expect side:

- consumes less memory. The queue is a primitive data type, whereas the tlm_fifo is composed of several class objects including the storage (mailbox) and the TLM exports, which are themselves composed of other objects.
- the queue can be searched in cases where out-of-order arrival of actuals is permitted. The tlm_fifo can not.
- the *write_expect* and *write_actual* methods provide a means of pre-qualifying the incoming transactions before putting them into their respective queues for later comparison.

Although not done here, the scoreboard model might employ a timeout mechanism so it does not prevent the run phase from ending indefinitely.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM analysis imps to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might analysis port.

../../../../uvmc/examples/connections/common/scoreboard.sv

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   Copyright 2012 Synopsys, Inc                             //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
```

```
//                                                              //
//   Unless required by applicable law or agreed to in          //
//   writing, software distributed under the License is         //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR      //
//   CONDITIONS OF ANY KIND, either express or implied.  See    //
//   the License for the specific language governing            //
//   permissions and limitations under the License.             //
//--------------------------------------------------------------//

import uvm_pkg::*;
`include "uvm_macros.svh"

`uvm_analysis_imp_decl(_expect)
`uvm_analysis_imp_decl(_actual)

class scoreboard extends uvm_component;

   uvm_tlm_gp qe[$];
   uvm_tlm_gp qa[$];

   uvm_analysis_imp_expect #(uvm_tlm_gp,scoreboard) expect_in;
   uvm_analysis_imp_actual #(uvm_tlm_gp,scoreboard) actual_in;

   `uvm_component_utils(scoreboard)

   uvm_comparer comparer;
   bit raised_bit;
   uvm_phase run_ph;

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      expect_in = new("expect_in", this);
      actual_in = new("actual_in", this);
      run_ph = uvm_run_phase::get();
   endfunction : new

   function void build_phase(uvm_phase phase);
      if (comparer == null)
        comparer = new;
      comparer.show_max = 100;
   endfunction

   virtual function void write_expect(uvm_tlm_gp t);
     uvm_tlm_gp t_copy;
     $cast(t_copy,t.clone());
     `uvm_info("SB/EXPECT/RECV",{"\n",t.sprint()},UVM_MEDIUM)
     qe.push_back(t_copy);
     if (!raised_bit) begin
       run_ph.raise_objection(this,"expect received, waiting for actual");
       raised_bit = 1;
     end
   endfunction

   virtual function void write_actual(uvm_tlm_gp t);
     uvm_tlm_gp t_copy;
     $cast(t_copy,t.clone());
     `uvm_info("SB/ACTUAL/RECV",{"\n",t.sprint()},UVM_MEDIUM)
     qa.push_back(t_copy);
     if (!raised_bit) begin
       run_ph.raise_objection(this,"actual received, waiting for expect");
       raised_bit = 1;
```

```
    end
  endfunction

  virtual task run_phase(uvm_phase phase);
    uvm_tlm_gp e,a;

    // wait for both sides to deliver
    forever begin
      @(qa.size() && qe.size());

      e = qe.pop_front();
      a = qa.pop_front();

      if (!a.compare(e))
        `uvm_error("SB/MISCOMPARE",
          $sformatf("%m: There were %0d miscompares:\nexpect=%s\nactual=%s",
            comparer.result,e.sprint(),a.sprint()))

      if (raised_bit && !qa.size() && !qe.size()) begin
        phase.drop_objection(this,"all packets matched; queues are empty");
        raised_bit = 0;
      end

    end

  endtask

endclass
```

22 Converters

This chapter shows how to write converters for your transactions. Converters facilitate data exchange between components residing in different languages.

If components were written in a common language, you could guarantee they exchanged compatible data simply by requiring they use the same transaction type. Such components are *strongly typed*. Any mismatch in type would be caught by the compiler.

Now let's say two components were developed such that each agreed more or less on the content of the transaction, but this time their transaction definitions were of different types. This condition always the case between components written in two different languages--they cannot possibly share a common transaction definition. To get such components talking to each other requires an adapter, or converter, that translates between the transaction types defined in each language.

22.1 Got Transactions?

UVM Connect imposes very few requirements on the transaction types being conveyed between TLM models in SV and SC, a critical requirement for enabling reuse of existing IP. The more restrictions imposed on the transaction type, the more difficult it will be to reuse the models that use them.

- No base classes required. It is not required that a transaction inherit from any base class to facilitate its conversion--in either SV or SC. The converter is ultimately responsible for all aspects of packing and unpacking the transaction object, and it can be implemented separately from the transaction proper.
- No factory registration required. It is not required that the transaction register with a factory--via a *`uvm_object_utils* macro inside the transaction definition or by any other means.
- It is not required that the transaction provide conversion methods. The default converter used in SV will expect the transaction type to implement the UVM pack/unpack API, but you can specify a different converter for each or every UVMC connection you make. Your converter class may opt to do the conversion directly, or it can delegate to any other entity capable of performing the operation.
- It is not required that the members (properties) of the transaction classes in both languages be of equal number, equivalent type, and declaration order. The converter can adapt disparate transaction definitions at the same time it serializes the data. The following are valid and compatible UVM Connect transaction definitions, assuming a properly coded converter:

```
SV                                SC
class C;                              struct C {
  cmd_t cmd;                            long addr;
  shortint unsigned address;            vector<char> data;
  int payload[MAX_LEN];                 bool write;
endclass                              };
```

- In UVM SV, it is required that the transaction class constructor not define any required arguments. It may have arguments, but they all must have default values. The first constructor argument must be ~string name=””~.

22.1.1 Do You Need a Converter?

If your models exchange non-extended TLM Generic Payload transactions, you do not need to concern yourself with transaction or converter definition. TLM GP definitions and converters are provided by the

libraries, so you may skip this section.

22.1.2 Easy When You Need Them

To enable non-TLM GP object transfer across the SV-SC boundary, you must define converters in both languages; UVMC makes this an easy process.

Defining a converter involves implementing two functions--*do_pack* and *do_unpack*--either inside your transaction definition or in a separate converter class. Although the means of conversion are similar between SV and SC, differences in these languages capabilities cause differences in conversion.

The following sections describe several options available to you for writing converters in SV and SC.

22.1.3 SV Conversion options

Here, we enumerate three different ways to define conversion functionality for your transaction type in SV.

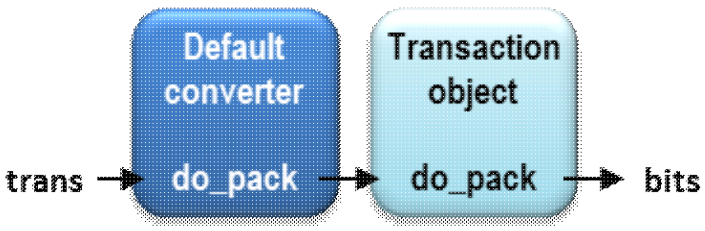
We illustrate each of these options using the following packet definition.

22.1.3.1 SV Transaction

```
class packet extends uvm_sequence_item;

  typedef enum { WRITE, READ, NOOP } cmd_t;

  `uvm_object_utils(packet)

  rand cmd_t cmd;
  rand int   addr;
  rand byte  data[$];
  ...

endclass
```

22.1.4 In-Transaction

This approach defines the conversion algorithm in transaction class itself.



A transaction in UVM is derives from *uvm_sequence_item*, which defines the *do_pack* and *do_unpack* virtual methods for users to implement the conversion functionality. This option is the recommended option for SV-based transactions.

To use this approach, you declare and define overrides for the virtual *do_pack* and *do_unpack* methods in your transaction class.

```
virtual function void do_pack   (uvmc_packer packer);
virtual function void do_unpack (uvmc_packer packer);
```

Most transactions in SV should be defined this way, as it is prescribed by UVM and it works with UVM Connect's SV default converter. See Default Converters for details.

The following SV packet definition implements the *do_pack* and *do_unpack* methods using a set of small utility macros included in the UVM. These macros implement the same packing functionality as using the *packer* API or *`uvm_field* macros, but are more efficient. See UVMC Converter Example - SV In-Transaction for a complete example of using this approach.

```
class packet extends uvm_sequence_item;

  typedef enum { WRITE, READ, NOOP } cmd_t;

  `uvm_object_utils(packet)

  rand cmd_t cmd;
  rand int   addr;
  rand byte  data[$];

  constraint C_data_size { data.size inside {[1:16]}; }

  function new(string name="");
    super.new(name);
  endfunction

  virtual function void do_pack(uvm_packer packer);
    super.do_pack(packer);
    `uvm_pack_enum(cmd)
    `uvm_pack_int(addr)
    `uvm_pack_queue(data)
  endfunction

  virtual function void do_unpack(uvm_packer packer);
    super.do_unpack(packer);
    `uvm_unpack_enum(cmd,cmd_t)
    `uvm_unpack_int(addr)
    `uvm_unpack_queue(data)
  endfunction

  ...

endclass
```

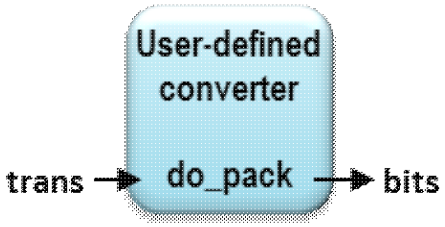
When implementing the *do_pack* and *do_unpack* methods, you may call various methods of the provided *packer* object, or you can use a set of small, more efficient convenience macros, also provided in UVM.

The following packs an *addr* field using each approach. Our example above uses the small-macro approach.

```
virtual function void do_pack(uvm_packer packer);

   `uvm_pack_int(addr)  -or-
    packer.pack_field(addr,$bits(addr)); //more overhead

endfunction
```

22.1.5 Converter Class

For transactions not extending *uvm_sequence_item*, you can define a separate converter class extending *uvmc_converter #(T)*. You then specify this converter type when calling The Connect Function.



The following SV packet definition implements the *do_pack* and *do_pack* methods required of any custom converter. The same small macros used in the In-Transaction approach can be used in this approach. See UVMC Converter Example - SV Converter Class for a complete example.

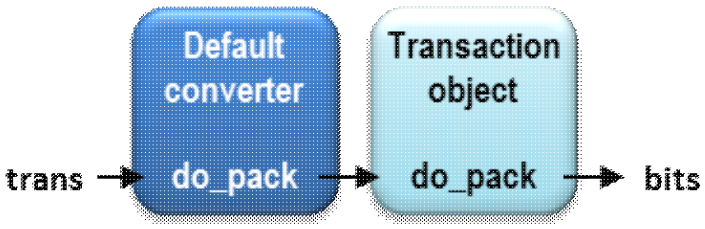
```
class convert_packet extends uvmc_converter #(packet);

   static function void do_pack(packet t, uvm_packer packer);
     `uvm_pack_enum(t.cmd)
     `uvm_pack_int(t.addr)
     `uvm_pack_queue(t.data)
   endfunction

   static function void do_unpack(packet t, uvm_packer packer);
     `uvm_unpack_enum(t.cmd,packet::cmd_t)
     `uvm_unpack_int(t.addr)
     `uvm_unpack_queue(t.data)
   endfunction

 endclass
```

To use a custom converter for a specific TLM connection, specify its type when making connection using The Connect Function.

```
uvmc_tlm #(packet, convert_packet)::connect( ... );
```

22.1.6 Field Macros

This approach defines in-transaction conversion via *`uvm_field* macro invocations.



While this approach also works with UVMC's default converter, it is far less desirable than the first In-Transaction option. The *`uvm_field* macros provide a convenient means of implementing the *do* methods of *uvm_object* for most data types, but they have recurring run-time costs and should be avoided, especially if

your transaction is slated for widespread reuse, as with VIP-related transactions.

Beause this approach is also compatible with the default converter UVMC uses, so you will not need to specify the converter type when making connections with The Connect Function. See UVMC Converter Example - SV In-Transaction via Field Macros for a complete example of using this approach.

```
class packet extends uvm_sequence_item;

  rand cmd_t cmd;
  rand int unsigned addr;
  rand byte data[$];

  constraint C_data_size { data.size inside {[1:16]}; }

  `uvm_object_utils_begin(packet)
    `uvm_field_int(cmd,UVM_ALL_ON)
    `uvm_field_int(addr,UVM_ALL_ON)
    `uvm_field_queue_int(data,UVM_ALL_ON)
  `uvm_object_utils_end

  function new(string name="");
    super.new(name);
  endfunction

endclass
```

While more succinct than our In-Transaction recommendation, we prefer optimizing for recurring run-time performance over one-time coding convenience. The macros' run-time performance is inferior, which affects every user of your transaction class in every simulation. Even small performance differences can be magnified and significantly affect the upper bound on performance speed-up with accelleration or emulation hardware. And, as stated before, post-macro-expansion yields hundreds more lines of code compared to implementations that do not employ the *`uvm_field* macros. You may eventually have to wade through this code during your debug sessions. See http://verificationacademy.com- /uvm-ovm- /MacroCostBenefit for more detail on the topic of macro usage in UVM.

22.1.7 SC Conversion Options

Conversion of the transaction type in SC can be defined in at least four ways.

We illustrate each of these options using the following packet definition.

22.1.7.1 SC Transaction

```
class packet {
  public:
  enum cmd_t { WRITE=0, READ, NOOP };
  cmd_t cmd;
  int addr;
  vector<char> data;
};
```

This transaction has no base class, no methods for packing or unpacking, no macros, etc. It is a simple container of data representing a bus transaction.

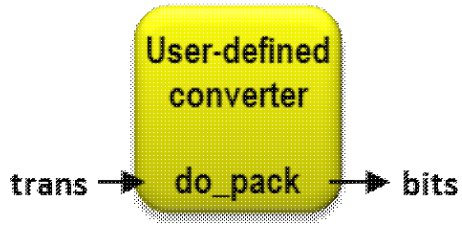
With SystemC, you are more easily able to decouple the transaction data from the algorithms that operate on the data. The SC transaction and converter definitions are typically in separate classes. Existing transaction definitions in SC can be used in a UVM Connect context by defining converters for them.

An external converter class requires that all transaction fields be public or have public accessor member functions. The *friend* construct in C++ lets you circumvent this protection, but it is not recommended.

22.1.8 Converter Specialization

Define a separate class for converting your transaction type.

In SC, conversion for a transaction is typically defined in a separate class called a *template specialization*. C++ allows you to specialize the default converter implementation for a specific transaction type, *T*.



The converter specialization for our *packet* type can be defined as follows

```
#include "uvmc.h"
using namespace uvmc;

template <>
class uvmc_converter<packet> {
  public:
  virtual void do_pack(const packet &t,
                       uvmc_packer &packer) {
    packer << t.cmd << t.addr << t.data;
  }
  virtual void do_unpack(packet &t,
                         uvmc_packer &packer) {
    packer >> t.cmd >> t.addr >> t.data;
  }
};
```

When implementing the converter's *do_pack* and *do_unpack* functions, you stream your transaction members to and from the *packer* variable, an instance of *uvmc_packer* that is inherited from an internal base class.

To pack, you stream the fields into the *packer*

```
packer << t.cmd << t.addr << t.data;
```

To unpack, you stream the fields from the *packer*

```
packer >> t.cmd >> t.addr >> t.data;
```

You can stream all the fields with one statement as shown above, or stream in separate statements, perhaps with some conditional and other code in between.

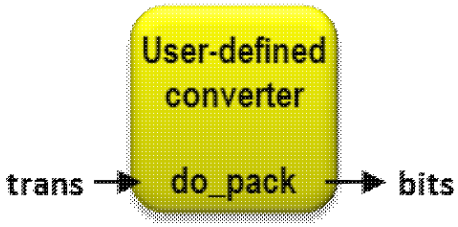
```
packer << t.cmd;
...
packer << t.addr;
...
packer << t.data;
```

With this approach, you will not need to specify the *CVRT* type parameter when calling The Connect Function. Your converter is automatically chosen by the compiler as an override (specialization) of the default converter. See UVMC Converter Example - SC Converter Class for a complete example of using this approach.

See Converter Specialization, Macro-Generated, next, for an approach that auto-generates the above converter specialization.

22.1.9 Converter Specialization, Macro-Generated

Invoke a convenience macro that defines the converter specialization for you.



The easist way to define a converter in SC is to invoke one of the UVMC_UTILS macros. Using this option, our conversion class definition reduces to:

```
#include "uvmc.h"
using namespace uvmc;

UVMC_UTILS_3 (packet,cmd,addr,data)
```

That's it. The *UVMC_UTILS_3* macro expands into the converter specialization defined in Converter Specialization, exactly. These macros are the "good" macros; they expand into efficient, readable code exactly as you would write it. See UVMC Converter Example - SC Converter Class, Macro-Generated for a complete example of using this approach.

Keep the following in mind when using the UTILS macros

- The number suffix in the macro name indicates the number of members of the class being converted. UVMC supports up to 20 field members, i.e. up to UVMC_UTILS_20.
- The macros only support types for which the *uvmc_packer* defines the << and >> operators. These include all C++ built-in types, strings, vectors, maps, and the SC types *sc_bit*, *sc_logic*, *sc_bv<N>*, *sc_lv<N>*, *sc_int<N>*, *sc_uint<N>*, *sc_unsigned<N>*, and *sc_time*. See UVMC Type Support for details.
- The UTILS macros also define the *operator<<* to the output stream for your transaction. This allows you to stream your transaction contents to *cout* and other output streams.

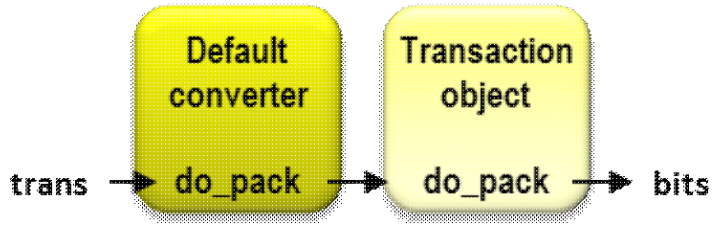
```
packet p;
...
cout << "Packet p contents are: " << p << endl;
```

- The UTILS definition of *operator<<* to the output stream may interfere with other functions that define *operator<<(&ostream)*. In this case, use a UVMC_CONVERT macro instead of the UVMC_UTILS macro. The convert macros only define the converter specialization. They also support up to 20 field members.
- All fields named in the UTILS macro invocation must be public members of the transaction. If they are not, and these members have accessor functions, you can still define an external converter via Converter Specialization without the macros.

22.1.10 In-Transaction - SC

Define *do_pack* and *do_unpack* methods in the SC transaction itself.

Although this option works with SC's default converter, it is not recommended because it hard-codes the transaction to a particular packing and unpacking algorithm. Keeping the conversion functionality separate allows you to apply different conversion algorithms without having to modify or derive new transaction subtypes.

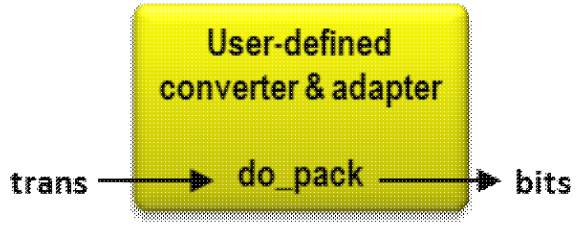


The following *packet* transaction definition provides both the content and conversion routines for the transaction. Because it is compatible with the default SC-side converter, you will not be required to specify converter class when connection with The Connect Function. See UVMC Converter Example - SC In-Transaction for a complete example of using this approach.

```
class packet;

  public:
  enum cmd_t { WRITE=0, READ, NOOP };

  cmd_t cmd;
  int addr;
  vector<char> data;

  virtual void do_pack(uvmc_packer &packer) const {
    packer << t.cmd << t.addr << t.data;
  }
  virtual void do_unpack(uvmc_packer &packer) {
    packer >> t.cmd >> t.addr >> t.data;
  }
};
```

22.1.11 Custom Adaptor

Define a custom converter for a transaction whose members differ in number, type, size, and declaration order from the corresponding transaction definition in the other language.



With UVMC support for a separate converter class, you are not limited to member-by-member, bit-compatible packing and unpacking. The only requirement is that the two sides' conversion routines agree on what and how data are represented in the bits being sent across the language boundary.

Thus, an array of four *bytes* in SV can be converted to a single *unsigned int* in SC. A single *longint* in SV can be mapped to many possible combinations of the SC integrals: *int[2]*, *char[8]*, *vector<char>*, *sc_bit<64>*, *sc_int<64>*, etc. See UVMC Converter Example - SC Adapter Class for a complete example of using a "full-custom" approach to transaction conversion.

22.1.12 Notes

22.1.13 Type Support

UVM Connect supports most of the built in types, arrays, and even sub-objects as properties of your transaction class.

The *uvmc_packer* supports packing and unpacking the following types

- bool
- char, unsigned char
- short, unsigned short
- int, unsigned int
- long, unsigned long
- long long, unsigned long long
- float
- double
- sc_bit
- sc_logic
- sc_bv<N>
- sc_int<N>
- sc_uint<N>
- sc_bigint<N>
- sc_biguint<N>
- enums
- T[N], where T is one of the above types
- vector<T>, where T is one of the above types
- list<T>, where T is one of the above types
- map<KEY,T>, where KEY and T are among the above types

See UVMC Type Support for more detail and examples.

22.1.14 On (not) using `uvm_field macros

The *`uvm_field* macros hurt run-time performance, can make debug more difficult, and can not accomodate custom behaviors, for example, conditional packing based on the value of another field.

In UVM 1.1a and prior, *uvm_tlm_generic_payload* uses the *`uvm_field* macros. Its definition expands into almost *600* lines of code, and it's wrong. The data and data_enable arrays should be packed according to the length and *byte_enable_length* fields, but the field macros do not accommodate this. They pack the entire data and byte_enable buffers, even if one one byte is valid. Replacing the `uvm_field macros with *do_* method implementations is planned for UVM 1.1b. See http://verificationacademy.com- /uvm-ovm-/MacroCostBenefit for more on macro usage in UVM.

22.1.15 Packing Algorithm

To pass an object across the language boundary, UVM Connect first calls *converter.do_pack*, which serializes the transaction contents to a simple bit-vector-like form. Upon return, your transaction's canonical representation will be stored inside the converter. UVM Connect retrieves and passes this canonical data across the language boundary using standard DPI-C.

On the target side, UVM Connect will unpack the canonical data into a new transaction object in the other language. To do this, UVM Connect first loads the data into the target-side converter. It then creates a new corresponding transaction object and passes it to converter.do_unpack, which does the reverse operation as pack. The converter unpacks the canonical data into the new transaction object, effectively reconstituting the original transaction object in the other language. This resulting transaction is then sent to the connected TLM target.

On the SV side, the default converter's implementations of *do_pack* and *do_unpack* delegate the work to the pack and unpack methods of your *uvm_object*-based transaction. If your transaction is not based on *uvm_object* (or *uvm_sequence_item*), or if your transaction object does not implement *do_pack* and *do_unpack*, you must define a converter for that transaction.

Like the SV side, the default converter on the SC side delegates to *do_pack* and *do_unpack* methods of the transaction object, *T*. SC-side transactions typically do not implement a UVM-compatible pack/unpack interface. In most cases, you will need to define a converter for each transaction type in SC. Fortunately, the UVM Connect library makes this very easy.

22.1.16 Conversion on the return path

TLM2 communication is pass-by-reference, which we emulate in UVM Connect by copying back changes to the original transaction object upon return from every interface method call.

TLM1 communication is a pass-by-value message passing mechanism, so no conversion back to the original request object is done on the return path.

22.1.17 Deletion on the return path

TLM2 rules mandate the same transaction object be used until the transaction execution is fully completed. This improves run-time efficiency.

For TLM2 blocking transport, the transaction is complete upon return, so it can be reused immediately in a subsequent call. The target should not retain a reference to any transaction object for this reason. It must copy the transaction before returning.

TLM2 non-blocking communication typically involves multiple calls back and forth between initiator and target. During this time, the same transaction object and its proxy on the other side are used throughout the call sequence. The target-side proxy object is not deleted (SC) or left for garbage collection (SV) until the returned status is TLM_COMPLETED.

TLM1 communication is a pass-by-value message passing mechanism, so the proxy transaction object on the target side is deleted or left for garbage collection upon return from the target.

22.1.18 Default Converters

UVM Connect defines default converters in both SV and SC

All converters define *do_pack* and *do_unpack* static methods. UVMC calls upon these methods to convert an object into a form that can be transferred across the language boundary. UVMC defines default implementations of the converter, one each for SV and SC.

22.1.19 Default SV Converter

The default converter on the SV side is designed to work with *uvm_object*-based UVM transactions. It delegates the actual work to the *pack* and *unpack* methods in the transaction, which in turn call the virtual user-defined methods, *do_pack* and *do_unpack*.

```
class uvmc_default_converter #(type T=int)
                      extends uvmc_converter #(T);

 static function void do_pack(T t, uvm_packer packer);
    t.pack(packer); // calls t.do_pack
 endfunction

 static function void do_unpack(T t, uvm_packer packer);
    t.unpack(packer); // calls t.do_unpack
  endfunction

endclass
```

Our *packet* definition for the In-Transaction approach is compatible with the default SV converter. This converter, as well as any custom converters you may define, are required to extend from *uvmc_converter #(your_trans_type)*.

22.1.20 Default SC Converter

The default converter on the SC side is meant to mirror the default in SV--it delegates to *do_pack* and *do_unpack* methods of your transaction type.

```
template <typename T>
class uvmc_converter {
  public:

  static void do_pack(const T &t, uvmc_packer &packer) {
```

```
    t.do_pack(packer);
  }

  static void do_unpack(T &t, uvmc_packer &packer) {
    t.do_unpack(packer);
  }
};
```

Most SC transaction definitions won't use this default converter. Instead, a Converter Specialization is defined to handle your specific transaction type. The C++ compiler will then automatically choose your specialized definition over the default converter.

22.1.21 Converter Parameters and Methods

The following describes the type parameters and methods of the converter class.

22.1.21.1 Parameters

T The transaction type to be converted. The default converters requires T to provide the packing and unpacking functionality in *do_pack* and *do_unpack* methods.

22.1.21.2 Methods

do_pack Packs the given object of type T. The default implementation in SV requires T be derived from uvm_object, whose *do_pack* implementation or *`uvm_field* macros provide the packing functionality. The default implementation in SC requires T implement a *do_pack* method with the following prototype:

```
void pack void do_pack (uvmc_packer &packer) const;
```

do_unpack Unpacks into given object of type T. The default implementation in SV requires T be derived from uvm_object, whose do_unpack implementation or `uvm_field macros provide the unpacking functionality. The default implementation in SC requires T implement an *do_unpack* method with the following prototype:

```
void do_unpack (uvmc_packer &packer);
```

22.1.22 Converter Examples

The directory *UVMC_HOME/examples/converters* contains several examples of transaction conversion in both SystemC (SC) and SystemVerilog (SV)

How a transaction is converted on one side does not effect your options on the other side. With four ways to convert a transaction in SC and three ways to do this in SV, there are a total of 12 combinations. We provide an example for each of these 12 combinations.

See <Getting Started> for setup requirements for running the examples. Specifically, you will need to have precompiled the UVM and UVMC libraries and set environment variables pointing to them.

Use *make help* to view the menu of available examples

```
> make help
```

You'll get a menu similar to the following

```
 ------------------------------------------------------------------
|                  UVMC EXAMPLES - CONVERTERS                      |
 ------------------------------------------------------------------
|                                                                  |
| Usage:                                                           |
|                                                                  |
|   make [UVM_HOME=path] [UVMC_HOME=path] <example>                |
|                                                                  |
| where <example> is one or more of:                               |
|                                                                  |
|   ex01 : SV conversion done in UVM transaction                   |
|          SC conversion done in macro-generated converter class   |
|                                                                  |
|   ex02 : SV conversion done in UVM transaction                   |
|          SC conversion done in separate converter class          |
|                                                                  |
|   ex03 : SV conversion done in UVM transaction                   |
|          SC conversion done in transaction                       |
|                                                                  |
|   ex04 : SV conversion done in UVM transaction via field macros  |
|          SC conversion done in macro-generated converter class   |
|                                                                  |
|   ex05 : SV conversion done in UVM transaction via field macros  |
|          SC conversion done in separate converter class          |
|                                                                  |
|   ex06 : SV conversion done in UVM transaction via field macros  |
|          SC conversion done in transaction                       |
|                                                                  |
|   ex07 : SV conversion done in separate converter class;         |
|                  transaction is not based on uvm_object          |
|          SC conversion done in macro-generated converter class   |
|                                                                  |
|   ex08 : SV conversion done in separate converter class;         |
|                  transaction is not based on uvm_object          |
|          SC conversion done in separate converter class          |
|                                                                  |
|   ex09 : SV conversion done in separate converter class;         |
|                  transaction is not based on uvm_object          |
|          SC conversion done in transaction                       |
|                                                                  |
|   ex10 : SV conversion done in UVM transaction                   |
|          SC-side implements converter that converts and adapts   |
|                  to an otherwise incompatible transaction type   |
|                                                                  |
|   ex11 : SV conversion done in UVM transaction via field macros  |
|          SC-side implements converter that converts and adapts   |
|                  to an otherwise incompatible transaction type   |
|                                                                  |
|   ex12 : SV-side implements converter in separate class;         |
|                  transaction is not based on uvm_object          |
|          SC-side implements converter that converts and adapts   |
|                  to an otherwise incompatible transaction type   |
|                                                                  |
| UVM_HOME and UVMC_HOME specify the location of the source        |
| headers and macro definitions needed by the examples. You must   |
| specify their locations via UVM_HOME and UVMC_HOME environment   |
| variables or make command line options. Command line options     |
| override any envrionment variable settings.                      |
|                                                                  |
| The UVM and UVMC libraries must be compiled prior to running     |
| any example. If the libraries are not at their default location  |
```

```
| (UVMC_HOME/lib) then you must specify their location via the    |
| UVM_LIB and/or UVMC_LIB environment variables or make command   |
| line options. Make command line options take precedence.        |
|                                                                 |
| Other options:                                                  |
|                                                                 |
|   all   : Run all examples                                      |
|   clean : Remove simulation files and directories               |
|   help  : Print this help information                           |
|                                                                 |
|                                                                 |
 -----------------------------------------------------------------
```

To run just one example

```
> make ex01
```

This compiles and runs Example 1, which demonstrates the recommended converter implementation option in both SC and SV. The UVM source location is defined by the *UVM_HOME* environment variable, and the UVM and UVMC compiled libraries are searched at their default location, *../../lib/uvmc_lib*.

To run all examples

```
> make all
```

The *clean* target deletes all the simulation files produced from previous runs.

```
> make clean
```

You can combine targets in one command line

```
> make clean ex03
```

The following runs the ‘ex10’ example, providing the path to the UVM source and compiled library on the *make* command line.

```
> make UVM_HOME=<path> UVM_LIB=<path> ex10

//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR    //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//
```

UVM Connect defines default converters in both SV and SC
This example shows how to implement the conversion routines in UVM-style transaction in the virtual *do_pack* and *do_unpack* functions inherited from the *uvm_object* base class.
The *connect* and *connect_hier* functions are used to register any type of TLM port, export, interface, imp, or socket for connection across the language boundary.
This approach defines the conversion algorithm in transaction class itself.
This example demonstrates how to define a custom converter for a transaction class that does not extend from *ovm_object*.
This example shows a UVM-style transaction that uses the `uvm_field macros to implement the required conversion functionality.
This example demonstrates how to define an external converter class for a given transaction type.
Invoke a convenience macro that defines the converter specialization for you.
Generate both a converter specialization and output stream *operator*<< for the given transaction *TYPE*.
Define a separate class for converting your transaction type.
This example demonstrates how to define an external converter for a transaction class using a UVMC_UTILS macro.
The members of your transaction definitions may be any collection of the following types, which have direct support in UVMC.
Generate a converter specialization of *uvmc_convert*<*T*> for the given transaction *TYPE*.
This example's packet class defines *do_pack* and *do_unpack* methods that are compatible with the default converter in SC.
This example demonstrates how to define a custom converter for a transaction class whose members differ in number, type, and size from the corresponding transaction definition in SV.

23 SC Macros

UVMC defines simple convenience macros for generating converter definitions and output stream operator (*operator<<(ostream&)*) so that you may use *cout* to print the contents of your SC transactions. These macros do not present any performance or debug difficulties beyond the very nature of their being macros. The code they expand into would be identical to code you would write yourself.

These macros are completely optional. You are encouraged to learn how to write converters and operator<<, perhaps using the macro definitions as templates to get you started.

23.1 UVMC_CONVERT

Generate a converter specialization of *uvmc_convert<T>* for the given transaction *TYPE*.

```
UVMC_CONVERT_N (TYPE, <list of N variables> )
UVMC_CONVERT_EXT_N (TYPE, BASE, <list of N variables> )
```

For the second form, the generated converter will pack/unpack the members in the provided *BASE* class before those in *TYPE*.

Invoke the macro whose numeric suffix equals the number of field members you wish to include in the pack, unpack, and print operations. These must all appear in the list of macro arguments in the order you want them streamed.

23.1.1 Example

```
UVMC_CONVERT_3( bus_trans, cmd, addr, data)
UVMC_CONVERT_EXT_1 (bus_error, bus_trans, crc)
```

The first macro generates a converter for the *bus_trans* class. Three member variables of *bus_trans* are included in the pack and unpack operations, in the order given: *cmd*, *addr*, and *data*. If there were other members in *bus_trans*, they will not be included in the converter implementations.

The second macro generates a converter and *operator<<* for the *bus_error* class. Packing, unpacking, and output streaming for the base type, *bus_trans* is performed first, followed by the *crc* field in *bus_error*.

The macros above generate the following code

```
template <>
class uvmc_converter<bus_trans> {
  public:
  static void do_pack(const bus_trans &t, uvmc_packer &packer) {
    packer << cmd << addr << data;
  }
  static void do_unpack(bus_trans &t, uvmc_packer &packer) {
    packer >> cmd >> addr >> data;
  }
};

template <>
class uvmc_converter<bus_error> {
  public:
  static void do_pack(const bus_error &t, uvmc_packer &packer) {
```

```
    uvmc_converter<base_trans>::do_pack(t,packer);
    packer << crc;
  }
  static void do_unpack(bus_error &t, uvmc_packer &packer) {
    uvmc_converter<base_trans>::do_unpack(t,packer);
    packer >> crc;
  }
};
```

23.1.1.1 Usage notes

- The default converter delegates to *T.do_pack* and *T.do_unpack*. For transactions that do not possess these member functions (likely most), use one of these macros to generate a simple converter class that packs and unpacks your transaction from outside your transaction class.
- All class members given in the list must be public members.
- The *uvmc_packer* must provide *operator*>> and *operator*<< for all the types passed in the list. See UVMC Type Support for a list of supported types.
- These macros define a simple converter that does bit-by-bit packing and unpacking in the order provided. Any customized conversions would require you write your own converter. See Converter Specialization for details.
- The macros support up to 20 member variables, i.e. up through UVM_UTILS_20 and UVM_UTILS_EXT_20.

23.1.2 UVMC_PRINT

Generate an *operator*<<*(ostream&)* implementation for use with *cout* and other output streams for the given transaction *TYPE*.

```
UVMC_PRINT_<N> (TYPE, <list of N variables> )
UVMC_PRINT_EXT_<N> (TYPE, BASE, <list of N variables> )
```

For the second form, the generated output stream operator will stream the contents of the provided *BASE* class before streaming those of *TYPE*.

Invoke the macro whose numeric suffix equals the number of field members you wish to include in the pack, unpack, and print operations. These must all appear in the list of macro arguments in the order you want them streamed.

23.1.2.1 Example

```
UVMC_PRINT_3( bus_trans, cmd, addr, data)
UVMC_PRINT_EXT_1 (bus_error, bus_trans, crc)
```

The first macro generates an *operator*<< for the *bus_trans* class. Three member variables of *bus_trans* are included in the output stream operation, in the order given: cmd, addr, and data. If there were other members in *bus_trans*, they will not be included in the *operator*<< implementations.

The second macro generates an *operator*<< for the *bus_error* class. Output streaming for the base type, *bus_trans* is performed first, followed by the *crc* field in *bus_error*.

The macros above generate the following code

```
template <>
```

```
class uvmc_print<bus_trans> {
  public:
  static void do_print(const bus_trans& t, ostream& os=cout) {
    os << hex << "cmd"  ":" << t.cmd << " "
                 "addr" ":" << t.addr << " "
                 "data" ":" << t.data << dec;
  }
  static void print(const bus_trans& t, ostream& os=cout) {
    os << "'{";
    do_print(t,os);
    os << " }";
  }
};
ostream& operator << (ostream& os, const bus_trans& v) {
  uvmc_print<bus_trans>::print(v,os);
}

template <>
class uvmc_print<bus_error> {
  public:
  static void do_print(const bus_error& t, ostream& os=cout) {
    uvmc_print<bus_trans>::do_print(t,os);
    os << " ";
    os << hex << "crc" ":" << t.cmd << " "
  }
  static void print(const bus_error& t, ostream& os=cout) {
    os << "'{";
    do_print(t,os);
    os << " }";
  }
};
ostream& operator << (ostream& os, const bus_error& v) {
  uvmc_print<bus_error>::print(v,os);
}
```

23.1.2.2 Usage notes

- All class members given in the list must be public members
- An *operator<<(ostream)* must be defined for each class member type listed. This is true for integral, string, and SystemC data types. UVMC also defines *operator<<(ostream&)* for STL *vector<T>*, *map<KEY,T>*, and *list<T>* types.
- The macros support up to 20 member variables, i.e. *UVM_UTILS_20* and *UVM_UTILS_EXT_20*.

Before using these macros, be sure the *#include* the appropriate headers that define *ostream* and other I/O utilities.

```
#include <iostream>
#include <iomanp>
```

23.1.3 UVMC_UTILS

Generate both a converter specialization and output stream *operator<<* for the given transaction *TYPE*.

```
UVMC_UTILS_<N> (TYPE, <list of N variables> )
UVMC_UTILS_EXT_<N> (TYPE, BASE, <list of N variables> )
```

For the second form, the generated converter and output stream operator will perform the operation in *BASE* before doing the same in *TYPE*.

Invoke the macro whose numeric suffix equals the number of field members you wish to include in the pack, unpack, and print operations.

The *UVMC_UTILS* macro simply calls the corresponding *UVMC_CONVERT* and *UVMC_PRINT* macros. See UVMC_CONVERT and UVMC_PRINT for detailed usage information.

23.1.3.1 Example

```
UVMC_UTILS_3( bus_trans, cmd, addr, data)
UVMC_UTILS_EXT_1 (bus_error, bus_trans, crc)
```

The first macro generates converter and output stream implementions for *bus_trans*. Three member variables are included in the pack, unpack, and output stream operations, in the order given: *cmd*, *addr*, and *data*. If there were other members in *bus_trans*, they will not be included in the converter or *operator<<* implementations.

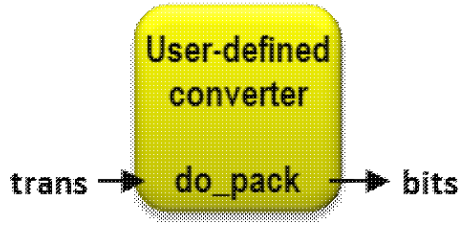
The second macro generates a converter and *operator<<* for the *bus_error* class. Packing, unpacking, and output streaming for the base type, *bus_trans* is performed first, followed by the *crc* field in *bus_error*.

The code that these macros expand into are provided in the descriptions of UVMC_CONVERT and UVMC_PRINT.

The members of your transaction definitions may be any collection of the following types, which have direct support in UVMC.
Define a separate class for converting your transaction type.
Generate a converter specialization of *uvmc_convert<T>* for the given transaction *TYPE*.
Generate an *operator<<(ostream&)* implementation for use with *cout* and other output streams for the given transaction *TYPE*.

24 UVMC Converter Example - SC Converter Class

This example demonstrates how to define an external converter class for a given transaction type. The user-defined converter class is a template specialization of the default converter, *uvmc_converter<T>*.



Because most SC transactions do not implement the pack and unpack member functions required by the default converter, a template specialization of *uvmc_converter<T>* is usually required.

You can define a template specialization for your transaction as in this example, or you could use one of the UVMC_UTILS macros to generate a definition for you. See UVMC Converter Example - SC Converter Class, Macro-Generated for details.

24.1 User Library

This section defines a "user library" consisting of a *packet* transaction class and a generic consumer model. This example will define a converter for this packet, then connect an instance of the consumer with an SV-side producer using a blocking transport interface conveying that transaction.

```
namespace user_lib {

  class packet_base
  {
    public:
    enum cmd_t { WRITE=0, READ, NOOP };

    cmd_t cmd;
    unsigned int addr;
    vector<unsigned char> data;
  };

  class packet : public packet_base
  {
    public:
    int extra_int;
  };

  // a generic target with a TLM2 b_transport export
  #include "consumer.cpp"

}
```

24.1.1 Conversion code

This section defines a converter specialization for our 'packet' transaction type.

We can not use the default converter because it delegates to *pack* and *unpack* methods of the transaction, which our packet class doesn't have. So, we define a converter template specialization for our packet type. You would implement a transaction converter for your specific transaction type in much the same manner.

The definition of a SC-side converter specialization is so regular that a set of convenient macros have been developed to produce a converter class definition for you. See UVMC Converter Example - SC Converter Class, Macro-Generated for for an example of using the UVMC_UTILS macros. See UVMC_PRINT for how to define *operator<<(ostream&)* so you can print your transaction contents to *cout* or any other output stream.

```
#include "uvmc.h"
using namespace uvmc;
using namespace user_lib;

template <>
struct uvmc_converter<packet_base> {
  static void do_pack(const packet_base &t, uvmc_packer &packer) {
    packer << t.cmd << t.addr << t.data;
  }
  static void do_unpack(packet_base &t, uvmc_packer &packer) {
    packer >> t.cmd >> t.addr >> t.data;
  }
};

template <>
struct uvmc_converter<packet> {
  static void do_pack(const packet &t, uvmc_packer &packer) {
    uvmc_converter<packet_base>::do_pack(t,packer);
    packer << t.extra_int;
  }
  static void do_unpack(packet &t, uvmc_packer &packer) {
    uvmc_converter<packet_base>::do_unpack(t,packer);
    packer >> t.extra_int;
  }
};

UVMC_PRINT_3(packet_base,cmd,addr,data)
UVMC_PRINT_EXT_1(packet,packet_base,extra_int)
```

24.1.2 Testbench code

This section defines our testbench environment. In the top-level module, we instantiate the generic consumer model. We also register the consumer's 'in' export to have a UVMC connection with a lookup string 'stimulus'. The SV-side will register its producer's 'out' port with the same 'stimulus' lookup string. UVMC will match these two strings to complete the cross- language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

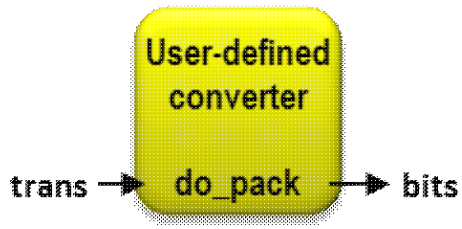
```
class sc_env : public sc_module
{
  public:
  consumer<packet> cons;

  sc_env(sc_module_name nm) : cons("cons") {
    uvmc_connect(cons.in,"stimulus");
  }
};
```

```
// Define sc_main, the vendor-independent means of starting a
// SystemC simulation.

int sc_main(int argc, char* argv[])
{
  sc_env env("env");
  sc_start();
  return 0;
}
```

Generate both a converter specialization and output stream *operator<<* for the given transaction *TYPE*.
This example demonstrates how to define an external converter for a transaction class using a UVMC_UTILS macro.
Generate an *operator<<(ostream&)* implementation for use with *cout* and other output streams for the given transaction *TYPE*.

25 UVMC Converter Example - SC Converter Class, Macro-Generated

This example demonstrates how to define an external converter for a transaction class using a UVMC_UTILS macro. The macro expands into a template specialization of *uvmc_converter*<*T*> for your type *T*, overriding the default converter's implementation. An example showing how to write such a template specialization without the convenience macros can be found in UVMC Converter Example - SC Converter Class.



In SC, a separate converter class definition is usually required because most SC transactions do not implement the pack and unpack member functions required by the default converter. To UVMC_UTILS macros can be used to quickly define a converter for your transaction type. The macro would expand into the same code you would write directly, so these macros do not suffer from the performance and debug costs associated with other macros such as the *`uvm_field* macros.

The UTILS macros also defined *operator*<< for the output stream (e.g. cout). With this, we can print the transaction contents to any output stream. For example:

```
packet p;
// initialize p...
cout << p;
```

This produces output similar to

```
'{cmd:2 addr:1fa34f22 data:'{4a, 27, de, a2, 6b, 62, 8d, 1d, 6} }
```

Template specializations are chosen automatically by the C+ compiler, so you will not need to explicitly specify your converter type when connecting via *uvmc_connect*.

25.1 User Library

This section defines a transaction class and generic consumer model. We will define a converter for this packet, then connect an instance of the consumer with an SV-side producer using a blocking transport interface conveying that transaction.

```
namespace user_lib {

  class packet_base
  {
    public:
    enum cmd_t { WRITE=0, READ, NOOP };

    cmd_t cmd;
    unsigned int addr;
```

```
    vector<unsigned char> data;
  };

  class packet : public packet_base
  {
    public:
    int extra_int;
  };

}
```

25.1.1 Conversion code

This section defines a converter for our *packet* transaction type using a UVMC_UTILS macro for each class in the *packet* inheritance hierarchy..

The definition of the SC-side template specialization is so regular that a set of convenient macros have been developed to produce a converter class definition for you. You need to invoke one of these macros, depending on the number of fields in your transaction class and whether it inherits from a base class.

```
#include "uvmc.h"
using namespace uvmc;
using namespace user_lib;

UVMC_UTILS_3(packet_base,cmd,addr,data)
UVMC_UTILS_EXT_1(packet,packet_base,extra_int)
```

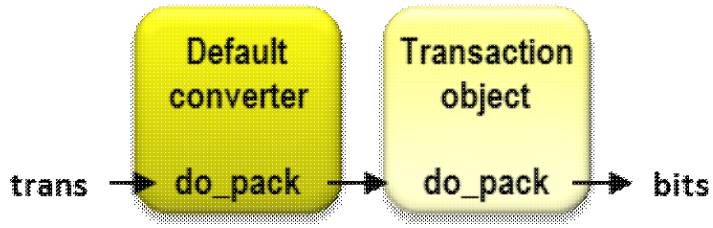
25.1.2 Testbench code

This section defines our testbench environment. In the top-level module, we instantiate the generic consumer model. We also register the consumer's 'in' export to have a UVMC connection with a lookup string 'stimulus'. The SV-side will register its producer's 'out' port with the same 'stimulus' lookup string. UVMC will match these two strings to complete the cross- language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

Generate both a converter specialization and output stream *operator*<< for the given transaction *TYPE*.
This example demonstrates how to define an external converter class for a given transaction type.

26 UVMC Converter Example - SC In-Transaction

This example's packet class defines *do_pack* and *do_unpack* methods that are compatible with the default converter in SC. The default converter merely delegates conversion to these two methods in the transaction class.



This approach is not very common in practice, as it couples your transaction types to the UVMC library.

Instead of defining member functions of the transaction type to do conversion, you should instead implement a template specialization of uvmc_convert<T>. This leaves conversion knowledge outside your transaction proper, and allows you to define different conversion algorithms without requiring inheritance.

26.1 User Library

This section defines a transaction class and generic consumer model. The transaction implements the *do_pack* and *do_unpack* methods required by the default converter.

Packing and unpacking involves streaming the contents of your transaction fields into and out of the *packer* object provided as an argument to *do_pack* and *do_unpack*. The packer needs to have defined *operator*<< for each type of field you stream. See UVMC Type Support for a list of supported transaction field types.

```
namespace user_lib {

  using namespace uvmc;

  class packet_base
  {
    public:
    enum cmd_t { WRITE=0, READ, NOOP };

    cmd_t cmd;
    unsigned int addr;
    vector<unsigned char> data;

    virtual void do_pack(uvmc_packer &packer) const {
      packer << cmd << addr << data;
    }

    virtual void do_unpack(uvmc_packer &packer) {
      packer >> cmd >> addr >> data;
    }
  };


  class packet : public packet_base
  {
```

```
    public:
    unsigned int extra_int;

    virtual void do_pack(uvmc_packer &packer) const {
      packet_base::do_pack(packer);
      packer << extra_int;
    }

    virtual void do_unpack(uvmc_packer &packer) {
      packet_base::do_unpack(packer);
      packer >> extra_int;
    }
  };

  // a generic target with a TLM2 b_transport export
  #include "consumer.cpp"

}
```

26.1.1 Conversion code

We do not need to define an external conversion class because it conversion is built into the transaction proper. The default converter will delegate to our transaction's *do_pack* and *do_unpack* methods.

We do, however, define *operator<< (ostream&)* for our transaction type using UVMC_PRINT macros. With this, we can print the transaction contents to any output stream.

For example

```
packet p;
...initialize p...
cout << p;
```

This produces output similar to

```
'{cmd:2 addr:1fa34f22 data:'{4a, 27, de, a2, 6b, 62, 8d, 1d, 6} }
```

You can invoke the macros in any namepace in which the uvmc namespace was imported and the macros #included.

```
using namespace user_lib;

UVMC_PRINT_3(packet_base,cmd,addr,data)
UVMC_PRINT_EXT_1(packet,packet_base,extra_int)
```

26.1.2 Testbench code

This section defines our testbench environment. In the top-level module, we instantiate the generic consumer model. We also register the consumer's *in* export to have a UVMC connection with a lookup string, *stimulus*. The SV-side will register its producer's *out* port with the same lookup string. UVMC will match these two strings to complete the cross-language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

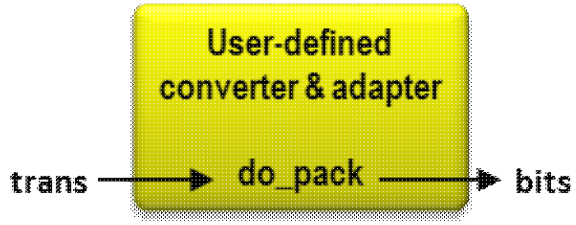
```
class sc_env : public sc_module
{
```

```
  public:
  consumer<packet> cons;

  sc_env(sc_module_name nm) : cons("cons") {
    uvmc_connect(cons.in,"stimulus");
  }
};

// Define sc_main, the vendor-independent means of starting a
// SystemC simulation.

int sc_main(int argc, char* argv[])
{
  sc_env env("env");
  sc_start();
  return 0;
}
```

The members of your transaction definitions may be any collection of the following types, which have direct support in UVMC.
Generate an *operator<<(ostream&)* implementation for use with *cout* and other output streams for the given transaction *TYPE*.

27 UVMC Converter Example - SC Adapter Class

This example demonstrates how to define a custom converter for a transaction class whose members differ in number, type, and size from the corresponding transaction definition in SV. This situation can arise in cases where the transaction types are pre-existing in both SC and SV yet have compatible content.



Because most SC transactions do not implement the pack and unpack member functions required by the default converter, a template specialization definition is required. A template specialization can be defined by hand or via a UVMC_UTILS macro, which defines the same converter specialization plus the operator<< for the default output stream (cout). This allows you to print your packet contents using cout << my_packet;

Template specializations are chosen automatically by the C+ compiler, so you will not need to specify the converter type explicitly when connecting via The Connect Function.

27.1 User Library

This section defines a transaction class and generic consumer model. We will define a converter for this packet, then connect an instance of the consumer with an SV-side producer using a blocking transport interface conveying that transaction.

```
namespace user_lib {

  class packet
  {
    public:
    short addr_hi;
    short addr_lo;
    unsigned int payload[4];
    char  len;
    bool  write; // 1=write, 0=read
  };
}
```

27.1.1 Conversion code

This section defines a converter specialization for our 'packet' transaction type. We can not use the default converter because it delegates to *pack* and *unpack* methods of the transaction, which our packet class doesn't have.

So, we define a converter template specialization for our packet type. Your transaction converters would implement the same template.

The definition of a SC-side converter specialization is so regular that a set of convenient macros have been developed to produce a converter class definition for you. You would invoke one of the macros from the set,

depending on the number of fields in your transaction class and whether it inherits from a base class. See UVMC Converter Example - SC Converter Class, Macro-Generated for for an example of using the UVMC_UTILS macros.

The corresponding transaction in SV declares the following fields, split across two classes (one inheriting from the other), in the given order.

```
class packet_base extends uvm_sequence_item:
  typedef enum {WRITE, READ, NOOP} cmd_t;
  cmd_t cmd;
  int   addr;
  byte  data[$];
endclass

class packet:
  int extra_int;
endclass
```

If we could define our SC-side transaction to suit this definition, we'd mirror the types, declaration order, and even the inheritance hierarchy. In this example, however, we are faced with having to adapt to a pre-existing transaction type.

When writing the converters on the SV and SC side, we can choose three different ways:

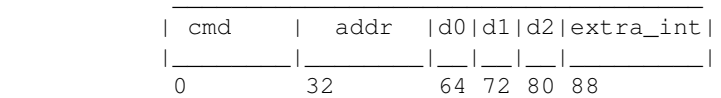
1: Let the SC converter pack/unpack normally; implement a custom SV converter to convert according to how the SC side expects to receive the bits.

2: Let the SV converter pack/unpack normally; implement a SC converter specialization of the default SC converter to convert according to how the SV side expects to receive the bits.

3: Let the SV converter pack/unpack normally; implement a subtype to the SC converter specialization to convert according to how the SV side expects to receive the bits. Specify the custom converter type when calling uvmc_connect.

A converter specialization, e.g. *template <> class uvmc_converter<packet>*, should be reserved for converting the SC transaction as it is defined, streaming each field in order and without adaptation. It should not be used to adapt to a custom mapping on the SV side, as in this example. For this reason, option 3 is the best.

The *packet* transaction in SV will be packed normally: *cmd*, *addr*, *data*, and *extra_int*. The SV packetized bits, assuming 3 bytes in the data array, looks like this:



In SC, we shall adapt as follows

- map the 32-bit *cmd* from SV to a single *bool* in SV
- map the 32-bit *addr* from SV to two *addr_lo* and *addr_hi* 16-bit values in SC
- map the *data* byte array data from SV to an integer array in SV

When dealing with built-in types, you should account for the endianess of your machine's architecture. This example assumes a little-endian architecture.

```
#include <vector>
#include <iomanip>
using std::vector;

#include "uvmc.h"
using namespace uvmc;
using namespace user_lib;

struct packet_converter : public uvmc_converter<packet>
{
  static void do_pack(const packet &t, uvmc_packer &packer) {
    int cmd_tmp;
    if (t.write)
      cmd_tmp = 0;
    else
      cmd_tmp = 1;
    packer << cmd_tmp
           << t.addr_lo << t.addr_hi
           << (int)(t.len) << t.payload;
  }

  static void do_unpack(packet &t, uvmc_packer &packer) {
    int cmd_tmp;
    vector<unsigned char> data_tmp;
    packer >> cmd_tmp >> t.addr_lo >> t.addr_hi >> data_tmp;
    t.len = data_tmp.size();
    if (cmd_tmp == 0)
      t.write = 1;
    else if (cmd_tmp == 1)
      t.write = 0;
    else
      cout << "packet cmd from SV side has unsupported value "
           << cmd_tmp << endl;
    for (int i=0;i<4;i++)
      t.payload[i]=0;
    for (int i=0;i<4;i++) {
      for (int j=0;j<4;j++) {
        if ((i*4+j)<t.len) {
          int b;
          b = data_tmp[i*4+j] << (8*j);
          t.payload[i] = t.payload[i] | b;
        }
        else {
          break;
        }
      }
    }
  }

};

UVMC_PRINT_4(packet,addr_hi,addr_lo,len,write)
```

27.1.2 Testbench code

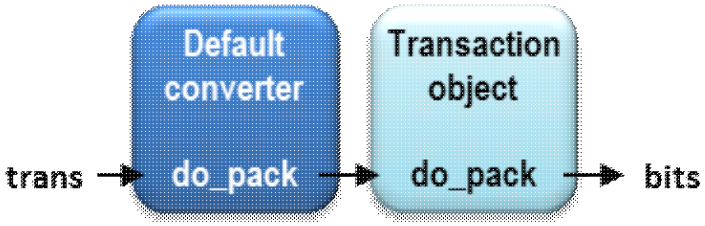
This section defines our testbench environment. In the top-level module, we instantiate the generic consumer model. We also register the consumer's 'in' export to have a UVMC connection with a lookup string 'stimulus'. The SV-side will register its producer's 'out' port with the same 'stimulus' lookup string. UVMC will match these two strings to complete the cross- language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

```
#include "systemc.h"
#include "tlm.h"
using namespace sc_core;
using namespace tlm;

// a generic target with a TLM2 b_transport export
#include "consumer2.cpp"

class sc_env : public sc_module
{
  public:
  consumer<packet> cons;

  sc_env(sc_module_name nm) : cons("cons") {
    uvmc_connect<packet_converter>(cons.in,"stimulus");
  }
};

// Define sc_main, the vendor-independent means of starting a
// SystemC simulation.

int sc_main(int argc, char* argv[])
{
  sc_env env("env");
  sc_start();
  return 0;
}
```

Generate both a converter specialization and output stream *operator*<< for the given transaction *TYPE*.
The *connect* and *connect_hier* functions are used to register any type of TLM port, export, interface, imp, or socket for connection across the language boundary.
This example demonstrates how to define an external converter for a transaction class using a UVMC_UTILS macro.

28 UVMC Converter Example - SV In-Transaction

This example shows how to implement the conversion routines in UVM-style transaction in the virtual *do_pack* and *do_unpack* functions inherited from the *uvm_object* base class.



Most SV transactions extend *uvm_sequence_item*, which extends *uvm_object*, which defines virtual *do_pack* and *do_unpack* methods for override in user-defined transaction types. The UVMC's default converter for SV works for these types of transactions. Defining SV-side transasctions in this way minimizes the extra code needed to make a cross-language connection.

28.1 User Library

This section defines a transaction class, *packet*, that indirectly extends *uvm_object*. It also defines a generic producer model via an `include. All transactions and components in the user library should be written to be independent of context, i.e. not assume a UVMC or any other outside connetion.

The *`uvm_pack_** and *`uvm_unpack_** macros expand into two or so lines of code that are more efficient than using the packer's API directly. These macros are part of the UVM standard and are documented under the *Macros* heading in the UVM Reference Manual.

```
package user_pkg;

  `include "uvm_macros.svh"
  import uvm_pkg::*;

  class packet_base extends uvm_sequence_item;

    `uvm_object_utils(packet_base)

    typedef enum { WRITE, READ, NOOP } cmd_t;

    rand cmd_t cmd;
    rand int   addr;
    rand byte  data[$];

    function new(string name="");
      super.new(name);
    endfunction

    constraint c_data_size { data.size() inside { [1:10] }; }

    virtual function void do_pack(uvm_packer packer);
      `uvm_pack_enum(cmd)
      `uvm_pack_int(addr)
      `uvm_pack_queue(data)
    endfunction
```

```
    virtual function void do_unpack(uvm_packer packer);
      `uvm_unpack_enum(cmd,cmd_t)
      `uvm_unpack_int(addr)
      `uvm_unpack_queue(data)
    endfunction

    virtual function string convert2string();
      return $sformatf("cmd:%s addr:%h data:%p",cmd,addr,data);
    endfunction

  endclass


  class packet extends packet_base;

    `uvm_object_utils(packet)

    rand int extra_int;

    function new(string name="");
      super.new(name);
    endfunction

    virtual function void do_pack(uvm_packer packer);
      super.do_pack(packer);
      `uvm_pack_int(extra_int)
    endfunction

    virtual function void do_unpack(uvm_packer packer);
      super.do_unpack(packer);
      `uvm_unpack_int(extra_int)
    endfunction

    virtual function string convert2string();
      return $sformatf("%s extra_int:%h",super.convert2string(),extra_int);
    endfunction

  endclass

  `include "producer.sv"

endpackage : user_pkg
```

28.1.1 Conversion code

This section is empty because our conversion functionality is built into the transaction proper.

```
/***  No external conversion code needed  ***/
```

28.1.2 Testbench code

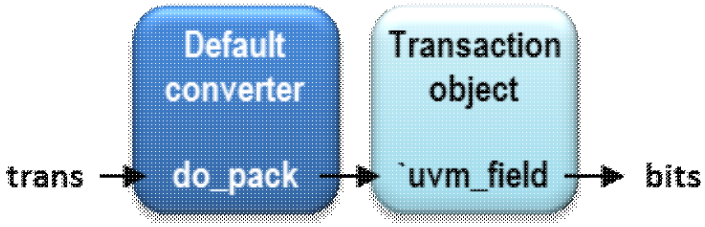
This section defines our testbench environment. In the env's *build* function, we instantiate the generic producer model. In the *connect* method, we register the producer's *out* port for UVMC connection using the lookup string 'stimulus'. The SC-side will register its consumer's *in* port with the same lookup string. UVMC will match these two strings and complete the cross- language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

Because our *packet* class implements the requisite *do_pack* and *do_unpack* methods, we can leverage UVMC's default converter, which delegates to these methods. When making the uvmc_tlm::connect call, we do not need to specify a custom converter type--only the transaction type.

```
module sv_main;

  `include "uvm_macros.svh"
  import uvm_pkg::*;
  import uvmc_pkg::*;
  import user_pkg::*;

  // Define env with connection specifying custom converter

  class sv_env extends uvm_env;

    producer #(packet) prod;

    `uvm_component_utils(sv_env)

    function new(string name, uvm_component parent=null);
       super.new(name,parent);
    endfunction

    function void build_phase(uvm_phase phase);
       prod = new("prod", this);
    endfunction

    function void connect_phase(uvm_phase phase);
      uvmc_tlm #(packet)::connect(prod.out, "stimulus");
    endfunction

  endclass


  sv_env env;

  initial begin
    env = new("env");
    run_test();
  end

endmodule
```

29 UVMC Converter Example - SV In-Transaction via Field Macros

This example shows a UVM-style transaction that uses the `uvm_field macros to implement the required conversion functionality.



UVMC's default converter for SV works for these types of transactions. Direct implementation of the *do_pack* and *do_unpack* methods are preferred over using the *`uvm_field* macros because of the performance impact and low debug support associated with these macros. See "Are UVM Macros Evil? A Cost Beneifit Analysis" white paper for detailed analysis of the *`uvm_field* and other UVM macros.

29.1 User Library

This section defines a transaction class, *packet*, that indirectly extends *uvm_object*. It also defines a generic producer model via an `include. All transactions and components in the user library should be written to be independent of context, i.e. not assume a UVMC or any other outside connetion.

The *`uvm_field* macros expand into hundreds of lines of code, perhaps thousands depending on the number and type of fields in your transaction. See the example showing direct implementation of *do_pack* and *do_unpack* for a better solution.

```
package user_pkg;

  `include "uvm_macros.svh"
  import uvm_pkg::*;

  class packet_base extends uvm_sequence_item;

    typedef enum { WRITE, READ, NOOP } cmd_t;

    rand cmd_t cmd;
    rand int   addr;
    rand byte  data[$];

    function new(string name="");
      super.new(name);
    endfunction

    constraint c_data_size { data.size() inside { [1:10] }; }

    `uvm_object_utils_begin(packet_base)
      `uvm_field_enum(cmd_t,cmd,UVM_ALL_ON)
      `uvm_field_int(addr,UVM_ALL_ON)
      `uvm_field_queue_int(data,UVM_ALL_ON)
    `uvm_object_utils_end
```

```
    virtual function string convert2string();
      return $sformatf("cmd:%s addr:%h data:%p",cmd,addr,data);
    endfunction

  endclass


  class packet extends packet_base;

    rand int extra_int;

    function new(string name="");
      super.new(name);
    endfunction

    `uvm_object_utils_begin(packet)
      `uvm_field_int(extra_int,UVM_ALL_ON)
    `uvm_object_utils_end

    virtual function string convert2string();
      return $sformatf("%s extra_int:%h",super.convert2string(),extra_int);
    endfunction

  endclass

  `include "producer.sv"

endpackage : user_pkg
```

29.1.1 Conversion code

This section is empty because our conversion functionality is built into the transaction type proper.

```
/***  No external conversion code needed  ***/
```

29.1.2 Testbench code

This section defines our testbench environment. In the env's *build* function, we instantiate the generic producer model. In the *connect* method, we register the producer's *out* port for UVMC connection using the lookup string 'stimulus'. The SC-side will register its consumer's *in* port with the same lookup string. UVMC will match these two strings and complete the cross- language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

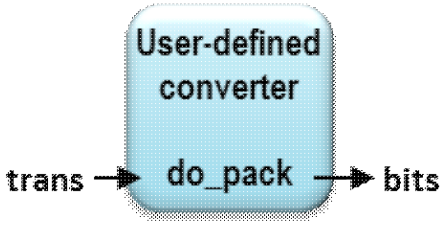
Because our *packet* class implements the requisite *do_pack* and *do_unpack* methods, we can leverage UVMC's default converter, which delegates to these methods. When making the uvmc_tlm::connect call, we do not need to specify a custom converter type--only the transaction type.

```
module sv_main;

  `include "uvm_macros.svh"
  import uvm_pkg::*;
  import uvmc_pkg::*;
  import user_pkg::*;

  // Define env with connection specifying custom converter
```

```
  class sv_env extends uvm_env;

    producer #(packet) prod;

    `uvm_component_utils(sv_env)

    function new(string name, uvm_component parent=null);
       super.new(name,parent);
    endfunction

    function void build_phase(uvm_phase phase);
       prod = new("prod", this);
    endfunction

    function void connect_phase(uvm_phase phase);
      uvmc_tlm #(packet)::connect(prod.out, "stimulus");
    endfunction

  endclass


  sv_env env;

  initial begin
    env = new("env");
    run_test();
  end

endmodule
```

30 UVMC Converter Example - SV Converter Class

This example demonstrates how to define a custom converter for a transaction class that does not extend from *uvm_object*.



Most SV transactions extend *uvm_object* and implement the *do_pack* and *do_unpack* methods. The default converter for SV works for these types of transactions, so in most cases you will not need to define an external converter class.

To apply the external converter to a particular cross-language connection, specify it as a type parameter when registering a UVMC connection

```
uvmc_tlm #(packet, my_converter)::connect( some_port, "some_lookup");
```

30.1 User Library

This section defines a transaction class, *packet*, that does not extend from any base class. It also defines a generic producer model via `include. All transactions and components in the user library should be written to be independent of context, i.e. not assume a UVMC or any other outside connetion.

30.1.1 Conversion code

This section defines a converter for our 'packet' transaction type. We will later use this converter when registering cross-language connections to SC.

The *`uvm_pack_** and *`uvm_unpack_** macros expand into two or so lines of code that are more efficient than using the packer's API directly. These macros are part of the UVM standard and are documented under the *Macros* heading in the UVM Reference Manual.

```
package convert_pkg;

  `include "uvm_macros.svh"
  import uvm_pkg::*;
  import uvmc_pkg::*;
  import user_pkg::*;

  class convert_packet_base extends uvmc_converter #(packet_base);

    static function void do_pack(packet_base t, uvm_packer packer);
      `uvm_pack_enum(t.cmd)
      `uvm_pack_int(t.addr)
      `uvm_pack_queue(t.data)
    endfunction

    static function void do_unpack(packet_base t, uvm_packer packer);
```

```
      `uvm_unpack_enum(t.cmd,packet_base::cmd_t)
      `uvm_unpack_int(t.addr)
      `uvm_unpack_queue(t.data)
    endfunction

  endclass


  class convert_packet extends uvmc_converter #(packet);

    static function void do_pack(packet t, uvm_packer packer);
      convert_packet_base::do_pack(t,packer);
      `uvm_pack_int(t.extra_int)
    endfunction

    static function void do_unpack(packet t, uvm_packer packer);
      convert_packet_base::do_unpack(t,packer);
      `uvm_unpack_int(t.extra_int)
    endfunction

  endclass

endpackage
```

30.1.2 Testbench code

This section defines our testbench environment. In the top-level module, we instantiate the generic producer model. We also register the producer's 'out' port to have a UVMC connection with a lookup string 'stimulus'. The SC-side will register its consumer's 'in' port with the same 'stimulus' lookup string. UVMC will match these two strings and complete the cross- language connection, i.e. the SV producer's *out* port will be bound to the SC consumer's *in* export.

```
module sv_main;

  `include "uvm_macros.svh"
  import uvm_pkg::*;
  import uvmc_pkg::*;
  import user_pkg::*;
  import convert_pkg::*;

  // Define env with connection specifying custom converter

  class sv_env extends uvm_env;

    producer #(packet) prod;

    `uvm_component_utils(sv_env)

    function new(string name, uvm_component parent=null);
       super.new(name,parent);
    endfunction

    function void build_phase(uvm_phase phase);
       prod = new("prod", this);
    endfunction

    function void connect_phase(uvm_phase phase);
      uvmc_tlm #(packet,uvm_tlm_phase_e,convert_packet)::
                              connect(prod.out, "stimulus");
```

```
    endfunction

  endclass


  sv_env env;

  initial begin
    env = new("env");
    run_test();
  end

endmodule
```

31 UVMC Converter Common Code - consumer

31.1 Description

A generic consumer parameterized on the transaction type. Used to illustrate different converter options using the same consumer class. Functionally, this consumer merely prints the transaction and inverts its address and data before returning. The producer will verify that the address and data have been inverted, which proves reasonably that the transaction successfully made the round trip to SV and back.

../../../uvmc/examples/converters/consumer.cpp

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

template
```

32 UVMC Converter Common Code - consumer2

32.1 Description

A generic consumer parameterized on the transaction type. Used to illustrate different converter options using the same consumer class. Functionally, this consumer merely prints the transaction and inverts its address and data before returning. The producer will verify that the address and data have been inverted, which proves reasonably that the transaction successfully made the round trip to SV and back.

../../../uvmc/examples/converters/consumer2.cpp

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

template
```

33 UVMC Converter Common Code - SV Producer

33.1 Description

A generic producer parameterized on transaction type. Used to illustrate different converter options using the same producer class.

../../../uvmc/examples/converters/producer.sv

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

class producer #(type T=int) extends uvm_component;

   uvm_tlm_b_transport_port #(T) out;

   int num_pkts;

   `uvm_component_param_utils(producer #(T))

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      out = new("out", this);
      num_pkts = 10;
   endfunction : new

   task run_phase (uvm_phase phase);
     uvm_tlm_time delay = new("delay",1.0e-12);

     phase.raise_objection(this);

     for (int i = 1; i <= num_pkts; i++) begin

       int unsigned exp_addr;
       byte exp_data[$];

       T pkt = new(); //$sformatf("packet%0d",i));
       assert(pkt.randomize());
       delay.set_abstime(1,1e-9);

       exp_addr = ~pkt.addr;
```

```
      foreach (pkt.data[i])
        exp_data[i] = ~pkt.data[i];

      `uvm_info("PRODUCER/PKT/SEND_REQ",
         $sformatf("SV producer request:\n   %s", pkt.convert2string()), UVM_MEDIUM)

      out.b_transport(pkt,delay);

      `uvm_info("PRODUCER/PKT/RECV_RSP",
         $sformatf("SV producer response:\n   %s\n", pkt.convert2string()), UVM_MEDIUM)

      if (exp_addr != pkt.addr)
        `uvm_error("PRODUCER/PKT/RSP_MISCOMPARE",
          $sformatf("SV producer expected returned address to be %h, got back %h",
                  exp_addr,pkt.addr))

      if (exp_data != pkt.data)
        `uvm_error("PRODUCER/PKT/RSP_MISCOMPARE",
          $sformatf("SV producer expected returned data to be %p, got back %p",
                    exp_data,pkt.data))
    end

    `uvm_info("PRODUCER/END_TEST","Dropping objection to ending the test",UVM_LOW)
    phase.drop_objection(this);

  endtask

endclass
```

34 Fast packer converters

34.1 Introduction

Now that you've read the chapter on *Converters* and understand how to create custom variations of converters and packers, this section shows 2 special types of custom converters called *fast packers*.

Fast packers are designed to operate specifically on TLM generic payload (TLM GP) transaction types (i.e. SV **class uvm_tlm_generic_payload**, SC **class tlm_generic_payload**) when the application passes them across UVM-Connect'ed sockets.

For the fast packer converter classes, the same technique described in the previous section for creating custom converter classes was used to create these particular converters for use on both sides of the UVM-Connect'ion.

34.1.1 Fast packer features

These "fast packers" add two features in contrast to their *default converter* counterparts,

- Improved performance
- Support for TLM generic payloads with no fixed limitations on data payload sizes (i.e. unlimited data payloads)

There are two flavors of fast packers,

```
1. class uvmc_xl_converter
2. class uvmc_tlm_gp_converter
```

The same class names were used for both the SV and SC versions of these classes.

The fast packer converter classes have both been enhanced for better performance than the default packers, and both support unlimited payloads. But there are slightly differing semantics for each of the two.

34.1.2 The class uvmc_xl_converter packer class

The **class uvmc_xl_converter** conforms in the strictest sense to the required semantics of the TLM-2.0 base protocol specifically with respect to modifiability of attributes (see IEEE 1666-2011 section on TLM-2.0 base protocol), and thus does not indiscriminately transfer all fields of the generic payload in both directions across the language boundary.

Rather it decides, depending on the mode of the transaction (READ or WRITE), and whether it is being transferred along the forward, backward, or return paths, which fields to transfer and which to leave alone.

For example WRITE transactions do not need to have the address, data, length, enables and control fields transferred long the return path - only the status.

READ transactions also do not need address, length, enables, transferred along the return path, only data and status. And furthermore for READs data does **not** need to be transferred along the forward/backward path (only return path).

These fine-tuned optimizations, along with some use of *pass-by-reference* semantics collectively have maximized the performance purely at the communication layers of the UVM-Connect'ion.

Additionally the **class uvmc_xl_converter** supports the passing of TLM GP *configuration extensions* that derive from **class uvmc_xl_config**. See the section on Configuration extensions for more details about how they can be used.

34.1.3 The class uvmc_tlm_gp_converter packer class

The **class uvmc_tlm_gp_converter** has the same features of unlimited payload size and efficient data payload passing techniques that use "C assist" and "pass by reference" that **class uvmc_xl_converter** above does, but it unconditionally transfers all fields of the generic payload along all paths without regard to modifiability of attributes which is more semantically compatible with the slower, size limited default packer, but which is less efficient that the **class uvmc_xl_converter** described above.

Both types of fast packers have been shown to be useful and are considered essential for different usage contexts.

34.1.4 How to use the fast packers

34.1.5 Specifying fast packers when uvmc_connect() is called

To use the fast packers for a specific TLM connection that uses TLM GPs as the transaction type, specify the desired converter type when calling the **uvmc_connect()** call.

Here is how it is done for the **class uvmc_xl_converter** variant,

- For SC side (see *sc2sv2sc_xl_gp_converter_loopback.cpp* for example):

```
int sc_main( int argc, char* argv[] ) {
  producer_uvm prod( "producer" );
  uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >( prod.out, "42" );
  uvmc_connect<uvmc_xl_converter<tlm_generic_payload> >( prod.in, "43" );
  sc_start(-1);
  return 0;
}
```

- For SV side (see *sc2sv2sc_xl_gp_converter_loopback.sv* for an example):

```
module sv_main;
  loopback loop = new( "loop" );

  initial begin
    uvmc_tlm #( uvm_tlm_generic_payload,
        uvm_tlm_phase_e, uvmc_xl_tlm_gp_converter)::connect( loop.in, "42" );
    uvmc_tlm #( uvm_tlm_generic_payload,
        uvm_tlm_phase_e, uvmc_xl_tlm_gp_converter)::connect( loop.out, "43");
    run_test();
  end
endmodule
```

And here is how it is done for the **class uvmc_tlm_gp_converter** variant,

- For SC side (see *sc2sv2sc_gp_converter_loopback.cpp* for example):

```
int sc_main( int argc, char* argv[] ) {
  producer_uvm prod( "producer" );
  uvmc_connect<uvmc_tlm_gp_converter>( prod.out,"42" );
  uvmc_connect<uvmc_tlm_gp_converter>( prod.in,"43" );
  sc_start(-1);
  return 0;
}
```

- For SV side (see *sc2sv2sc_gp_converter_loopback.sv* for an example):

```
module sv_main;
  loopback loop = new( "loop" );
  initial begin
    uvmc_tlm #( uvm_tlm_generic_payload,
        uvm_tlm_phase_e, uvmc_tlm_gp_converter) ::connect( loop.in, "42" );
    uvmc_tlm #( uvm_tlm_generic_payload,
        uvm_tlm_phase_e, uvmc_tlm_gp_converter) ::connect( loop.out, "43");
    run_test();
  end
endmodule
```

34.1.6 Fast packer source code

- For the fast packer source code files themselves see,

```
src/connect/
  sc/uvmc_tlm_gp_converter.*
  sc/uvmc_xl_converter.*
  sv/uvmc_converter.svh
  sv/uvmc_xl_converter.svh
```

34.1.7 Fast-packer converter examples

34.1.8 Running the examples

For examples that use the fast packer converters see,

```
examples/xlerate.connections/Makefile
```

This directory contains several examples of TLM-2 UVM-Connect'ions that pass TLM-2 generic payloads (TLM GPs) between SystemC (SC) and SystemVerilog (SV).

Use *make help* to view the menu of available tests,

```
make help
```

To run just one test such as *sc2sv2sc_loopback*,

```
make sc2sv2sc_loopback
```

This compiles then runs the *sc2sv2sc_loopback* test. See listing below for all possible tests in the *xlerate.connections/* suite.

To run all tests, check them, and clean up afterwards, i.e. *sim:*, *check:*, and *clean:* targets, use the *all:* target as follows,

```
make all
```

Or you can run individual sub-targets.

The *sim* target compiles and runs all the tests.

```
make sim
```

The *check* target checks results of all the tests.

```
make check
```

The *clean* target deletes all the simulation files produced from previous runs.

```
make clean
```

You can combine targets in one command line

```
make sim check
```

The following runs the 'sim' target, providing the path to the UVM source and compiled library on the *make* command line.

```
make UVM_HOME=<path> UVM_LIB=<path> sim
```

34.1.9 List of tests and what they do.

In all the tests below we want to benchmark the transfer of 80 2MB "HD-image" payloads across a UVM-Connect'ion.

With truly unlimited generic payloads (TLM GPs), we can represent each image as a single GP transaction.

However, because the default UVM packer based implementation limits the entire payload to 4KBytes (see default value of **`define UVM_PACKER_MAX_BYTES 4096** for UVM packers) we must break the image down into a series of small payload fragments. Each fragment must fit in a maximally sized generic payload as supported by UVM-Connect. This means, address, command, status, byte enables, lengths, and the payload data itself must all fit in the 4KB payload.

So, the tests below that use default UVM packers (see tests *sc2sv2sc_loopback* and *sv2sc2sv_loopback*) need the actual data payloads themselves to fit in 2KByte fragments which would leave ample room for the rest of the fixed sized data fields of the TLM GP (address, command, byte enables, etc) to fit in the remaining 2KB of the payload.

So with this in mind we can fragment our 80 2MB HD-images into 2KB chunks as,

```
  80 x 2 x 1024   x 1024 bytes
= 80 x 2 x 1024/2 x 2048 bytes
= 80 x 2 x 512            2048 byte payloads
= 81920                   2048 byte payloads
```

```
i.e. NUM_TRANSACTIONS = 81920, PAYLOAD_NUM_BYTES = 2048
```

Now with the 2 flavors of improved packers, in addition to getting better trans-language communication performance, there is no limit on payload size. In fact, there is no need for a globally defined static maximum byte stream size at all.

Having a statically specified global maximum of any kind always begs the question, “how big is big enough ?”. And so in many accelerated applications we try to avoid statically specified maximum payload sizes entirely on the HVL side of the link.

So without this limitation, we can restructure the test as,

```
80 x 2MB payloads

i.e. NUM_TRANSACTIONS = 80, PAYLOAD_NUM_BYTES = 2 * 1024 * 1024 = 2 MB
```

In the tests below this is referred to as a “whole image payload”. So for each of the tests listed in the Makefile’s you’ll see that each test specifies both of these parameters, **NUM_TRANSACTIONS** and **PAYLOAD_NUM_BYTES**, depending on whether the test is dividing each image into multiple 2KB chunks or sending the *whole image payload* 80 times.

Here is a listing of the tests in this example suite the indicates what packers are used and what payload kinds are used for each test (listing is generated with “make help” command).

```
 ----------------------------------------------------------------
|                  UVMC EXAMPLES - FAST PACKERS                  |
 ----------------------------------------------------------------
|                                                                |
| Usage:                                                         |
|                                                                |
|   make [UVM_HOME=path] [UVMC_HOME=path] <example>              |
|                                                                |
| where <example> is one or more of:                             |
|                                                                |
|   sc2sv2sc_loopback:                                           |
|                  SC producer <-> SV loopback                   |
|                  Connection is made via UVMC                   |
|                  with native default uvmc_converter's          |
|                                                                |
|   sv2sc2sv_loopback:                                           |
|                  SV producer <-> SC loopback                   |
|                  Connection is made via UVMC                   |
|                  with native default uvmc_converter's          |
|   ------------------------------------------------------------ |
|   sc2sv2sc_gp_converter_loopback:                              |
|                  SC producer <-> SV loopback                   |
|                  Connection is made via UVMC                   |
|                  with uvmc_tlm_gp_converter's                  |
|                                                                |
|   sv2sc2sv_gp_converter_loopback:                              |
|                  SV producer <-> SC loopback                   |
|                  Connection is made via UVMC                   |
|                  with uvmc_tlm_gp_converter's                  |
|                                                                |
|   sc2sv2sc_gp_converter_loopback_whole_image_payloads:         |
|                  SC producer <-> SV loopback                   |
```

```
|                 Connection is made via UVMC                      |
|                 with uvmc_tlm_gp_converter's                     |
|                 and big, nasty packets                           |
|                                                                  |
|   sv2sc2sv_gp_converter_loopback_whole_image_payloads:           |
|                 SV producer <-> SC loopback                      |
|                 Connection is made via UVMC                      |
|                 with uvmc_tlm_gp_converter's                     |
|                 and big, nasty packets                           |
|   ---------------------------------------------------------      |
|   sc2sv2sc_xl_gp_converter_loopback:                             |
|                 SC producer <-> SV loopback                      |
|                 Connection is made via UVMC                      |
|                 with XLerated converters                         |
|                                                                  |
|   sv2sc2sv_xl_gp_converter_loopback:                             |
|                 SV producer <-> SC loopback                      |
|                 Connection is made via UVMC                      |
|                 with XLerated converters                         |
|                                                                  |
|   sc2sv2sc_xl_gp_converter_loopback_whole_image_payloads:        |
|                 SC producer <-> SV loopback                      |
|                 Connection is made via UVMC                      |
|                 with XLerated converters                         |
|                 and big, nasty packets                           |
|                                                                  |
|   sv2sc2sv_xl_gp_converter_loopback_whole_image_payloads:        |
|                 SV producer <-> SC loopback                      |
|                 Connection is made via UVMC                      |
|                 with XLerated converters                         |
|                 and big, nasty packets                           |
|   ---------------------------------------------------------      |
|   sc2sc2sc_uvmc_loopback:                                        |
|                 SC producer <-> SC loopback                      |
|                 Connection is made via UVMC                      |
|                 with native default uvmc_converter's             |
|                                                                  |
|   sc2sc2sc_xl_gp_converter_uvmc_loopback:                        |
|                 SC producer --> SC loopback                      |
|                 Connection is made via UVMC                      |
|                 with XLerated converters                         |
|                                                                  |
|   sc2sc2sc_xl_gp_converter_uvmc_loopback_whole_image_payloads    |
|                 SC producer <-> SC loopback                      |
|                 Connection is made via UVMC.                     |
|                 with XLerated converters                         |
|                 and big, nasty packets                           |
|   ---------------------------------------------------------      |
|   sc2sc2sc_uvmc_loopback:                                        |
|                 SC producer <-> SC loopback                      |
|                 Connection is made via UVMC                      |
|                 with native default uvmc_converter's             |
|   ---------------------------------------------------------      |
|   sv2sv2sv_uvmc_loopback:                                        |
|                 SV producer <-> SV loopback                      |
|                 Connection is made via UVMC                      |
|                 with native default uvmc_converter's             |
|                                                                  |
|   sv2sv2sv_xl_gp_converter_uvmc_loopback:                        |
|                 SV producer --> SV loopback                      |
|                 Connection is made via UVMC                      |
```

```
|                  with XLerated converters                          |
|                                                                    |
|   sv2sv2sv_xl_gp_converter_uvmc_loopback_whole_image_payloads      |
|                  SV producer <-> SV loopback                       |
|                  Connection is made via UVMC.                      |
|                  with XLerated converters                          |
|                  and big, nasty packets                            |
|                                                                    |
| UVM_HOME and UVMC_HOME specify the location of the source          |
| headers and macro definitions needed by the examples. You must     |
| specify their locations via UVM_HOME and UVMC_HOME environment     |
| variables or make command line options. Command line options       |
| override any envrionment variable settings.                        |
|                                                                    |
| The UVM and UVMC libraries must be compiled prior to running       |
| any example. If the libraries are not at their default location    |
| (UVMC_HOME/lib) then you must specify their location via the       |
| UVM_LIB and/or UVMC_LIB environment variables or make command      |
| line options. Make command line options take precedence.           |
|                                                                    |
| Other options:                                                     |
|                                                                    |
|   all   : Run all examples                                         |
|   clean : Remove simulation files and directories                  |
|   help  : Print this help information                              |
|                                                                    |
 --------------------------------------------------------------------

//------------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation                //
//   All Rights Reserved Worldwide                                  //
//                                                                  //
//   Licensed under the Apache License, Version 2.0 (the            //
//   "License"); you may not use this file except in                //
//   compliance with the License.  You may obtain a copy of         //
//   the License at                                                 //
//                                                                  //
//       http://www.apache.org/licenses/LICENSE-2.0                 //
//                                                                  //
//   Unless required by applicable law or agreed to in              //
//   writing, software distributed under the License is             //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR         //
//   CONDITIONS OF ANY KIND, either express or implied.  See        //
//   the License for the specific language governing                //
//   permissions and limitations under the License.                 //
//------------------------------------------------------------------//
```

35 Configuration extensions

35.1 Introduction

Configuration extensions are ignorable extensions (in the sense of TLM-2.0 generic payloads) that can be used to pass configurations which accompany generic payloads that travel from TLM-2.0 initiators to targets.

The UVMC config extension base class **uvmc_xl_config** contains a simple abstraction of a set configuration registers that can act as shadows of the associated configuration register set one might find in the target model.

There are two types of configuration extensions that are handled by **class uvmc_xl_config**,

- Static configurations
- Sideband configurations

35.1.1 Static configurations

- Static configurations are sent as separate dedicated transactions to update configuration register sets on the target side of the connection.
- Static configs can be used for configuring things that don't change often such as UART baud rate, AXI randomized wait state bounds and cross channel latencies.

35.1.2 Sideband configurations

- Sideband confgurations are unconditionally sent with each and every generic payload transaction along the forward path to the target.
- These should be used for things that typically change as frequently as every transaction such as tid's for AXI transactions, tags for Wishbone transactions, etc.

35.1.3 How to use configuration extensions

35.1.4 Defining your config extension classes

NOTE: The **class uvmc_xl_config** TLM GP extension is designed to be used **only** with TLM GPs passed to the **class uvmc_xl_converter** fast packer described above. You can attach them to TLM GPs that use other packers but the extension itself may not accompany the TLM GP across the TLM channel in that case (certainly not for UVMC *default converters* or **class uvmc_tlm_gp_converter** fast packers).

For the config extensions themselves see the **class uvmc_xl_config** definitions in these files,

```
src/connect/
  sc/uvmc_xl_config.*
  sv/uvmc_xl_config.svh
```

To create a custom configuration extension suitable for passing static or sideband configurations attached to TLM GPs across UVM-Connect'ions, simply define a class that derives from **class uvmc_xl_config** for each language (SC, SV).

For use by the examples to follow, we'll first define a custom configuration extension for TLM GP's that one might see being used with the AXI protocol.

Please see the section Customizing class uvmc_xl_config for AXI configuration below for more details but here we simply show how the class is defined and how the constructor is done for each language. In both cases a custom configuration extension is created for the AXI master TLM connections in the examples by deriving the custom config from **class uvmc_xl_config** as follows,

- For SC-side,

```
class AxiConfig : public uvmc::uvmc_xl_config {

  public:
    AxiConfig()
      : uvmc::uvmc_xl_config(
          ( 16*2 + 8*3 ) / 8,      // staticConfigNumBytes
          ( 32*1 + 8*1 ) / 8 ) { } // sidebandConfigNumBytes
...
```

- for SV-side,

```
class AxiConfig extends uvmc_xl_config #(
    ( 16*2 + 8*3 ) / 8,      // staticConfigNumBytes
    ( 32*1 + 8*1 ) / 8 );    // sidebandConfigNumBytes

    `uvm_object_utils( AxiConfig )
```

Note that in both cases the **class uvmc_xl_config** is dimensioned to the number of bytes in the static configuration and the number of bytes in the sideband configuration. In the case of SC the dimensions are given in the constructor whereas in the case of SV they are given with class parametrizations.

The single configuration extension class definition will handle both types of configurations in all TLM GP transaction communications.

So those are the basics of how to create a custom configuration. Again see the section Customizing class uvmc_xl_config for AXI configuration for more details about the example AXI configuration that is used in the examples and how to perform *config query* and *config update* operations on the various config register fields that are managed by the **class AxiConfig** config extension itself.

35.1.5 Running the examples

For examples that use the configuration extensions see,

```
examples/config_exts/Makefile
```

This directory contains several examples of TLM-2 UVM-Connect'ions that pass TLM-2 generic payloads (TLM GPs) between SystemC (SC) and SystemVerilog (SV).

They modified variants of tests lifted from *examples/xlerate.connections*, namely the following two,

```
TESTS = \
  sc2sv2sc_xl_gp_converter_loopback_whole_image_payloads \
  sv2sc2sv_xl_gp_converter_loopback_whole_image_payloads
```

This allows illustrations for config extension *update* or *query* operations traveling from SC -> SV -> SC or from SV -> SC -> SV respectively.

Use *make help* to view the menu of available tests,

```
make help
```

To run just one test such as *sc2sv2sc_xl_gp_converter_loopback_whole_image_payloads*,

```
make sc2sv2sc_xl_gp_converter_loopback_whole_image_payloads
```

This compiles then runs the *sc2sv2sc_xl_gp_converter_loopback_whole_image_payloads* test.

To run all tests, check them, and clean up afterwards, i.e. *sim:*, *check:*, and *clean:* targets, use the *all:* target as follows,

```
make all
```

Or you can run individual sub-targets.

The *sim* target compiles and runs all the tests.

```
make sim
```

The *check* target checks results of all the tests.

```
make check
```

The *clean* target deletes all the simulation files produced from previous runs.

```
make clean
```

You can combine targets in one command line

```
make sim check
```

The following runs the ‘sim’ target, providing the path to the UVM source and compiled library on the *make* command line.

```
make UVM_HOME=<path> UVM_LIB=<path> sim
```

```
//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
```

```
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//
```

This example demonstrates an AXI master transactor protocol specific extension to the TLM generic payload that can be used as an ignorable extension to the generic payload transactions used by AXI master initiators and targets.

36 AXI config extension SC example

36.1 Customizing class uvmc_xl_config for AXI configuration

This example demonstrates an AXI master transactor protocol specific extension to the TLM generic payload that can be used as an ignorable extension to the generic payload transactions used by AXI master initiators and targets.

Specifically it can be used to specify AXI master transactor configurations and to allow *config query* and *config update* operations on those configurations.

This allows for optionally simultaneously updating the configuration when using the generic payload for its normal purpose which is to allow the AXI master client to send WRITE or READ transactions along the forward path.

This extension is ignorable as per the definition of “ignorable extensions” in the TLM-2.0 LRM. If not specified, configurations are not updated in the transactor - just the bus transaction is sent.

So, it is possible to send a transaction to the target conduit in 4 modes:

- Send a mainstream GP transaction only
- Send a transaction + **sideband confuration** update, by specifying a config extension piggy-backed on a non-empty mainstream TLM GP
- **Update** the *static configuration only*, by specifying extension and indicating **numBytes=0, is_write()** in TLM GP (all other fields of the TLM GP are unused in this mode)
- **Query** the *static configuration only*, by specifying extension and indicating **numBytes=0, is_read()** in the TLM GP (all other fields of the TLM GP are unused in this mode)

For *static config upate* operations the new values of the register fields will be propgated from the initiator to the target.

For *static config query* operations the current values of the register fields on the target side queried (i.e. read) by the initiator.

For example an AXI master transactor IP model’s API might allow specification of multiple parameters related to the transactions and/or transactor register configurations all of which must received over the UVM-Connect’ed TLM-2 socket via the TLM GP in any of the 4 modes shown above.

If there is no corresponding pre-defined TLM GP field for a given AXI parameter (such as address, command, data, byte enables) then it is passed in a special way via TLM GP extensions as either a *static configuration* or a *sideband configuration*. The next section explains more about this in detail.

36.1.1 AXI configuration register field definitions

The following table shows our AXI master target IP’s API parameters and how they might map directly to TLM GP fields or indirectly to config extension fields,

```
AXI param      Data type      What it is     TLM GP parameter
---------      ---------      ----------     ----------------
addr           uint_64t       address        address
```

```
size           axi_size_e     beat size      streaming width
burst          axi_burst_e    burst type     sideband config (GP extension)
lock           axi_lock_e     lock type      static config (GP extension)
cache          axi_cache_e    cache type     static config (GP extension)
prot           axi_prot_e     prot type      static config (GP extension)
id             unsigned       AID            sideband config (GP extension)
burst_length   unsigned       burst length   data_length
data_words     svBitVecVal[]  payload        data_ptr
write_strobes  svBitVecVal[]  byte enables   byte_enables
resp           axi_response_e response       status response
auser_resp     svBitVecVal[]                 unsupported for now
```

Static configurations are automatically passed via the generic payload as a *separate dedicated transaction* from the intiator to transactor IP target whenever the conduit automatically detects they have changed on the intiator side.

Based on the table above, for our AXI transactor example we'll choose the following config register parameters to be represented by the *static configuration only*,

```
Static configuration data payload layout LSWs -> MSWs:
    [16] DATA_WIDTH - READ ONLY
    [16] ADDR_WIDTH - READ ONLY
    [ 8] LOCK_TYPE
    [ 8] CACHE_TYPE
    [ 8] PROT_TYPE
```

Sideband configurations are things that can change frequently (i.e. with each transaction typically). So they are "piggy backed" with each TLM GP transaction as an extension.

For our AXI transactor example we'll choose the following config register parameters to be represented by the *sideband configuration only*,

```
Sideband configuration data payload layout LSBs -> MSBs:
    [32] AID - Xaction ID
    [ 8] BURST_TYPE
```

36.1.2 class AxiConfig (SC-side definition)

Now that we've decided the width of each register field and which fields should be included as part of the static configuration vs. the sideband configuration, we can define the **class AxiConfig** itself as an extension of reusable **class uvmc_xl_config**.

```
class AxiConfig : public uvmc::uvmc_xl_config {
```

36.1.3 ::AxiConfig()

And, in the definition of its constructor we can specify the total number of bytes taken up by the static config and that taken up by the sideband config ...

```
AxiConfig()
  : uvmc::uvmc_xl_config(
      ( 16*2 + 8*3 ) / 8,      // staticConfigNumBytes
      ( 32*1 + 8*1 ) / 8 ) { } // sidebandConfigNumBytes
```

36.1.4 AXI configuration register field accessors

OK, now our **class AxiConfig** will need special accessors to access the individual fields to *set* and *get* individual register field values. Each of these accessors can use the underlying *::set_static_config(), ::get_static_config(), ::set_sideband_config()*, and *::get_sideband_config()* accessors of base **class uvmc_xl_config**.

The *::set_*()* base class accessor functions each take value, starting bit, and field width values as follows,

```
void uvmc_xl_config::set_static_config(
    unsigned value, unsigned i, unsigned w );

void uvmc_xl_config::set_sideband_config(
    unsigned value, unsigned i, unsigned w )
```

The *::get_*()* base class accessor functions each take starting bit, and field width and return values as follows,

```
unsigned uvmc_xl_config::get_static_config(
    unsigned i, unsigned w ) const

unsigned uvmc_xl_config::get_sideband_config(
    unsigned i, unsigned w ) const
```

NOTE: These are modeled after SV 1800 DPI utility functions, *svPutPartselBit()* and *svGetPartselBit()*. Each value can have a width up to, but not exceeding 32 bits. Larger widths will need to be represented with multiple fields. Doing it this way facilitates dovetailing UVM-Connect TLM fabric with DPI based APIs.

OK, here then are the specific accessors for our **class AxiConfig**,

```
// Static config 'set' accessors
void setDataWidth( unsigned dataWidth ) {
    set_static_config( dataWidth, 0, 16 ); }
void setAddrWidth( unsigned addrWidth ) {
    set_static_config( addrWidth, 16, 16 ); }

void setLockType( axi_lock_e lockType ) {
    set_static_config( (unsigned)lockType, 32, 8 ); }
void setCacheType( axi_cache_e cacheType ) {
    set_static_config( (unsigned)cacheType, 40, 8 ); }
void setProtType( axi_prot_e protType ) {
    set_static_config( (unsigned)protType, 48, 8 ); }

// Sideband config 'set' accessors
void setAid( unsigned aid ) { set_sideband_config( aid, 0, 32 ); }
void setBurstType( axi_burst_e burstType ) {
    set_sideband_config( (unsigned)burstType, 32, 8 ); }

// Configuration queries are always sent back via the return payload of
// a static configuration update with identical layout.

// Static config 'get' accessors
unsigned getDataWidth() const   { return get_static_config(  0, 16 ); }
unsigned getAddrWidth() const   { return get_static_config( 16, 16 ); }

axi_lock_e  getLockType() const{
    return (axi_lock_e)get_static_config( 32, 8 ); }
axi_cache_e getCacheType() const{
```

```
    return (axi_cache_e)get_static_config( 40, 8 ); }
axi_prot_e getProtType() const{
    return (axi_prot_e)get_static_config( 48, 8 ); }

// Sideband config 'get' accessors
unsigned getAid() const { return get_sideband_config( 0, 32 ); }
axi_burst_e getBurstType() const{
    return (axi_burst_e)get_sideband_config( 32, 8 ); }
```

37 AXI config extension SV example

For the SV-side we necessarily define all the AXi config fields exactly identically to those for the SC-side. See AXI config extension SC example for details of field definitions for static and sideband configuration extensions.

37.1 package AxiConfigPkg (SV-side definition)

For the SV-side we define a package to contain the config extension class as well as the **class AxiConfig** definition itself, again as an extension of reusable **class uvmc_xl_config**.

```
package AxiConfigPkg; // {
```

37.1.1 class AxiConfig (SV-side definition)

This is an AXI master transactor protocol specific extension to the TLM generic payload that can be used as an ignorable extension to the generic payload transactions used by AXI master initiators.

Notice that in the SV-side definition, the static and sideband configuration dimensioning is done using class parametrization (in contrast to SC-side which uses constructor args).

See comments in SC-side AxiConfig.h for more details.

```
class AxiConfig extends uvmc_xl_config #(
    ( 16*2 + 8*3 ) / 8,      // staticConfigNumBytes
    ( 32*1 + 8*1 ) / 8 );    // sidebandConfigNumBytes
// {
    `uvm_object_utils( AxiConfig )

// ...
```

37.1.2 AXI configuration register field accessors

Like its SC-side counterpart, our **class AxiConfig** will need special accessors to access the individual register fields to *set* and *get* individual register field values. See *AxiConfig.h* for more explanation of the SC-side accessors as these are defined in an identical way here.

```
// Static config 'set' accessors
function void setDataWidth( int unsigned dataWidth );
    set_static_config( dataWidth, 0, 16 ); endfunction
function void setAddrWidth( int unsigned addrWidth );
    set_static_config( addrWidth, 16, 16 ); endfunction

function void setLockType( axi_lock_e lockType );
    set_static_config( unsigned'(lockType), 32, 8 ); endfunction
function void setCacheType( axi_cache_e cacheType );
    set_static_config( unsigned'(cacheType), 40, 8 ); endfunction
function void setProtType( axi_prot_e protType );
    set_static_config( unsigned'(protType), 48, 8 ); endfunction

// Sideband config 'set' accessors
function void setAid( int unsigned aid );
    set_sideband_config( aid, 0, 32 ); endfunction
function void setBurstType( axi_burst_e burstType );
```

```
    set_sideband_config( unsigned'(burstType), 32, 8 ); endfunction

// Static config 'get' accessors
function int unsigned getDataWidth();
    return get_static_config(  0, 16 ); endfunction
function int unsigned getAddrWidth();
    return get_static_config( 16, 16 ); endfunction

function axi_lock_e getLockType();
    return axi_lock_e'(get_static_config( 32, 8 )); endfunction
function axi_cache_e getCacheType();
    return axi_cache_e'(get_static_config( 40, 8 )); endfunction
function axi_prot_e getProtType();
    return axi_prot_e'(get_static_config( 48, 8 )); endfunction

// Sideband config 'get' accessors
function int unsigned getAid();
    return get_sideband_config( 0, 32 ); endfunction
function axi_burst_e getBurstType();
    return axi_burst_e'(get_sideband_config( 32, 8 )); endfunction
```

38 SC -> SV -> SC loopback example

38.1 SC initiator and target use of config extensions

This example is was modified from one of the variations under *../xlerate.connections* to demonstrate the use of static and sideband *configuration extensions*.

Configuration extensions can be passed “piggy back” alongside the generic payload (TLM GP) transaction when it is desired to do *static configuration* register updates/queries or to pass along *sideband configuration* items.

In this case the configurations begin deployed are hypothetical configuration objects that might be used in conjunction with AXI bus protocol transactors.

The configuration extensions are used to carry ancillary parameters that need to accompany TLM GP’s to represent basic AXI bus transactions and AXI transactor configuration operations.

See AXI config extension SC example for a detailed description of the actual **class AxiConfig** config extension used in the examples blow.

In these examples we demonstrate 3 aspects of using configuration extensions,

- Demonstrate the initiator querying of “read-only” parameters in the target using *static configuration query* ops.
- Demonstrate the initiator updating config parameters in the target using *static configuration update* ops.
- Demonstrate initiator passing extra ancillary parameters with each and every TLM GP transaction using using *sideband configurations*.

In the example below, because it is demonstrating an SC -> SV -> SC loopback both the initiator and the target of each operation is implemented in the same SC **class producer** shown blow.

38.1.1 class producer - SC initiator and target

The **class producer** is an **sc_module** that defines both initiator and target TLM-2 ports since this is a loopback and the same producer plays the role of both initiator and eventual target.

In this example the static config TLM-2 channel is made separate from the “mainstream” TLM-2 channel although this does not need to be the case. Because static configs are handled as distinct operations, a single TLM-2 channel can be overloaded for use with static config ops and mainstream TLM GP transation ops (with optionally “piggy backed” sideband configs).

```
class producer : public sc_module {

  public:
    simple_initiator_socket<producer> out;        // "mainstream" port
    simple_initiator_socket<producer> out_config; // static config port
    simple_target_socket<producer> in;
    simple_target_socket<producer> in_config;
```

38.1.2 ::run()

This is the test thread that implements the *initiator* function of the example.

At the high level it performs the following ops,

- Learn (query) the static config from the target.
- Update the static config back to the target with desired persistent register field settings.
- Re-learn (query) the static config from the target to show that static config registers were updated from previous op.
- Set up BURST_TYPE sideband config value that will be used for all iterations of the main test loop.
- foreach of NUM_TRANSACTIONS

```
- Increment AID sideband config value for this transaction
- Set up remaining main TLM GP fields for transaction
- Send transaction to target by calling socket's ::b_transport()
```

- Check that checksum of data sent matches that of data received by target.

38.1.3 ::b_transport()

This is the target's implementation of the mainstream **::b_transport()** method to process all TLM GP transactions in the SC -> SV -> SC loopback test.

Mainly it just updates the checksum that will be checked at the end of the test with the data contents.

But notice how it also accesses the **AID** sideband config parameter accompanying the main TLM GP transaction and reflects that in the checksum as well.

```
virtual void b_transport(tlm_generic_payload &gp, sc_time &t) {
    char unsigned *data = gp.get_data_ptr();

    AxiConfig *configExt;

    gp.get_extension( configExt );

    actualChecksum += configExt->getAid();
    for( unsigned long long i=0; i<gp.get_data_length(); i++ )
        actualChecksum += data[i];

    wait(t);
    t = SC_ZERO_TIME;
    gp.set_response_status( tlm::TLM_OK_RESPONSE );
}
```

38.1.4 ::nb_transport_fw()

This is the target's implementation of the non-blocking **::nb_transport_fw()** used only for the static config target port.

If it is a *static config query* (READ op), it copies the local target's config extension object into the static config extension member of the passed in TLM GP for return back to the initiator.

Otherwise if it is a *static config update* (WRITE op), it from copies from the static config extension member of the passed in TLM GP object arriving from the initiator, into the local target's config extension object.

```
tlm::tlm_sync_enum nb_transport_fw(
    tlm::tlm_generic_payload &trans,
    tlm::tlm_phase &phase,
    sc_time &delay )
{
    int status;
    AxiConfig *configExtension;

    // Standard checks that pure TLM-2.0 base protocol rules are
    // being properly followed ...
    if( phase != BEGIN_REQ )
        errorOnTransportPhase(
            "producer::nb_transport_fw()", "BEGIN_REQ", phase,
            __LINE__, __FILE__ );

    // Innocent until proven guilty.
    trans.set_response_status( tlm::TLM_OK_RESPONSE );

    trans.get_extension( configExtension );

    // If a configuration extension has been passed alongside the
    // generic payload transaction then it is possible we're doing
    // a static configuration register update or query

    if( configExtension ) {

        // if( trans.get_data_length() == 0 )
        //     We assume static config if data length=0
        if( trans.get_data_length() == 0 ) {
            // if( we're just querying the static configuration )
            //   We can just transfer it from the local target
            //   config extension.
            if( trans.is_read() )
                configExtension->copy_from( targetConfig );

            // else we assume a config update.
            else targetConfig.copy_from( *configExtension );

            return tlm::TLM_COMPLETED;
        }
    }

    // Proven guilty ! (This function now only handles
    // static config updates/queries - use ::b_transport() for actual
    // LT-mode-only mainstream transactions.)
    trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );

    return tlm::TLM_COMPLETED;
}
```

38.1.5 ::learnBusParameters()

Send a transaction to QUERY the target configuration. Full configuration will arrive in the generic payload on the on the return path of this **nb_transport_fw()** call.

Notice the convenient use of the base class **uvmc_xl_config::query_trans()** method that returns a *pre-fab'ed* TLM GP container that can conveniently be used (and reused) as a carrier strictly for the purpose of performing *config query* operations.

```
void learnBusParameters( AxiConfig &config ) {

    sc_time delay = SC_ZERO_TIME;
    tlm_phase phase = BEGIN_REQ;

    if( out_config->nb_transport_fw(
            config.query_trans(), phase, delay ) != TLM_COMPLETED ||
            config.query_trans().get_response_status() != TLM_OK_RESPONSE ){
        errorOnTransport( "producer::learnBusParameters()",
            __LINE__, __FILE__ );
        return;
    }

    cout << sc_time_stamp()
         << " [PRODUCER/CONFIG/QUERY]"
         << " ADDR_WIDTH=" << config.getAddrWidth()
         << " DATA_WIDTH=" << config.getDataWidth()
         << " LOCK_TYPE=" << config.getLockType()
         << " CACHE_TYPE=" << config.getCacheType()
         << " PROT_TYPE=" << config.getProtType()
         << endl;
}
```

38.1.6 ::updateTargetConfig()

Send a transaction to UPDATE the initial target configuration.

Again, notice use of the **::update_trans()** method of the base **class uvm_xl_config** as a convenient pre-configured and reusable TLM GP that acts as a carrier for the *config update* operations.

```
void updateTargetConfig( AxiConfig &config ) {

    sc_time delay = SC_ZERO_TIME;
    tlm_phase phase = BEGIN_REQ;

    if( out_config->nb_transport_fw(
            config.update_trans(), phase, delay ) != TLM_COMPLETED ||
            config.update_trans().get_response_status()
                != TLM_OK_RESPONSE )
        errorOnTransport( "producer::updateTargetConfig()",
            __LINE__, __FILE__ );
      return;
}
```

39 SV -> SC -> SV loopback example

39.1 SV initiator and target use of config extensions

This example is was modified from one of the variations under *../xlerate.connections* to demonstrate the use of static and sideband *configuration extensions*.

Configuration extensions can be passed “piggy back” alongside the generic payload (TLM GP) transaction when it is desired to do *static configuration* register updates/queries or to pass along *sideband configuration* items.

In this case the configurations being deployed are hypothetical configuration objects that might be used in conjunction with AXI bus protocol transactors.

The configuration extensions are used to carry ancillary parameters that need to accompany TLM GP’s to represent basic AXI bus transactions and AXI transactor configuration operations.

See AXI config extension SV example for a detailed description of the actual **class AxiConfig** config extension used in the examples blow.

In these examples we demonstrate 3 aspects of using configuration extensions,

- Demonstrate the initiator querying of “read-only” parameters in the target using *static configuration query* ops.
- Demonstrate the initiator updating config parameters in the target using *static configuration update* ops.
- Demonstrate initiator passing extra ancillary parameters with each and every TLM GP transaction using using *sideband configurations*.

In the example below, because it is demonstrating an SV -> SC -> SV loopback both the initiator and the target of each operation is implemented in the same SV **class producer** shown blow.

39.1.1 class producer - SV initiator and target

The **class producer** is a **uvm_component** that defines both initiator and target TLM-2 ports since this is a loopback and the same producer plays the role of both initiator and eventual target.

In this example the static config TLM-2 channel is made separate from the “mainstream” TLM-2 transaction channel because of a limitation in SV-UVM that the same ports cannot be used for both blocking and non-blocking transport operations. And because static config query/updates are done using non-blocking transports while mainstream transactions use blocking transports, two sets of ports are created.

```
class producer extends uvm_component; // {

    `uvm_component_utils(producer)

    // All ports default to TLM GP as transaction kind.
    uvm_tlm_b_initiator_socket  #()           out;        // "mainstream" port
    uvm_tlm_nb_initiator_socket #( producer ) out_config; // static config port
    uvm_tlm_b_target_socket     #( producer ) in;
    uvm_tlm_nb_target_socket    #( producer ) in_config;
```

39.1.2 ::run()

This is the test thread that implements the *initiator* function of the example.

At the high level it performs the following ops,

- Learn (query) the static config from the target.
- Update the static config back to the target with desired persistent register field settings.
- Re-learn (query) the static config from the target to show that static config registers were updated from previous op.
- Set up BURST_TYPE sideband config value that will be used for all iterations of the main test loop.
- foreach of NUM_TRANSACTIONS

```
- Increment AID sideband config value for this transaction
- Set up remaining main TLM GP fields for transaction
- Send transaction to target by calling socket's ::b_transport()
```

- Check that checksum of data sent matches that of data received by target.

39.1.3 ::b_transport()

This is the target's implementation of the mainstream **::b_transport()** method to process all TLM GP transactions in the SV -> SC -> SV loopback test.

Mainly it just updates the checksum that will be checked at the end of the test with the data contents.

But notice how it also accesses the **AID** sideband config parameter accompanying the main TLM GP transaction and reflects that in the checksum as well.

```
virtual task b_transport( uvm_tlm_gp t, uvm_tlm_time delay );

    AxiConfig configExt = new();
    uvmc_xl_config_base configExtension;

    $cast( configExtension, t.get_extension(uvmc_xl_config_base::ID()) );
    assert( configExtension != null );

    configExt.copy( configExtension );

    actualChecksum += configExt.getAid();
    for( int unsigned i=0; i < t.get_data_length(); i++ )
        actualChecksum += t.m_data[i];

    #(delay.get_abstime(1e-9));
    delay.reset();
    t.set_response_status( UVM_TLM_OK_RESPONSE );
endtask
```

39.1.4 ::nb_transport_fw()

This is the target's implementation of the non-blocking **::nb_transport_fw()** used only for the static config target port.

If it is a *static config query* (READ op), it copies the local target's config extension object into the static config extension member of the passed in TLM GP for return back to the initiator.

Otherwise if it is a *static config update* (WRITE op), it from copies from the static config extension member of the passed in TLM GP object arriving from the initiator, into the local target's config extension object.

```
virtual function uvm_tlm_sync_e nb_transport_fw(
    uvm_tlm_gp trans, ref uvm_tlm_phase_e phase,
    input uvm_tlm_time delay );
// {
    int status;
    uvmc_xl_config_base configExtension;

    // Standard checks that pure TLM-2.0 base protocol rules are
    // being properly followed ...
    if( phase != BEGIN_REQ )
        `uvm_error( get_type_name(),
            $psprintf(
                "Phase error on transport socket '%s' expectedPhase=%s actualPhase=%0d",
                "producer::nb_transport_fw()",
                "BEGIN_REQ", int'(phase) ) )

    // Innocent until proven guilty.
    trans.set_response_status( UVM_TLM_OK_RESPONSE );

    // If a configuration extension has been passed alongside the
    // generic payload transaction then it is possible we're doing
    // a static configuration register update or query

    $cast( configExtension, trans.get_extension(uvmc_xl_config_base::ID()) );

    if( configExtension != null ) begin // {

        // if( trans.get_data_length() == 0 )
        //     We assume static config if data length=0
        if( trans.get_data_length() == 0 ) begin // {

            // if( we're just querying the static configuration )
            //   We can just transfer it from the local target
            //   config extension.
            if( trans.is_read() )
                configExtension.copy( targetConfig );

            // else we assume a config update.
            else targetConfig.copy( configExtension );

            return UVM_TLM_COMPLETED;
        end // }
    end // }

    // Proven guilty ! (This function now only handles
    // static config updates/queries - use ::b_transport() for actual
    // LT-mode-only mainstream transactions.)
    trans.set_response_status( UVM_TLM_GENERIC_ERROR_RESPONSE );

    return UVM_TLM_COMPLETED;
endfunction // }
```

39.1.5 ::learnBusParameters()

Send a transaction to QUERY the target configuration. Full configuration will arrive in the generic payload on the on the return path of this **nb_transport_fw()** call.

Notice the convenient use of the base class **uvmc_xl_config::query_trans()** method that returns a *pre-fab'ed* TLM GP container that can conveniently be used (and reused) as a carrier strictly for the purpose of performing *config query* operations.

```
function void learnBusParameters( AxiConfig configExt ); // {

    uvm_tlm_time delay = new("del",1e-9);
    uvm_tlm_phase_e phase = BEGIN_REQ;
    uvm_tlm_generic_payload q = configExt.query_trans();

    if( out_config.nb_transport_fw( q, phase, delay )
            != UVM_TLM_COMPLETED
        || q.get_response_status() != UVM_TLM_OK_RESPONSE )
    begin
        `uvm_error( get_type_name(),
            $psprintf(
                "Error on transport socket '%s' ",
                "producer::learnBusParameters()" ) )
        return;
    end

    `uvm_info( get_type_name(),
        $psprintf( "[PRODUCER/CONFIG/QUERY] ADDR_WIDTH=%0d DATA_WIDTH=%0d LOCK_TYPE=%0d
            configExt.getAddrWidth(),
            configExt.getDataWidth(),
            configExt.getLockType(),
            configExt.getCacheType(),
            configExt.getProtType() ), UVM_LOW )
endfunction // }
```

39.1.6 ::updateTargetConfig()

Send a transaction to UPDATE the initial target configuration.

Again, notice use of the **::update_trans()** method of the base **class uvm_xl_config** as a convenient pre-configured and reusable TLM GP that acts as a carrier for the *config update* operations.

```
function void updateTargetConfig( AxiConfig configExt ); // {

    uvm_tlm_time delay = new("del",1e-9);
    uvm_tlm_phase_e phase = BEGIN_REQ;
    uvm_tlm_generic_payload u = configExt.update_trans();

    if( out_config.nb_transport_fw( u, phase, delay )
            != UVM_TLM_COMPLETED
        || u.get_response_status() != UVM_TLM_OK_RESPONSE )

        `uvm_error( get_type_name(), $psprintf(
            "Error on transport socket '%s' ",
            "producer::updateTargetConfig()" ) )
endfunction // }
```

For the SV-side we necessarily define all the AXi config fields exactly identically to those for the SC-side.

40 UVMC Type Support

The members of your transaction definitions may be any collection of the following types, which have direct support in UVMC. For any type not listed, there is likely a supported type to which your converter can adapt.

40.1 Supported Data Types

The following types are supported by UVMC for packing and unpacking via the streaming operators.

```
SV                  |   SC

longint             |   long long
int                 |   int
shortint            |   short
byte                |   char
bit                 |   bool
                    |
longint unsigned    |   unsigned long long
int unsigned        |   unsigned int
shortint unsigned   |   unsigned short
byte unsigned       |   unsigned char
bit unsigned        |   bool
                    |
shortreal           |   float
real                |   double
string              |   string
time                |   sc_time
                    |
T                   |   T
T arr[N]            |   T arr[N];
T q[$]              |   vector<T>
T da[]              |   list<T>
T aa[KEY]           |   map<KEY,T>
                    |
sc_bit              |   sc_bit
sc_logic            |   sc_logic
sc_lv [L:R]         |   sc_lv<N>
sc_bv [L:R]         |   sc_bv<N>
                    |
bit [N-1:0]         |   sc_int<N>
bit [N-1:0]         |   sc_uint<N>
bit [N-1:0]         |   sc_bigint<N>
bit [N-1:0]         |   sc_biguint<N>
```

40.1.1 Choosing the type mappings

- Each row in the table shows a *suggested* mapping between SV and SC types. Many other mappings are possible. For example, you can unpack an *int* from the source language into many types other than *int* in the target language, *char[4]*, *sc_lv<32>*, and *bit [31:0]* to name a few.
- Be sure to consider platform dependencies such as 32-bit versus 64-bit, and big versus little endian.
- When you have the freedom to define an equivalent transaction type from scratch, you would typically define the transaction to have the same number of fields with equivalent bit sizes.
- When both of the transaction objects in SV and SC are pre-existing, they may not have the exact same number of members of equivalent types declared in the exact same order. With UVM Connect, your ability to define custom converters allows you to get your models communicating despite the

disparate transaction types.
- Conversion can even span multiple members of the target transaction type. That int might represent a 32-bit address on the source side, and two 16-bit high/low addresses on the target side.
- Conversion can map between fields of different bit widths. For example, a 32-bit address can be mapped to a 16-bit address, as long as steps are taken to avoid truncation of the larger field.

40.1.1.1 Usage Limitations

Integral size	Each integral field is limited to 4K bits
Total size	Each packed transaction is limited to 4K bytes
Arrays	Supports one dimension only
`uvm_field macros	Instead of implementing the *do_** methods directly, you may opt to employ the `uvm_field macros, which expand into code that implements these operations for most types. This is not recommended. The field macros expand into large amounts of code that affect run-time performance in every simulation in which they are used. They limit the upper-bound on accelleration performance. The code they expand into is unreadable and may hinder debug should you ever have to step through it. For details on these and other costs associated with `uvm_field and all other UVM macros, see the white paper, *OVM/UVM Macros: A Cost-Benefit Analysis*, at http://www.verificationacademy.com

40.1.2 Type-Support Examples

How to run the example demonstrating type support

See <Getting Started> for setup requirements for running the examples.

40.1.2.1 Use *make help* to view the menu of available examples

```
> make help
```

Choose an example to run from the menu, say

```
> make all
```

This compiles and runs an example demonstrating packing and unpacking of all directly supported data types allowed as members of your transaction classes. Other types require conversion to the supported types.

```
//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
```

```
//------------------------------------------------------------//
```

41 Data Type Support

This example shows a full implementation of a UVM-compliant transaction type whose fields represent all the data types supported by the UVMC library. Although the example defines all the *do_** operations, UVMC requires only *do_pack* and *do_unpack*.

41.1 packet

Defines a packet class containing a field for each of the data types supported by UVMC.

integrals	bit, byte, shortint, int, longint long, and their unsigned counterparts.
enum	user-defined enumeration types are packed by their numeric value. A compatible enumeration type must be defined on the SC side.
reals	shortreal and real translate to SC-side float and double, respectively.
strings	use only for ASCII strings. Use vector<char> for an array of bytes whose elements can include the '0' value
arrays	fixed arrays, queues, dynamic arrays, and associate arrays are supported as long as the element and key types are among the supported types. These types are analogous to the STL vector<T>, list<T>, and map<KEY,T> types in C++.
time	packed as 64-bit values. Equates to the SC 'sc_time' type.
sc data types	The SV bit-vector and logic-vector types are mapped to any of the following SC built-in types: sc_bit, sc_logic, sc_bv<N>, sc_lv<N>, sc_int<N>, sc_uint<N>, sc_bigint<N>, and sc_biguint<N>, for any valid width, N. For any given N, the SV-side declaration should be *bit [N-1:0] var_name*. See the SC-side definition of *packet*.

Although do_pack and do_unpack are the only methods required by UVMC, this example also implements do_copy, do_compare, do_print, and do_record. There are many reasons for opting to implement the *do_** methods as below instead of using the *`uvm_field* macros. For details, see the white paper, *OVM/UVM Macros: A Cost-Benefit Analysis*, at http://www.verificationacademy.com.

```
class packet extends uvm_object;

   // default converter assumes trans derives from uvm_object
   `uvm_object_utils(packet)

   function new(string name="");
     super.new(name);
   endfunction

    typedef enum { ADD, SUBTRACT, MULTIPLY, DIVIDE } cmds_t;

    rand cmds_t            enum32;

    rand longint           int64;
    rand int               int32;
    rand shortint          int16;
    rand byte              int8;
    rand bit               int1;

    rand longint unsigned  uint64;
    rand int unsigned      uint32;
    rand shortint unsigned uint16;
    rand byte unsigned     uint8;
    rand bit unsigned      uint1;
```

```
     real              real64;

rand time              time64;

     string            str;

rand int               arr[3];
rand byte              q[$];
rand shortint          da[];
     shortint          aa[shortint];

rand sub_object        obj;

rand bit               scbit;
rand logic             sclogic;
rand bit [16:0]        scbv;
rand logic [34:0]      sclv;
rand bit [5:0]         scint;
rand bit [24:0]        scuint;
rand bit [36:0]        scbigint;
rand bit [61:0]        scbiguint;


constraint C_q_size  { q.size  inside {[1:11]}; }
constraint C_da_size { da.size inside {[1:11]}; }
```

41.1.1 Methods

41.1.2 do_pack

Converts this transaction's contents into a form transferrable outside SystemVerilog.

Each field is packed using the smaller, more efficient packing macros included in UVM. Associative arrays are packed by first packing its size in a 32-bit value. Then, you pack each key-value pair use the macro appropriate for their types.

Subobjects are packed by calling *packer.pack_object(subob);*.

If your transaction extends a base class with its own fields, call *super.do_pack(packer)* before packing anything in this class.

```
virtual function void do_pack(uvm_packer packer);

  `uvm_pack_int(enum32)
  `uvm_pack_int(int64)
  `uvm_pack_int(int32)
  `uvm_pack_int(int16)
  `uvm_pack_int(int8)
  `uvm_pack_int(int1)
  `uvm_pack_int(uint64)
  `uvm_pack_int(uint32)
  `uvm_pack_int(uint16)
  `uvm_pack_int(uint8)
  `uvm_pack_int(uint1)
  `uvm_pack_int(scbit)
  `uvm_pack_int(sclogic)
  `uvm_pack_int(scbv)
```

```
    `uvm_pack_int(sclv)
    `uvm_pack_int(scint)
    `uvm_pack_int(scuint)
    `uvm_pack_int(scbigint)
    `uvm_pack_int(scbiguint)
    `uvm_pack_int(time64)
    `uvm_pack_real(real64)
    `uvm_pack_string(str)
    `uvm_pack_sarray(arr)
    `uvm_pack_queue(q)
    `uvm_pack_array(da)

    `uvm_pack_intN(aa.size(),32)
    foreach (aa[i]) begin
      `uvm_pack_intN(i,16)
      `uvm_pack_intN(aa[i],16)
    end

    packer.pack_object(obj);

  endfunction
```

41.1.3 do_unpack

Converts a bit-vector representation of a transaction into this transaction object.

The order and manner of unpacking must be identical to how packing was performed. Packing an object then unpacking into a new instance of the object should be equivalent to copying the original object, minus any fields that were not packed.

Each field is unpacked using the smaller, more efficient unpacking macros included in UVM.

Associative arrays are packed by first packing its size in a 32-bit value. Then, you pack each key-value pair use the macro appropriate for their types.

To unpack sub-objects, first call *packer.is_null()* to determine if the packed sub-object is null or now. If not, you set this transaction's sub-object to null. If *is_null* returns 0, then alloate *obj* if its null and call *packer.unpack_object()*.

If your transaction extends a base class with its own fields, call *super.do_unpack(packer)* before unpacking anything in this class.

Instead of the macros, you could call methods of the packer for every field, just as you do for sub-objects. However, packing of built-in integral types is less efficient, and there is no methods for unpacking arrays.

```
  virtual function void do_unpack(uvm_packer packer);

    int unsigned n;

    `uvm_unpack_enum(enum32,cmds_t)
    `uvm_unpack_int(int64)
    `uvm_unpack_int(int32)
    `uvm_unpack_int(int16)
    `uvm_unpack_int(int8)
    `uvm_unpack_int(int1)
    `uvm_unpack_int(uint64)
```

```
    `uvm_unpack_int(uint32)
    `uvm_unpack_int(uint16)
    `uvm_unpack_int(uint8)
    `uvm_unpack_int(uint1)
    `uvm_unpack_int(scbit)
    `uvm_unpack_int(sclogic)
    `uvm_unpack_int(scbv)
    `uvm_unpack_int(sclv)
    `uvm_unpack_int(scint)
    `uvm_unpack_int(scuint)
    `uvm_unpack_int(scbigint)
    `uvm_unpack_int(scbiguint)
    `uvm_unpack_int(time64)
    `uvm_unpack_real(real64)
    `uvm_unpack_string(str)
    `uvm_unpack_sarray(arr)
    `uvm_unpack_queue(q)
    `uvm_unpack_array(da)

    aa.delete();
    `uvm_unpack_int(n)
    for (int i=0; i<n; i++) begin
      shortint k, val;
      `uvm_unpack_int(k)
      `uvm_unpack_int(val)
      aa[k]=val;
    end

    if (packer.is_null())
      obj = null;
    else begin
      if (obj == null)
        obj = new();
      packer.unpack_object(obj);
    end

  endfunction
```

41.1.4 do_copy

Copies the values of fields from another object of the same type into this object.

Do_copy uses the assignment operator for all built-in types. Subobjects are copied by calling *subobj.copy(_rhs.subobj)*.

If your transaction extends a base class with its own fields, call *super.do_copy(rhs)* before copying anything in this class.

```
  function void do_copy(uvm_object rhs);

    packet _rhs;
    assert($cast(_rhs,rhs));

    enum32    = _rhs.enum32;
    int64     = _rhs.int64;
    int32     = _rhs.int32;
    int16     = _rhs.int16;
    int8      = _rhs.int8;
    int1      = _rhs.int1;
```

```
  uint64    = _rhs.uint64;
  uint32    = _rhs.uint32;
  uint16    = _rhs.uint16;
  uint8     = _rhs.uint8;
  uint1     = _rhs.uint1;
  time64    = _rhs.time64;
  str       = _rhs.str;
  arr       = _rhs.arr;
  q         = _rhs.q;
  da        = _rhs.da;
  aa        = _rhs.aa;
  real64    = _rhs.real64;
  scbit     = _rhs.scbit;
  sclogic   = _rhs.sclogic;
  scbv      = _rhs.scbv;
  sclv      = _rhs.sclv;
  scint     = _rhs.scint;
  scuint    = _rhs.scuint;
  scbigint  = _rhs.scbigint;
  scbiguint = _rhs.scbiguint;
  obj.copy(_rhs.obj);
endfunction
```

41.1.5 do_compare

Compares the values of fields with those of another object of the same type, returning 1 if a match, 0 otherwise.

Use the boolean equality operator (==) for most built-in types. It is more efficient than the provided methods in the *comparer* policy class.

If you opt to employ direct comparison with the == operator as in this example, you must still set the *comparer.result* to 0 if there were no miscompares, or to if there was a miscompare.

Leverage short-circuit expression evaluation for higher efficiency. Expression evaluation stops as soon as a result is certain. For example, given an expression *(a && b && c && d)*, if *a* or *b* are 0, the whole expression evaluates to 0, so there is no point in examining *c* or *d*. The expression evaluates to 0 no matter what their values. In this *packet* class, if the *int64* property doesn't match with the right hand side, the remaining 25 equality comparisons that follow are not evaluated, thus speeding up the comparison operation considerably.

Comparison of scalars is more efficient than comparison of arrays and other composite types (e.g. sub-objects). So, put composite types at the end of the expression. That way, these types will not be compared unless all the previous expression terms evaluate to true.

Subobjects are compared by calling *subobj.compare(_rhs.subobj,comparer)*.

If your transaction extends a base class with its own fields, call *super.do_compare(rhs,comparer)* before comparing any fields of this class. If *super.do_compare* returns 0, return immediately with 0. Otherwise, continue with comparing this transaction's fields.

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);

  packet _rhs;
  assert($cast(_rhs,rhs));
  comparer.result = 1;
```

```
    do_compare =
            enum32    == _rhs.enum32 &&
            int64     == _rhs.int64 &&
            int32     == _rhs.int32 &&
            int16     == _rhs.int16 &&
            int8      == _rhs.int8 &&
            int1      == _rhs.int1 &&
            uint64    == _rhs.uint64 &&
            uint32    == _rhs.uint32 &&
            uint16    == _rhs.uint16 &&
            uint8     == _rhs.uint8 &&
            uint1     == _rhs.uint1 &&
            time64    == _rhs.time64 &&
            str       == _rhs.str &&
            arr       == _rhs.arr &&
            q         == _rhs.q &&
            da        == _rhs.da &&
            scbit     == _rhs.scbit &&
            sclogic   == _rhs.sclogic &&
            scbv      == _rhs.scbv &&
            sclv      == _rhs.sclv &&
            scint     == _rhs.scint &&
            scuint    == _rhs.scuint &&
            scbigint  == _rhs.scbigint &&
            scbiguint == _rhs.scbiguint &&
            $realtobits(real64) == $realtobits(_rhs.real64)
            ;

    if (!do_compare)
      return 0;

    // temporary limitation: assoc arrays must be compared "item by item"
    if (aa.size() != _rhs.aa.size())
      return 0;
    foreach (aa[i])
      if (!(_rhs.aa.exists(i) && aa[i] == _rhs.aa[i]))
        return 0;

    // compare sub-object deeply
    if (obj == null) begin
      if (_rhs.obj != null)
        return 0;
    end
    else
      do_compare = obj.compare(_rhs.obj,comparer);

    comparer.result = 1-do_compare;

  endfunction
```

41.1.6 do_print

Implements printing of all fields in this transaction using the provided *printer* policy class.

To cut down on repetitive typing, small macros are used to “inline” verbose calls to *printer.print_generic*.

If your transaction extends a base class with its own fields, call *super.do_print(printer)* before printing any fields of this class.

```
virtual function void do_print(uvm_printer printer);

  `define do_print_int(VAR,TYP,SZ) \
    printer.print_generic(`"VAR`", `"TYP`",SZ,$sformatf("'h%h",VAR));

  `define do_print_ele(VAR,TYP,SZ) \
    printer.print_generic($sformatf("[%0d]",i),\
          `"TYP`",SZ,$sformatf("'h%h",VAR));

  printer.print_generic("cmd","cmds_t",32,enum32.name());
  `do_print_int(int64,     longint,           64)
  `do_print_int(int32,     int,               32)
  `do_print_int(int16,     shortint,          16)
  `do_print_int(int8,      byte,               8)
  `do_print_int(int1,      bit,                1)
  `do_print_int(uint64,    longint unsigned,  64)
  `do_print_int(uint32,    int unsigned,      32)
  `do_print_int(uint16,    shortint unsigned, 16)
  `do_print_int(uint8,     byte unsigned,      8)
  `do_print_int(uint1,     bit unsigned,       1)
  `do_print_int(scbit,     bit,                1)
  `do_print_int(sclogic,   logic,              1)
  `do_print_int(scbv,      bit[16:0],         17)
  `do_print_int(sclv,      bit[34:0],         35)
  `do_print_int(scint,     bit[5:0],           6)
  `do_print_int(scuint,    bit[24:0],         25)
  `do_print_int(scbigint,  bit[36:0],         37)
  `do_print_int(scbiguint, bit[61:0],         62)
  printer.print_time   ("time64",time64);
  printer.print_real   ("real64",real64);
  printer.print_string ("str",  str);

  // print arrays one element at a time, between a header
  // and footer
  printer.print_array_header("arr",3,"int[3]");
  foreach (arr[i])
    `do_print_ele(arr[i],int,32)
  printer.print_array_footer(3);

  printer.print_array_header("q",q.size(),"byte[$]");
  foreach (q[i])
    `do_print_ele(q[i],byte,8)
  printer.print_array_footer(q.size());

  printer.print_array_header("da",da.size(),"shortint[]");
  foreach (da[i])
    `do_print_ele(da[i],shortint,16)
  printer.print_array_footer(da.size());

  printer.print_array_header("aa",aa.num(),"shortint[shortint]");
  foreach (aa[i])
    `do_print_ele(aa[i],shortint,16)
  printer.print_array_footer(aa.num());

  printer.print_object("obj",obj);

endfunction
```

41.1.7 do_record

Records all members of this transaction class for later viewing in the GUI's wave window.

This implementation uses the small *uvm_record_field* macro to record most field types. Arrays are recorded iteratively using the same macro.

To record a subobject, call the recorder's *record_object* method.

The component's *end_tr* method indirectly calls this method, but only if its *recording_detail* configuration parameter is set to something above UVM_NONE.

If your transaction extends a base class with its own fields, call *super.do_record(recorder)* before recording any fields of this class.

```
virtual function void do_record(uvm_recorder recorder);
  int unsigned real_bits32;
  int unsigned n;
  `uvm_record_field("enum32",enum32)
  `uvm_record_field("int64",int64)
  `uvm_record_field("int32",int32)
  `uvm_record_field("int16",int16)
  `uvm_record_field("int8",int8)
  `uvm_record_field("int1",int1)
  `uvm_record_field("uint64",uint64)
  `uvm_record_field("uint32",uint32)
  `uvm_record_field("uint16",uint16)
  `uvm_record_field("uint8",uint8)
  `uvm_record_field("uint1",uint1)
  `uvm_record_time("time64",time64)
  `uvm_record_field("real64",real64)
  `uvm_record_string("str",str)
  foreach(arr[i])
    `uvm_record_field($sformatf("arr[%0d]",i),arr[i])
  foreach(q[i])
    `uvm_record_field($sformatf("q[%0d]",i),q[i])
  foreach(da[i])
    `uvm_record_field($sformatf("da[%0d]",i),da[i])
  foreach (aa[i]) begin
    string val = $sformatf("'h%h",aa[i]);
    `uvm_record_field($sformatf("aa[%0d]",i),val)
  end
  recorder.record_object("obj",obj);

  `uvm_record_field("scbit",scbit);
  `uvm_record_field("sclogic",sclogic);
  `uvm_record_field("scbv",scbv);
  `uvm_record_field("sclv",sclv);
  `uvm_record_field("scint",scint);
  `uvm_record_field("scuint",scuint);
  `uvm_record_field("scbigint",scbigint);
  `uvm_record_field("scbiguint",scbiguint);
endfunction
```

41.1.8 pre_randomize

Randomizes the string variable, *str*, and associative array, *aa*.

For strings, we randomize its length to be within a narrow range, then randomize each character to be within the range of printable characters.

For the associative array, we randomize its size (number of entries) to be within a narrow range, then randomize each key/value pair.

```
function void pre_randomize();
  int aa_size;
  int str_size;

  // randomize assoc array
  void'(std::randomize(aa_size) with { aa_size inside {[4:11]}; });
  aa.delete();
  for (int i=0; i < aa_size; i++) begin
    shortint key;
    shortint val;
    key = $urandom;
    val = $urandom;
    aa[key] = val;
  end

  // randomize string
  void'(std::randomize(str_size) with { str_size inside {[4:11]}; });
  str = "";
  for (int i=0; i < str_size; i++) begin
    byte ele;
    void'(std::randomize(ele) with { ele inside {[32:126]}; });
$sformat(str, "%s%x", str, ele); // str = {str, string'(ele)};
  end

  // allocate sub-object
  if (obj == null)
     obj = new("obj");

endfunction
```

41.1.9 pre_randomize

Provides rough randomization of real and shortreal fields by casting randomized integrals to reals then taking their quotients.

```
    function void post_randomize();
      // reals derive from quotient of two randomized ints
      real64 = real'(uint64) / real'(uint32);
    endfunction

endclass : packet
```

41.2 producer

A simple producer that generates packet transactions and sends them out its *out* blocking-put and *ap* analysis ports.

```
class producer extends uvm_component;

   uvm_blocking_put_port #(packet) out;
   uvm_analysis_port #(packet) ap;

   `uvm_component_utils(producer)

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      out = new("out", this);
      ap = new("ap", this);
   endfunction : new

   task run_phase (uvm_phase phase);

     packet pkt;

     phase.raise_objection(this);

     for (int i = 1; i <= NUM_PKTS; i++) begin
       pkt = new;
       assert(pkt.randomize());

       `uvm_info("PRODUCER/SEND",
          $sformatf("Sending packet #%0d",i),UVM_MEDIUM)
          pkt.print();

       ap.write(pkt);
       out.put(pkt);

     end

       `uvm_info("PRODUCER/STOP", "Stopping the test", UVM_LOW);

     phase.drop_objection(this);

   endtask

endclass
```

41.3 scoreboard

A simple scoreboard that implements in-order comparison of *expect* and *actual* packets. The *expect* packets arrive via the *expect_in* analysis export. They are stored in an internal FIFO for later comparison with incoming *actual* packets, which arrive on its *actual_in* analysis imp. Strict comparison is performed as each *actual* arrives.

```
`uvm_analysis_imp_decl(_expect)
`uvm_analysis_imp_decl(_actual)

class scoreboard extends uvm_component;

   packet expect_q[$];
   uvm_analysis_imp_expect  #(packet, scoreboard) expect_in;
   uvm_analysis_imp_actual  #(packet, scoreboard) actual_in;

   `uvm_component_utils(scoreboard)

   function new(string name, uvm_component parent=null);
```

```
      super.new(name,parent);
      actual_in   = new("actual_in", this);
      expect_in   = new("expect_in", this);
    endfunction : new

    virtual function void write_expect(packet t);
      expect_q.push_back(t);
      if (run_ph.phase_done.get_objection_count(this) == 0)
        run_ph.raise_objection(this, "Scoreboard has expect packet");
    endfunction

    virtual function void write_actual(packet t);
      packet exp;

      `uvm_info("SCOREBD/RECV_ACTUAL",
             $sformatf("SV scoreboard recevied actual:\n  %p",t),UVM_HIGH);

      if (expect_q.size() == 0)
        uvm_report_fatal("SCOREBD/NO_EXPECT",
          $psprintf("%m: No expect packet to compare with incoming actual."));

      exp = expect_q.pop_front();

      if (!exp.compare(t))
        uvm_report_error("SCOREBD/MISCOMPARE",
          $psprintf("Actual does not match expect:\nexpect=%p\nactual=%p",
            exp,t));

      if (expect_q.size() == 0 && run_ph.phase_done.get_objection_count(this) != 0)
        run_ph.drop_objection(this, "Scoreboard has no more expect");

    endfunction

  endclass
```

41.4 sv_main

module sv_main

Creates an instance of a producer and scoreboard, makes both native and cross-language connections using UVM Connect, then calls *run_test*.

```
module sv_main;

  import uvmc_pkg::*;

  producer prod = new("prod");
  scoreboard sb = new("sb");

  initial begin

    // expect path = normal TLM connection between producer and scoreboard
    prod.ap.connect(sb.expect_in);

    // actual path - SC-side consumer to SV-side scoreboard
    uvmc_tlm1 #(packet)::connect(prod.out,"foo");
    uvmc_tlm1 #(packet)::connect(sb.actual_in,"bar");

    run_test();
```

```
  end

endmodule
```

class packet extends uvm_object

Defines a packet class containing a field for each of the data types supported by UVMC.
class producer extends uvm_component

A simple producer that generates packet transactions and sends them out its *out* blocking-put and *ap* analysis ports.
`uvm_analysis_imp_decl(
_expect
) `uvm_analysis_imp_decl(_actual) class scoreboard extends uvm_component

A simple scoreboard that implements in-order comparison of *expect* and *actual* packets.
module sv_main

Creates an instance of a producer and scoreboard, makes both native and cross-language connections using UVM Connect, then calls *run_test*.
Converts this transaction's contents into a form transferrable outside SystemVerilog.
Converts a bit-vector representation of a transaction into this transaction object.
Copies the values of fields from another object of the same type into this object.
Compares the values of fields with those of another object of the same type, returning 1 if a match, 0 otherwise.
Implements printing of all fields in this transaction using the provided *printer* policy class.
Records all members of this transaction class for later viewing in the GUI's wave window.

42 SC Type Support

This example defines and uses a transaction that declares as members each of the data types supported for cross-language transfer by UVMC. The example consists of the following classes and functions:

42.1 Packet

First, we define a transaction class, *Packet*, declaring an instance of each supported data type for UVMC transfer. It contains the following fields:

integrals	bool, char, short, int, long long, and their unsigned counterparts.
enum	user-defined enumeration types packed by their numeric value. A compatible enumeration type must be defined on the SV side.
reals	float and double translate to shortreal and real, respectively.
strings	terminated with the NULL character. Use vector<char> for an array of bytes whose elements can include the '0' value
arrays	fixed arrays, and STL vector<T>, list<T>, and map<KEY,T> are supported as long as T and KEY are also supported. These types are similar to the SV queue, dynamic array, and associate array types, respectively. Be sure to #include the definitions and declare you're 'using' any STL types in your transaction.
time	packed as 64-bit values. Equates to the SV 'time' type.
sc data types	sc_bit, sc_logic, sc_bv<N>, sc_lv<N>, sc_int<N>, sc_uint<N>, sc_bigint<N>, and sc_biguint<N>, for any valid width, N. These translate to bit and logic vectors on the SV side.

Here is the complete definition of our SC-side Packet class. Note that it does not inherit from any base class.

```
class Packet {

  public:
    enum cmds_t { ADD, SUBTRACT, MULTIPLY, DIVIDE };

    cmds_t             enum32;  // Enumerations

    long long          int64;   // Signed integrals
    int                int32;
    short              int16;
    char               int8;
    bool               int1;

    unsigned long long uint64;  // Unsigned integrals
    unsigned int       uint32;
    unsigned short     uint16;
    unsigned char      uint8;
    bool               uint1;

    double             real64;

    sc_time            time64;  // Time

    string             str;     // Strings

    int                arr[3];  // Arrays
    vector<char>       q;
    list<short>        da;
    map<short,short>   aa;
```

```
    sub_object          obj;     // Sub-objects

    sc_bit              scbit;   // SystemC data types
    sc_logic            sclogic;
    sc_bv<17>           scbv;
    sc_lv<35>           sclv;
    sc_int<6>           scint;
    sc_uint<25>         scuint;
    sc_bigint<37>       scbigint;
    sc_biguint<62>      scbiguint;
};
```

42.2 uvmc_converter<Packet>

Next, we defined a template specialization of uvmc_converter<T> for our Packet type. This class defines do_pack and do_unpack methods that convert our Packet using the *uvmc_packer* object passed as an argument. Unlike in SV, we are defining packing and unpacking functionality in a separate class and not as methods of the Packet itself. This allows us to define custom packing algorithms, or "policies", without having to subtype the Packet class.

42.2.1 Methods

42.2.2 do_pack

Serializes the packet *t* using the provided *uvmc_packer* argument, *packer*.

All fields are "streamed" into the packer object using operator<<, which the packer defines for each of of the supported data types.

In effect, we are packing our object just as you might print your object contents to *cout*. Instead of streaming to *cout*, you are streaming to the *packer*. Thus the process of packing an object is easy. You need only ensure that the field order of packing and unpacking are the same for the converters on both sides of the language boundary.

The resulting bits are retained by the *packer* for subsequent extraction and transfer to SV.

The bits representing the serialized transaction will be transferred to SV, preloaded into a SV packer object, then unpacked via a compatible *unpack* operation in SV.

This example packs all members with one statement. You may need to split into several statements, e.g. when conditionally packing sub-objects.

When packing sub-objects, you must first pack a 4-bit value as follows

0 if the sub-object is NULL or is not to be packed
1 if the sub-object is non-NULL and packed.
If you pack a '1' as above, then pack the sub-object by calling

```
uvmc_conveter<obj type>::do_pack(packer);
    static void do_pack(const Packet &t, uvmc_packer &packer)
    {
      packer
        << t.enum32
```

```
      << t.int64  << t.int32   << t.int16  << t.int8  << t.int1
      << t.uint64 << t.uint32  << t.uint16 << t.uint8 << t.uint1
      << t.scbit  << t.sclogic << t.scbv   << t.sclv
      << t.scint  << t.scuint  << t.scbigint << t.scbiguint
      << t.time64 << t.real64
      << t.str << t.arr << t.q << t.da << t.aa;

    // pack object:
    packer << (sc_bv<4>(1));
    uvmc_converter<sub_object>::do_pack(t.obj, packer);

  }
```

42.2.3 do_unpack

Extracts a serialized version of a packet into the given Packet object. The bits representing the serialized transaction have been packed via a compatible *do_pack* operation, transferred across the language boundary if necessary, then preloaded into the *packer* object before this function is called.

All fields are "streamed" out of the packer object using operator>>, which the packer defines for each of of the supported data types.

In effect, we are unpacking into a Packet object just as you might stream Packet contents from *cin*. Instead of streaming from *cin*, you are streaming from the *packer*. Thus, the process of unpacking into an object is easy. You need only ensure that the field order of packing and unpacking are the same for the converters on both sides of the language boundary.

When unpacking sub-objects, you first unpack a 4-bit value and interpret as follows

0 set the sub-object to NULL
1 unpack the sub-object by calling *uvmc_conveter<type>::do_unpack*
This example unpacks all members with one statement. You may need to split into several statements, e.g. when conditionally unpacking sub-objects.

```
  static void do_unpack(Packet &t, uvmc_packer &packer)
  {
    packer
      >> t.enum32
      >> t.int64  >> t.int32   >> t.int16  >> t.int8  >> t.int1
      >> t.uint64 >> t.uint32  >> t.uint16 >> t.uint8 >> t.uint1
      >> t.scbit  >> t.sclogic >> t.scbv   >> t.sclv
      >> t.scint  >> t.scuint  >> t.scbigint >> t.scbiguint
      >> t.time64 >> t.real64
      >> t.str >> t.arr >> t.q >> t.da >> t.aa;

    sc_bv<4> is_null;
    packer >> is_null;
    if (is_null != 0)
      uvmc_converter<sub_object>::do_unpack(t.obj, packer);

  }

}; // uvmc_conveter<Packet>
```

42.3 uvmc_print<Packet>

A template specialization of uvmc_print<T>, this class is used by operator<<(ostream,Packet) to print the contents of a Packet.

The print method is the entry point; it calls do_print, which performs the actual streaming.

An overload of non-member function, *operator<<*, for the *ostream* is defined for Packet types. Its implementation calls this class' print method.

With these defined, any Packet object can be output as follows

```
#include <iostream>
using std::ostream;
using std::cout;

Packet t;
...
cout << t << endl;
```

42.3.1 Methods

42.3.2 do_print

Streams the data members of the Packet object *t* to the provided output stream *os*, which defaults to the standard output stream, *cout*.

All data members, including sub-objects, must have *operator<<(ostream&)* defined. This is true for all C++ and SC built-in types. The UVMC library provides *operator<<(ostream&)* for the STL *vector<T>*, *list<T>*, and *map<KEY,T>* types.

Infinite recursion through self-reference is not caught. You are responsible for making sure sub-objects, if streamed, do not refer to an object already streamed.

Although example streams all members with one statement, you may split the task into as many statements as you like.

```
template <>
class uvmc_print<Packet> {
  public:

  static void do_print(const Packet& t, ostream& os=cout) {
        os << " enum32:" << t.enum32
           << " int64:"  << t.int64
           << " int32:"  << t.int32
           << " int16:"  << t.int16
           << " int8:"   << t.int8
           << " int1:"   << t.int1
           << " int64:"  << t.uint64
           << " int32:"  << t.uint32
           << " int16:"  << t.uint16
           << " int8:"   << t.uint8
           << " int1:"   << t.uint1
           << " scbit:"  << t.scbit
```

```
              << " sclogic:"<< t.sclogic
              << " scbv:"   << t.scbv
              << " sclv:"   << t.sclv
              << " scint:"  << t.scint
              << " scuint:"    << t.scuint
              << " scbigint:"  << t.scbigint
              << " scbiguint:" << t.scbiguint
              << " time64:" << t.time64
              << " real64:" << t.real64
              << " str:"    << t.str
              << " arr:"    << t.arr
              << " q:"      << t.q
              << " da:"     << t.da
              << " aa:"     << t.aa
              << " obj:"    << t.obj;
  }
```

42.3.3 print

The entry point for printing our Packet. We output a brace, {, then call do_print, then output a closing brace, }. This structure prevents superfluous braces for transactions inheriting from base classes with fields. See the Converters examples set to see examples of converters and printers for transactions with base classes.

```
  static void print(const Packet& t, ostream& os=cout) {
    os << "'{";
    do_print(t,os);
    os << " }";
  }

};
```

42.4 operator<<(ostream,Packet)

We next defines *operator<< (ostream&)* for Packet types, enabling us to output Packet objects to cout and other output streams. Our Consumer uses *operator<<* to output Packets it receives.

Example usage

```
#include <iostream>
using std::ostream;
using std::cout;

Packet t;
...
cout << t << endl;
ostream& operator << (ostream& os, const Packet& v) {
  uvmc_print<Packet>::print(v,os);
}
```

42.5 Consumer

Defines a simple consumer of Packets. Each packet it gets is printed and sent out its *analysis_out* port.

```
using namespace sc_core;
using namespace tlm;
```

```
class Consumer : public sc_module,
                 public tlm_blocking_put_if<Packet> {

  public:
  sc_export<tlm_blocking_put_if<Packet> > in;

  tlm_analysis_port<Packet> analysis_out;

  Consumer(sc_module_name nm) : in("in"),
                                analysis_out("analysis_out")
  {
    in(*this);
  }

  virtual void put(const Packet &t) {

    cout << sc_time_stamp()
         << " SC consumer executing packet:" << endl
         << "  " << t << endl;

    wait(123,SC_NS);

    analysis_out.write(t);
  }
};
```

42.6 sc_main

Finally, in *sc_main*, we simply instantiate a Consumer of Packets, register tUVMC connections to its *in* export and *analysis_out* port, then start SC simulation. A SV-side producer will drive this consumer with transactions once SV's reaches its *run_phase*.

```
int sc_main(int argc, char* argv[])
{
  Consumer cons("consumer");

  uvmc_connect(cons.in,"foo");
  uvmc_connect(cons.analysis_out,"bar");

  sc_start();
  return 0;
}
```

First, we define a transaction class, *Packet*, declaring an instance of each supported data type for UVMC transfer.
We next defines *operator<< (ostream&)* for Packet types, enabling us to output Packet objects to cout and other output streams.
Defines a simple consumer of Packets.
Serializes the packet *t* using the provided *uvmc_packer* argument, *packer*.
Extracts a serialized version of a packet into the given Packet object.
The entry point for printing our Packet.
Streams the data members of the Packet object *t* to the provided output stream *os*, which defaults to the standard output stream, *cout*.

43 UVMC Command API

This section describes the API for accessing and controlling UVM simulation in SystemVerilog from SystemC (or C or C++). To use, the SV side must have called the *uvmc_init*, which starts a background process that receives and processes incoming commands.

The UVM Connect library provides an SystemC API for accessing SystemVeilog UVM during simulation. With this API users can:

- Wait for a UVM to reach a given simulation phase
- Raise and drop objections
- Set and get UVM configuration
- Send UVM report messages
- Set type and instance overrides in the UVM factory
- Print UVM component topology

While most commands are compatible with C, a few functions take object arguments or block on sc_events. Therefore, SystemC is currently the only practical source language for using the UVM Command API. Future releases may separate the subset of C-compatible functions into a separate shared library that would not require linking in SystemC.

The following may appear in a future release, based on demand

- Callbacks on phase and objection activity
- Receive UVM report messages, i.e. SC-side acts as report catcher or server
- Integration with SystemCÂ s sc_report_handler facility
- Print factory contents, config db contents, and other UVM information
- Separate command layer into SC, C++, and C-accessible components; i.e. not require SystemC for non-SystemC-dependent functions

The following sections provide details and examples for each command in the UVM Command API. To enable access to the UVM command layer, you must call the uvmc_cmd_init function from an initial block in a top-level module as in the following example:

43.1 SystemVerilog

```
module sv_main;

  import uvm_pkg::*;
  import uvmc_pkg::*;

  initial begin
    uvmc_cmd_init();
    run_test();
  end

endmodule
```

SystemC-side calls to the UVM Command API will block until SystemVerilog has finished elaboration and the uvmc_cmd_init function has been called. Because any call to the UVM Command layer may block, calls must be made from within thread processes.

All code provided in the UVM Command descriptions that follow are SystemC unless stated otherwise.

43.1.1 Enumeration Constants

43.1.1.1 The following enumeration constants are used in various UVM Commands

43.1.2 uvmc_phase_state

The state of a UVM phase

UVM_PHASE_DORMANT	Phase has not started yet
UVM_PHASE_STARTED	Phase has started
UVM_PHASE_EXECUTING	Phase is executing
UVM_PHASE_READY_TO_END	Phase is ready to end
UVM_PHASE_ENDED	Phase has ended
UVM_PHASE_DONE	Phase has completed

43.1.3 uvmc_report_severity

The severity of a report

UVM_INFO	Informative message. Verbosity settings affect whether they are printed.
UVM_WARNING	Warning. Not affected by verbosity settings.
UVM_ERROR	Error. Error counter incremented by default. Not affected by verbosity settings.
UVM_FATAL	Unrecoverable error. SV simulation will end immediately.

43.1.4 uvmc_report_verbosity

The verbosity level assigned to UVM_INFO reports

UVM_NONE	report will always be issued (unaffected by verbosity level)
UVM_LOW	report is issued at low verbosity setting and higher
UVM_MEDIUM	report is issued at medium verbosity and higher
UVM_HIGH	report is issued at high verbosity and higher
UVM_FULL	report is issued only when verbosity is set to full

43.1.5 uvmc_wait_op

The relational operator to apply in uvmc_wait_for_phase calls

UVM_LT	Wait until UVM is before the given phase
UVM_LTE	Wait until UVM is before or at the given phase
UVM_NE	Wait until UVM is not at the given phase
UVM_EQ	Wait until UVM is at the given phase
UVM_GT	Wait until UVM is after the given phase
UVM_GTE	Wait until UVM is at or after the given phase

43.1.6 Topology

43.1.7 uvmc_print_topology

Prints the current UVM testbench topology.

If called prior to UVM completing the build phase, the topology will be incomplete.

43.1.7.1 Arguments

context	The hierarchical path of the component from which to start printing topology. If unspecified, topology printing will begin at uvm_top. Multiple components can be specified by using glob wildcards (* and ?), e.g. “top.env.*.driver”. You can also specify a POSIX extended regular expression by enclosing the contxt in forward slashes, e.g. “/a[hp]b/”. Default: “” (uvm_top)
depth	The number of levels of hierarchy to print. If not specified, all levels of hierarchy starting from the given context are printed. Default: -1 (recurse all children)

43.1.8 Reporting

The reporting API provides the ability to issue UVM reports, set verbosity, and other reporting features.

43.1.9 uvmc_report_enabled

Returns true if a report at the specified verbosity, severity, and id would be emitted if made within the specified component contxt.

The primary purpose of this command is to determine whether a report has a chance of being emitted before incurring the high run-time overhead of formatting the string message.

A report at severity UVM_INFO is ignored if its verbosity is greater than the verbosity configured for the component in which it is issued. Reports at other severities are not affected by verbosity settings.

If the action of a report with the specified severity and id is configured as UVM_NO_ACTION within the specified component contxt.

Filtration by any registered report_catchers is not considered.

43.1.9.1 Arguments

verbosity	The uvmc_report_verbosity of the hypothetical report.
severity	The uvmc_report_severity of the hypothetical report. Default: UVM_INFO.
id	The identifier string of the hypothetical report. Must be an exact match. If not specified, then uvmc_report_enabled checks only if UVM_NO_ACTION is the configured action for the given severity at the specified context. Default: “” (unspecified)
context	The hierarchical path of the component that would issue the hypothetical report. If not specified, the context is global, i.e. uvm_top. Reports not issued by components come from uvm_top. Default: “” (unspecified)

43.1.9.2 Example

```
if (uvmc_report_enabled(UVM_HIGH, UVM_INFO, "PRINT_TRANS") {
  string detailed_msg;
  ...prepare message string here...
  uvmc_report(UVM_INFO, "PRINT_TRANS", detailed_msg, UVM_HIGH);
}
```

43.1.10 uvmc_set_report_verbosity

Sets the run-time verbosity level for all UVM_INFO-severity reports issued by the component(s) at the specified context. Any report from the component context whose verbosity exceeds this maximum will be ignored.

Reports issued by SC via uvmc_report are affected only by the verbosity level setting for the global context, i.e. context="". To have finer-grained control over SC-issued reports, register a uvm_report_catcher with uvm_top.

43.1.10.1 Arguments

level	The verbosity level. Specify UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH, or UVM_FULL. Required.
context	The hierarchical path of the component. Multiple components can be specified by using glob wildcards * and ?, e.g. "top.env.*.driver". You can also specify a POSIX extended regular expression by enclosing the contxt in forward slashes, e.g. "/a[hp]b/". Default: "" (uvm_top)
recurse	If true, sets the verbosity of all descendents of the component(s) matching context. Default:false

43.1.10.2 Examples

43.1.10.3 Set global UVM report verbosity to maximum (FULL) output

```
uvmc_set_report_verbosity(UVM_FULL);
```

Disable all filterable INFO reports for the top.env.agent1.driver, but none of its children:

```
uvmc_set_report_verbosity(UVM_NONE, "top.env.agent1.driver");
```

Set report verbosity for all components to UVM_LOW, except for the troublemaker component, which gets UVM_HIGH verbosity:

```
uvmc_set_report_verbosity(UVM_LOW, true);
uvmc_set_report_verbosity(UVM_HIGH, "top.env.troublemaker");
```

In the last example, the recursion flag is set to false, so all of troublemaker's children, if any, will remain at UVM_LOW verbosity.

43.1.11 uvmc_report

Send a report to UVM for processing, subject to possible filtering by verbosity, action, and active report catchers.

See uvmc_report_enabled to learn how a report may be filtered.

The UVM report mechanism is used instead of $display and other ad hoc approaches to ensure consistent output and to control whether a report is issued and what if any actions are taken when it is issued. All reporting methods have the same arguments, except a verbosity level is applied to UVM_INFO-severity reports.

43.1.11.1 Arguments

severity	The report severity: specify either UVM_INFO, UVM_WARNING, UVM_ERROR, or UVM_FATAL. Required argument.
id	The report id string, used for identification and targeted filtering. See context description for details on how the report's id affects filtering. Required argument.
message	The message body, pre-formatted if necessary to a single string. Required.
verbosity	The verbosity level indicating an INFO report's relative detail. Ignored for warning, error, and fatal reports. Specify UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH, or UVM_FULL. Default: UVM_MEDIUM.
context	The hierarchical path of the SC-side component issuing the report. The context string appears as the hierarchical name in the report on the SV side, but it does not play a role in report filtering in all cases. All SC-side reports are issued from the global context in UVM, i.e. uvm_top. To apply filter settings, make them from that context, e.g. uvm_top.set_report_id_action(). With the context fixed, only the report's id can be used to uniquely identify the SC report to filter. Report catchers, however, are passed the report's context and so can filter based on both SC context and id. Default: "" (unspecified)
filename	The optional filename from which the report was issued. Use __FILE__. If specified, the filename will be displayed as part of the report. Default: "" (unspecified)
line	The optional line number within filename from which the report was issued. Use __LINE__. If specified, the line number will be displayed as part of the report. Default: 0 (unspecified)

43.1.11.2 Examples

43.1.11.3 Send a global (uvm_top-sourced) info report of medium verbosity to UVM

```
uvmc_report(UVM_INFO, "SC_READY", "SystemC side is ready");
```

Issue the same report, this time with low verbosity a filename and line number.

```
uvmc_report(UVM_INFO, "SC_READY", "SystemC side is ready",
                      UVM_LOW, "", __FILE__, __LINE__);
```

UVM_LOW verbosity does not mean lower output. On the contrary, reports with UVM_LOW verbosity are printed if the run-time verbosity setting is anything but UVM_NONE. Reports issued with UVM_NONE verbosity cannot be filtered by the run-time verbosity setting.

The next example sends a WARNING and INFO report from an SC-side producer component. In SV, we disable the warning by setting the action for its effective ID to UVM_NO_ACTION. We also set the verbosity threshold for INFO messages with the effective ID to UVM_NONE. This causes the INFO report to be filtered, as the run-time verbosity for reports of that particular ID are now much lower than the reportÂ s stated verbosity level (UVM_HIGH).

```
class producer : public sc_module {
 ...
  void run_thread() {
    ...
    uvmc_report(UVM_WARNING, "TransEvent",
           "Generated error transaction.",, this.name());
    ...
    uvmc_report(UVM_INFO, "TransEvent",
           "Transaction complete.", UVM_HIGH, this.name());
    ...
  }
```

```
};
```

43.1.11.4 To filter SC-sourced reports on the SV side

```
uvm_top.set_report_id_action("TransEvent@top/prod",UVM_NO_ACTION);
uvm_top.set_report_id_verbosity("TransEvent@top/prod",UVM_NONE);
uvm_top.set_report_id_verbosity("TransDump",UVM_NONE);
```

The last statement disables all reports to the global context (uvm_top) having the ID “TransDump”. Note that it is currently not possible to set filters for reports for several contexts at once using wildcards. Also, the hierarchical separator for SC may be configurable in your simulator, and thus could affect the context provided to these commands.

43.1.12 uvmc_report_info

Equivalent to uvmc_report (UVM_INFO, ...)

43.1.13 uvmc_report_warning

Equivalent to uvmc_report (UVM_WARNING, ...)

43.1.14 uvmc_report_error

Equivalent to uvmc_report (UVM_ERROR, ...)

43.1.15 uvmc_report_fatal

Equivalent to uvmc_report (UVM_FATAL, ...)

43.1.16 Report Macros

Convenience macros to uvmc_report. See uvmc_report for details on macro arguments.

```
UVMC_INFO    (ID, message, verbosity, context)
UVMC_WARNING (ID, message, context)
UVMC_ERROR   (ID, message, context)
UVMC_FATAL   (ID, message, context)
```

Before sending the report, the macros first call uvmc_report_enabled to avoid sending the report at all if its verbosity or action would prevent it from reaching the report server. If the report is enabled, then uvmc_report is called with the filename and line number arguments provided for you.

Invocations of these macros must be terminated with semicolons, which is in keeping with the SystemC convention established for the *SC_REPORT* macros. Future releases may provide a UVMC sc_report_handler that you can use to redirect all SC_REPORTs to UVM.

43.1.16.1 Example

```
UVMC_ERROR("SC_TOP/NO_CFG","Missing required config object", name());
```

43.1.17 Phasing

An API that provides access UVM's phase state and the objection objects used to control phase progression.

43.1.18 uvmc_wait_for_phase

Wait for a UVM phase to reach a certain state.

The call must be made from a SystemC process thread that is either statically declared via SC_THREAD or dynamically started via sc_spawn. If the latter, you must include the following define on the sccom command line: -DSC_INCLUDE_DYNAMIC_PROCESSES.

43.1.18.1 Arguments

phase — The name of the phase to wait on. The built-in phase names are, in order of execution: build, connect, end_of_elaboration, start_of_simulation, run, extract, check, report. The fine-grained run-time phases, which run in parallel with the run phase, are, in order: pre_reset, reset, post_reset, pre_configure, configure, post_configure, pre_main, main, post_main, pre_shutdown, shutdown, and post_shutdown.

state — The state to wait on. A phase may transition through *UVM_PHASE_JUMPING* instead of UVM_PHASE_READY_TO_END if its execution had been preempted by a phase jump operation. Phases execute in the following state order:

```
UVM_PHASE_STARTED
UVM_PHASE_EXECUTING
UVM_PHASE_READY_TO_END | UVM_PHASE_JUMPING
UVM_PHASE_ENDED
UVM_PHASE_DONE
```

condition — The state condition to wait for. When state is *UVM_PHASE_JUMPING*, *condition* must be UVM_EQ. Default is *UVM_EQ*. Valid values are:

```
UVM_LT - Phase is before the given state.
UVM_LTE - Phase is before or at the given state.
UVM_EQ - Phase is at the given state.
UVM_GTE - Phase is at or after the given state.
UVM_GT - Phase is after the given state.
UVM_NE - Phase is not at the given state.
```

43.1.18.2 Examples

The following example shows how to spawn threads that correspond to UVM's phases. Here, the *run_phase* method executes during the UVM's run phase. As any UVM component executing the run phase would do, the SC component's *run_phase* process prevents the UVM (SV-side) run phase from ending until it is finished by calling uvmc_drop_objection.

```
SC_MODULE(top)
{
  sc_process_handle run_proc;

  SC_CTOR(top) {
    run_proc = sc_spawn(sc_bind(&run_phase,this),"run_phase");
  };

  void async_reset() {
    if (run_proc != null && run_proc.valid())
      run_proc.reset(SC_INCLUDE_DESCENDANTS);
```

```
  }

  void run_phase() {
   uvmc_wait_for_phase("run",UVM_PHASE_STARTED,UVM_EQ);
   uvmc_raise_objection("run","","SC run_phase executing");
     ...
   uvmc_drop_objection("run","","SC run_phase finished");
  }
};
```

If *async_reset* is called, the run_proc process and all descendants are killed, then run_proc is restarted. The run_proc calls run_phase again, which first waits for the UVM to reach the run_phase before resuming its work.

43.1.19 uvmc_raise_objection

43.1.20 uvmc_drop_objection

Raise or drop an objection to ending the specified phase on behalf of the component at a given context. Raises can be made before the actual phase is executing. However, drops that move the objection count to zero must be avoided until the phase is actually executing, else the all-dropped condition will occur prematurely. Typical usage includes calling uvmc_wait_for_phase.

43.1.21 Factory

This API provides access to UVM's object and component factory.

43.1.22 uvmc_print_factory

Prints the state of the UVM factory, including registered types, instance overrides, and type overrides.

43.1.22.1 Arguments

all_types — When all_types is 0, only type and instance overrides are displayed. When all_types is 1 (default), all registered user-defined types are printed as well, provided they have type names associated with them. (Parameterized types usually do not.) When all_types is 2, all UVM types (prefixed with uvm_) are included in the list of registered types.

43.1.22.2 Examples

Print all type and instance overrides in the factory.

```
uvmc_print_factory(0);
```

Print all type and instance overrides, plus all registered user-defined types.

```
uvmc_print_factory(1);
```

Print all type and instance overrides, plus all registered types, including UVM types.

```
uvmc_print_factory(2);
```

43.1.23 uvmc_set_factory_type_override

43.1.24 uvmc_set_factory_inst_override

Set a type or instance override. Instance overrides take precedence over type overrides. All specified types must have been registered with the factory.

43.1.24.1 Arguments

requested_type	The name of the requested type.
override_type	The name of the override type. Must be an extension of the requested_type.
context	The hierarchical path of the component. Multiple components can be specified by using glob wildcards (* and ?), e.g. “top.env.*.driver”. You can also specify a POSIX extended regular expression by enclosing the context string in forward slashes, e.g. “/a[hp]b/”.
replace	If true, replace any existing type override. Default: true

43.1.24.2 Examples

The following sets an instance override in the UVM factory. Any component whose inst path matches the glob expression “e.*” that requests an object of type scoreboard_base (i.e. scoreboard_base:type_id::create) will instead get an object of type scoreboard.

```
uvmc_set_factory_inst_override("scoreboard_base","scoreboard","e.*");
```

The following sets a type override in the UVM factory. Any component whose hierarchical path matches the glob expression “e.*” that requests an object of type producer_base (i.e. producer_base:type_id::create) will instead get an object of type producer.

```
uvmc_set_factory_type_override("producer_base","producer");
```

The following sets an override chain. Given any request for an atype, a ctype object is returned, except for the component with hierarchical name “e.prod”, which will get a dtype.

```
uvmc_set_factory_type_override("atype","btype");
uvmc_set_factory_type_override("btype","ctype");
uvmc_set_factory_inst_override("ctype","dtype","e.prod");
```

43.1.25 uvmc_debug_factory_create

Display detailed information about the object type the UVM factory would create given a requested type and context, listing each override that was applied to arrive at the result.

43.1.25.1 Arguments

requested	The requested type name for a hypothetical call to create.
context	The hierarchical path of the object to be created, which is a concatenation of the parent’s hierarchical name with the name of the object being created. Wildcards or regular expressions are not allowed. The context must exactly match an existing component’s hierarchical name, or can be the empty string to specify global context.

43.1.25.2 Example

The following example answers the question: If the component at hierarchical path *env.agent1.scoreboard* requested an object of type scoreboard_base, what are all the applicable overrides, and which of those were applied to arrive at the result?

```
uvmc_debug_factory_create("scoreboard_base","env.agent1.scoreboard");
```

43.1.26 uvmc_find_factory_override

Returns the type name of the type that would be created by the factory given the requested type and context.

43.1.26.1 Arguments

requested	The requested type name for a hypothetical call to create.
context	The hierarchical path of the component that would make the request. Wildcards or regular expressions are not allowed. The context must exactly match an existing component's hierarchical name, or can be the empty string to specify global context.

43.1.26.2 Examples

The following examples assume all types, A through D, have been registered with the UVM factory. Given the following overrides:

```
uvmc_set_type_override("B","C");
uvmc_set_type_override("A","B");
uvmc_set_inst_override("D", "C", "top.env.agent1.*");
```

43.1.26.3 The following will display "C"

```
$display(uvmc_find_factory_override("A"));
```

43.1.26.4 The following will display "D"

```
$display(uvmc_find_factory_override("A", "top.env.agent1.driver"));
```

The returned string can be used in subsequent calls to uvmc_set_factory_type_override and uvmc_set_factory_inst_override.

43.1.27 set_config

Creates or updates a configuration setting for a field at a specified hierarchical context.

These functions establish configuration settings, storing them as resource entries in the resource database for later lookup by get_config calls. They do not directly affect the field values being targeted. As the component hierarchy is being constructed during UVM's build phase, components spring into existence and establish their context. Once its context is known, each component can call get_config to retrieve any configuration settings that apply to it.

The context is specified as the concatenation of two arguments, context and inst_name, which are separated by a "." if both context and inst_name are not empty. Both context and inst_name may be glob style or regular expression style expressions.

The name of the configuration parameter is specified by the *field_name* argument.

The semantic of the set methods is different when UVM is in the build phase versus any other phase. If a set call is made during the build phase, the context determines precedence in the database. A set call from a higher level in the hierarchy (e.g. mytop.test1) has precedence over a set call to the same field from a lower level (e.g. mytop.test1.env.agent). Set calls made at the same level of hierarchy have equal precedence, so each set call overwrites the field value from a previous set call.

After the build phase, all set calls have the same precedence regardless of their hierarchical context. Each set call overwrites the value of the previous call.

43.1.27.1 Arguments

type_name	For uvmc_set_config_object only. Specifies the type name of the equivalent object to set in SV. UVM Connect will utilize the factory to allocate an object of this type and unpack the serialized value into it. Parameterized classes are not supported.
context	The hierarchical path of the component, or the empty string, which specifies uvm_top. Multiple components can be specified by using glob wildcards (* and ?), e.g. “top.env.*.driver”. You can also specify a POSIX extended regular expression by enclosing the contxt in forward slashes, e.g. “/a[hp]b/”. Default: “” (uvm_top)
inst_name	The instance path of the object being configured, relative to the specified context(s). Can contain wildcards or be a regular expression.
field_name	The name of the configuration parameter. Typically this name is the same as or similar to the variable used to hold the configured value in the target context(s).
value	The value of the configuration parameter. Integral values currently cannot exceed 64 bits. Object values must have a uvmc_convert<object_type> specialization defined for it. Use of the converter convenience macros is acceptable for meeting this requirement.

43.1.27.2 Examples

The following example sets the configuration object field at path “e.prod.trans” to the tr instance, which is type uvm_tlm_generic_payload.

```
uvmc_set_config_object("uvm_tlm_generic_payload","e.prod","","trans", tr);
```

The next example sets the string property at hierarchical path “e.prod.message” to “Hello from SystemC!”.

```
uvmc_set_config_string ("e.prod", "", "message", "Hello from SystemC!");
```

The next example sets the integral property at hierarchical path “e.prod.start_addr” to hex 0x1234.

```
uvmc_set_config_int ("e.prod", "", "start_addr", 0x1234);
```

43.1.28 uvmc_set_config_int

Set an integral configuration value

43.1.29 uvmc_set_config_string

Set a string configuration value

43.1.30 uvmc_set_config_object

Set an object configuration value using a custom converter

43.1.31 uvmc_set_config_object

Set an object configuration value using the default converter

43.1.32 get_config

Gets a configuration field *value* at a specified hierarchical *context*. Returns true if successful, false if a configuration setting could not be found at the given *context*. If false, the *value* reference is unmodified.

The *context* specifies the starting point for a search for a configuration setting for the field made at that level of hierarchy or higher. The *inst_name* is an explicit instance name relative to context and may be an empty string if *context* is the full context that the configuration setting applies to.

The *context* and *inst_name* strings must be simple strings--no wildcards or regular expressions.

See the section on set_config for the semantics that apply when setting configuration.

43.1.32.1 Arguments

type_name	For uvmc_get_config_object only. Specifies the type name of the equivalent object to retrieve in SV. UVM Connect will check that the object retrieved from the configuration database matches this type name. If a match, the object is serialized (packed) and returned across the language boundary. Once on this side, the object data is unpacked into the object passed by reference via the value argument. Parameterized classes are not supported.
context	The hierarchical path of the component on whose behalf the specified configuration is being retrieved. Wildcards or regular expressions are not allowed. The context must exactly match an existing component's hierarchical name, or be the empty string, which specifies uvm_top.
inst_name	The instance path of the object being configured, relative to the specified context(s).
field_name	The name of the configuration parameter. Typically this name is the same as or similar to the variable used to hold the configured value in the target context(s).
value	The value of the configuration parameter. Integral values currently cannot exceed 64 bits. Object values must have a uvmc_convert<object_type> specialization defined for it. Use of the converter convenience macros is acceptable for meeting this requirement. The equivalent class in SV must be based on uvm_object and registered with the UVM factory, i.e. contain a `uvm_object_utils macro invocation.

43.1.32.2 Examples

The following example retrieves the uvm_tlm_generic_payload configuration property at hierarchical path "e.prod.trans" into tr2.

```
uvmc_get_config_object("uvm_tlm_generic_payload","e","prod","trans", tr2);
```

The context specification is split between the context and inst_name arguments. Unlike setting configuration, there is no semantic difference between the context and inst_name properties. When getting configuration, the full context is always the concatenation of the context, ".", and inst_name. The transaction tr2 will effectively become a copy of the object used to set the configuration property.

The next example retrieves the string property at hierarchical path “e.prod.message” into local variable str.

```
uvmc_get_config_string ("e", "prod", "message", str);
```

The following example retrieves the integral property at hierarchical path “e.prod.start_addr” into the local variable, saddr.

```
uvmc_get_config_int ("e.prod", "", "start_addr", saddr);
```

43.1.33 uvmc_get_config_int

Set an integral configuration value.

43.1.34 uvmc_get_config_string

Set a string configuration value.

43.1.35 uvmc_get_config_object

Set an object configuration value using a custom converter

43.1.36 uvmc_get_config_object

Set an object configuration value using the default converter

Wait for a UVM phase to reach a certain state.
Send a report to UVM for processing, subject to possible filtering by verbosity, action, and active report catchers.
The severity of a report
Returns true if a report at the specified verbosity, severity, and id would be emitted if made within the specified component contxt.
Raise or drop an objection to ending the specified phase on behalf of the component at a given context.
Set a type or instance override.
Creates or updates a configuration setting for a field at a specified hierarchical context.
Set an object configuration value using a custom converter

44 UVM Command Examples

The *examples/commands* directory contains several examples of using the UVMC Command API from SystemC to query, configure, and control UVM simulation in SystemVerilog.

See <Getting Started> for setup requirements before running the examples. Specifically, you will need to have precompiled the UVM and UVMC libraries and set environment variables pointing to them.

44.1 Use *make help* to view the menu of available examples

```
> make help
```

You'll get a menu similar to the following

```
 ----------------------------------------------------------------
|                   UVMC EXAMPLES - UVM COMMANDS                 |
 ----------------------------------------------------------------
|                                                                |
| Usage:                                                         |
|                                                                |
|   make [UVM_HOME=path] [UVMC_HOME=path] [TRACE=1] <example>    |
|                                                                |
| where <example> is one of                                      |
|                                                                |
|   config    : shows usage of the UVMC set/get config API       |
|                                                                |
|   reporting : shows how to issue and filter UVM standard       |
|               reports                                          |
|                                                                |
|   factory   : shows how to set type and instance overrides and |
|               dump factory state and perform factory debug     |
|                                                                |
|   topology  : illusrates how (and when) to dump UVM topology   |
|                                                                |
|   phasing   : show how SC can wait for any UVM phase state     |
|               and raise/drop objections to control their       |
|               progression                                      |
|                                                                |
| UVM_HOME and UVMC_HOME specify the location of the source      |
| headers and macro definitions needed by the examples. You must |
| specify their locations via UVM_HOME and UVMC_HOME environment |
| variables or make command line options. Command line options   |
| override any envrionment variable settings.                    |
|                                                                |
| The UVM and UVMC libraries must be compiled prior to running   |
| any example. If the libraries are not at their default location |
| (UVMC_HOME/lib) then you must specify their location via the   |
| UVM_LIB and/or UVMC_LIB environment variables or make command  |
| line options. Make command line options take precedence.       |
|                                                                |
| If TRACE=1 is used, UVM command tracing is enabled (try it!)   |
|                                                                |
| Other options:                                                 |
|                                                                |
|   all   : Run all examples                                     |
|   clean : Remove simulation files and directories              |
|   help  : Print this help information                          |
|                                                                |
```

```
-------------------------------------------------------------
```

To run just the ‘phasing’ example

```
> make phasing
```

This runs the ‘phasing’ example with the UVM source location defined by the *UVM_HOME* environment variable and the UVM and UVMC compiled libraries at their default location, *../../lib/uvmc_lib*.

To run all *UVM Command* examples

```
prompt> make all
```

The *clean* target deletes all the simulation files produced from previous runs.

```
prompt> make clean
```

You can combine targets in one command line

```
prompt> make clean all
```

The following runs the ‘phasing’ example using the UVM library at the given path, which overrides the UVM_HOME environment variable.

```
> make UVM_HOME=<path> phasing
```

Assuming your environment is properly set up, choose an example to run from the menu, say

```
> make phasing
```

This compiles and runs the example that demonstrates SC waiting for and controlling phase progression in SV UVM.

```
//------------------------------------------------------------//
//   Copyright 2009-2015 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwide                            //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//
```

45 UVMC Command API Example - Configuration

This example demonstrates usage the set_config and get_config portion of the UVMC Command API.

For setting an object, a converter must be defined for the object being transferred. See Converters for how to do that. This requirement is most easily met using one of the UVMC_UTILS or UVMC_CONVERT macros, which generate the converter specialization class.

uvmc_set_config - The 1st arg is the full path to the component(s) on whose behalf you are setting the configuration. If empty (“”), *uvm_top* is used. The 2nd argument is the path to the component target(s) whose configuration value you want to set, relative to the path given by the 1st argument. The 3rd argument is the name of the configuration field to set (as documented by the target component(s)). The full path to the target field(s) is thus

```
{arg1,".",arg2,".",arg3}
```

uvmc_get_config- The 1st arg is the full path to the component target whose configuration value you want to get. The 2nd argument is empty. (The 2nd arg is there to match the prototype of the uvm_config_db::get method in UVM.) The 3rd argument is the name of the config field to get. If a configuration was set at the path {arg1,”.”,arg3}, then the 4th argument contains the value of that config field and ‘true’ is returned, otherwise false.

45.1 prod_cfg

The *prod_cfg* class is the configuration object we’ll be sending to the producer component on the SV side. It dictates the address range, data array length ranges, and the maximum number of transactions to produce. We use the UVMC_UTILS macro option to quickly define a UVMC converter for it.

```
// (begin inline source)
#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;


class prod_cfg {
  public:
  int min_addr;
  int max_addr;
  int min_data_len;
  int max_data_len;
  int max_trans;
};

UVMC_UTILS_5(prod_cfg,min_addr,max_addr,
             min_data_len,max_data_len,max_trans)
```

45.2 top

Our top-level SC module does the following

- Creates an instance of a generic consumer. The consumer merely prints the transactions it receives side and sends them out its analysis port.

- Spawn a thread function, *show_uvm_config*
- Register the consumer's ports for UVMC connection.

The *show_uvm_config* thread will perform a set and get config operation on an integral, string, and object type. The object we use, prod_cfg, configures in one go the SV-side producer with max transaction count and constraints on address and data array length.

Note the use of the UVM_INFO and UVM_ERROR macros, which issue reports to UVM for filtering, formatting, and display.

```
#include <string>
#include <iostream>
#include "systemc.h"
#include "tlm.h"
#include "uvmc.h"

using namespace std;
using namespace sc_core;
using namespace tlm;
using namespace uvmc;

#include "consumer.cpp"


SC_MODULE(top)
{
  consumer cons;

  SC_CTOR(top) : cons("consumer") {
    SC_THREAD(show_uvm_config);
    uvmc_connect(cons.in,"foo");
    uvmc_connect(cons.analysis_out,"bar");
  }

  void show_uvm_config();

};
```

45.2.1 Methods

45.2.2 show_uvm_config

The *show_uvm_config* thread will perform a set and get config operation on an integral, string, and object type. The object we use, prod_cfg, configures in one go the SV-side producer with max transaction count and constraints on address and data array length.

Note the use of the UVM_INFO and UVM_ERROR macros, which issue reports to UVM for filtering, formatting, and display.

```
void top::show_uvm_config()
{
  string s = "Greetings from SystemC";
  uint64 i = 2;
  prod_cfg cfg;

  cfg.min_addr=0x100;
```

```
  cfg.max_addr=0x10f;
  cfg.min_data_len=1;
  cfg.max_data_len=8;
  cfg.max_trans=2;

  wait(SC_ZERO_TIME);


  UVMC_INFO("TOP/SET_CFG",
    "Calling set_config_* to SV-side instance 'e.prod'",
    UVM_MEDIUM,"");

  uvmc_set_config_int    ("e.prod", "", "some_int", i);

  uvmc_set_config_string ("", "e.prod", "some_string", s.c_str());

  uvmc_set_config_object ("prod_cfg", "e", "prod", "config", cfg);


  // Wait until the build phase. The SV side will have used get_config
  // to retreive our settings
  uvmc_wait_for_phase("build", UVM_PHASE_ENDED);

  i=0;
  s="";
  cfg.min_addr=0;
  cfg.max_addr=0;

  UVMC_INFO("TOP/GET_CFG", \
    "Calling get_config_* from SV-side context 'e.prod'", \
    UVM_MEDIUM,"");


  // Get and check our int, string, and object configuration
  if (uvmc_get_config_int ("e.prod", "", "some_int", i))
    cout << "get_config_int : some_int=" << hex << i << endl;
  else
    UVMC_ERROR("GET_CFG_INT_FAIL", "get_config_int failed",name());

  if (uvmc_get_config_string ("e.prod", "", "some_string", s))
    cout << "get_config_string: some_string=" << s << endl;
  else
    UVMC_ERROR("GET_CFG_STR_FAIL", "get_config_string failed",name());

  if (uvmc_get_config_object ("prod_cfg", "e.prod", "", "config", cfg))
    cout << "get_config_object: config = " << cfg << endl;
  else
    UVMC_ERROR("GET_CFG_OBJ_FAIL", "get_config_object failed",name());
}
```

45.3 SC_MAIN

Creates an instance of our top module then calls *sc_start* to start SC simulation.

```
int sc_main(int argc, char* argv[])
{
  top t("top");
  sc_start();
  return 0;
}
```

Creates or updates a configuration setting for a field at a specified hierarchical context.
Gets a configuration field *value* at a specified hierarchical *context*.
Our top-level SC module does the following
This chapter shows how to write converters for your transactions.
Generate both a converter specialization and output stream *operator*<< for the given transaction *TYPE*.
Generate a converter specialization of *uvmc_convert*<*T*> for the given transaction *TYPE*.
The *prod_cfg* class is the configuration object we'll be sending to the producer component on the SV side.

46 UVMC Command API Example - Factory

This example demonstrates making UVM factory queries and setting type and instance overrides. The UVMC factory API mirrors the methods provided in UVM:

uvmc_print_factory	print the contents of the UVM factory
uvmc_find_factory_override	get the name of the type the factory would create given a requested type and context.
uvmc_debug_factory_create	print detailed information on how the factory arrives the type it returns given a requested type and a context.
uvmc_set_factory_type_override	set a type-wide override whereby all requests for a type will instead produce the override type for all instances of the requested type.
uvmc_set_factory_inst_override	set a instance override whereby a request for a type will instead produce the override type for all instances matching the given path. Glob and regular expressions are allowed.

See the Factory command descriptions for more details.

46.1 top

Our top-level SC module does the following

- Creates an instance of a generic consumer. The consumer merely prints the transactions it receives side and sends them out its analysis port.
- Spawn a thread function, *show_uvm_factory*
- Register the consumer's ports for UVMC connection.

```
#include "systemc.h"
#include "tlm.h"
#include <string>
#include <iostream>

using namespace std;
using namespace sc_core;
using namespace tlm;

#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;

#include "consumer.cpp"


SC_MODULE(top)
{
  consumer cons;

  SC_CTOR(top) : cons("consumer") {
    SC_THREAD(show_uvm_factory);
    uvmc_connect(cons.in,"foo");
    uvmc_connect(cons.analysis_out,"bar");
  }

  void show_uvm_factory();

};
```

46.1.1 show_uvm_factory

The *show_uvm_factory* thread will show usage of the UVMC factory commands. In this example, we configure type and instance overrides to make the UVM factory create extensions to the default producer and scoreboard components. We call on uvmc_print_factory and uvmc_debug_factory_create to confirm our settings, and we call ~uvmc_find_factory_override> to confirm that our override has taken effect.

Note the use of the UVM_INFO and UVM_ERROR macros, which issue reports to UVM for filtering, formatting, and display.

```
void top::show_uvm_factory()
{
  string override;

  // print the factory before we do anything
  uvmc_print_factory();

  // what type would the factory give if we asked for a producer?
  override = uvmc_find_factory_override("producer","e.prod");

  UVMC_INFO("SHOW_FACTORY",
    (string("Factory override for type 'producer' ") +
     + "in context 'e.prod' is " + override).c_str(),
     UVM_NONE,"");

  // show how factory chooses what type it creates
  uvmc_debug_factory_create("producer","e.prod");

  // set a type and instance override
  uvmc_set_factory_type_override("producer","producer_ext","e.*");
  uvmc_set_factory_inst_override("scoreboard","scoreboard_ext","e.*");

  // print the factory after setting overrides
  uvmc_print_factory();

  uvmc_debug_factory_create("producer","e.prod");
  uvmc_debug_factory_create("scoreboard","e.sb");

  // NOW what type would the factory give if we asked for a producer?
  override = uvmc_find_factory_override("producer","e.prod");

  UVMC_INFO("SHOW_FACTORY",
    (string("Factory override for type 'producer' ") +
     + "with context 'e.prod' is " + override).c_str(),
     UVM_NONE,"");

  // What type would the factory give if we asked for a scoreboard?
  override = uvmc_find_factory_override("scoreboard","e.*");

  UVMC_INFO("SHOW_FACTORY",
    (string("Factory override for type 'scoreboard' ") +
     + "given a context 'e.*' is " + override).c_str(),
     UVM_NONE,"");
}
```

46.1.2 sc_main

Creates an instance of our top module then calls *sc_start* to start SC simulation.

```
int sc_main(int argc, char* argv[])
{
  top t("top");
  sc_start();
  return 0;
}
```

This API provides access to UVM's object and component factory.
Our top-level SC module does the following
Prints the state of the UVM factory, including registered types, instance overrides, and type overrides.
Display detailed information about the object type the UVM factory would create given a requested type and context, listing each override that was applied to arrive at the result.

47 UVMC Command API Example - Phase Control

This code provides an example of waiting for each UVM phase to reach a specified state and then, if the phase is a task phase, controlling its progression by raising and dropping the objection that governs it.

uvmc_wait_for_phase	block until UVM has reached a certain phase. You may also wait for certain phase state (e.g. started, ended, etc.)
uvmc_raise_objection	prevent a UVM phase from ending
uvmc_drop_objection	remove your objection to ending a UVM phase

See the Phasing command descriptions for more details.

47.1 top

Our top-level SC module does the following

- Creates an instance of a generic consumer. The consumer merely prints the transactions it receives side and sends them out its analysis port.
- Spawn a thread function, *show_uvm_phasing*
- Register the consumer's ports for UVMC connection.

```
#include "systemc.h"
#include "tlm.h"
#include <string>
#include <iostream>

using namespace std;
using namespace sc_core;
using namespace tlm;

#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;

#include "consumer.cpp"

SC_MODULE(top)
{
  consumer cons;

  SC_CTOR(top) : cons("consumer") {
    SC_THREAD(show_uvm_phasing);
    uvmc_connect(cons.in,"foo");
    uvmc_connect(cons.analysis_out,"bar");
  }

  void show_uvm_phasing();

  private:
  void spawn_phase_control_proc(const char* phase, bool is_task_phase);
  void wait_phase_started(const char* ph_name, bool is_task_phase);
};
```

47.1.1 show_uvm_phasing

The *show_uvm_phasing* thread spawns as many sub-processes as there are predefined phases in UVM, where each thread will wait for its associated phase. If the phase is a task-based phase, each thread will raise an objection, delay, then drop the objection. This shows how SC can prevent a UVM phase from ending. The UVMC Connection Example - SC to SV, SC side shows one practical use for phase control.

```
void top::show_uvm_phasing()
{
  wait(SC_ZERO_TIME);

  // common phases
  spawn_phase_control_proc("build",0);
  spawn_phase_control_proc("connect",0);
  spawn_phase_control_proc("end_of_elaboration",0);
  spawn_phase_control_proc("start_of_simulation",0);
  spawn_phase_control_proc("run",1);
  spawn_phase_control_proc("extract",0);
  spawn_phase_control_proc("check",0);
  spawn_phase_control_proc("report",0);

  // uvm run-time phases
  spawn_phase_control_proc("pre_reset",1);
  spawn_phase_control_proc("reset",1);
  spawn_phase_control_proc("post_reset",1);
  spawn_phase_control_proc("pre_configure",1);
  spawn_phase_control_proc("configure",1);
  spawn_phase_control_proc("post_configure",1);
  spawn_phase_control_proc("pre_main",1);
  spawn_phase_control_proc("main",1);
  spawn_phase_control_proc("post_main",1);
  spawn_phase_control_proc("pre_shutdown",1);
  spawn_phase_control_proc("shutdown",1);
  spawn_phase_control_proc("post_shutdown",1);
}
```

47.1.2 spawn_phase_control_proc

A convenience function for spawning a dynamic SC thread.

```
void top::spawn_phase_control_proc(const char* phase, bool is_task_phase)
{
  sc_spawn(sc_bind(&top::wait_phase_started,this,phase,
         is_task_phase), (string("wait_for_") + phase).c_str());
}
```

47.1.3 wait_phase_started

This function is spawned as a dynamic SC thread for each predefined phase in UVM. Each thread waits for the UVM phase given by *ph_name* to reach the started state. If the phase is a task phase, it will raise an objection, wait 10ns, then drop the objection. The reports that get emitted will show that UVM phases are being controlled by these threads.

```
void top::wait_phase_started(const char* ph_name, bool is_task_phase)
{
```

```
  UVMC_INFO("SC_TOP/WAITING",(string("Waiting for phase ") +
            ph_name + " to start...").c_str(),UVM_LOW,"");

  uvmc_wait_for_phase(ph_name, UVM_PHASE_STARTED);

  UVMC_INFO("SC_TOP/PH_STARTED", (string(name()) + ": Phase " +
            ph_name + " has started").c_str(), UVM_MEDIUM,"");

  // if a task, raise and drop objection

  if (is_task_phase)
  {
    UVMC_INFO("SC_TOP/RAISE_OBJ",
              (string(name()) + " raising objection in phase "
              + ph_name).c_str(), UVM_MEDIUM,"");

    // if we're the 'run' phase, wait until post_shutdown phase
    if (!strcmp(ph_name ,"run"))
      uvmc_wait_for_phase("post_shutdown",UVM_PHASE_STARTED);

    uvmc_raise_objection(ph_name, name(), "SC waiting 10ns");

    // wait some delay to prove we are in control...
    wait(sc_time(10,SC_NS));

    UVMC_INFO("SC_TOP/DROP_OBJ",
              (string(name()) + " dropping objection in phase "
              + ph_name).c_str(), UVM_MEDIUM,"");

    uvmc_drop_objection(ph_name, name(), "10ns has passed");
  }
}
```

47.1.4 sc_main

Creates an instance of our top module then calls *sc_start* to start SC simulation.

```
int sc_main(int argc, char* argv[])
{
  top t("top");
  sc_start();
  return 0;
}
```

An API that provides access UVM's phase state and the objection objects used to control phase progression.
Our top-level SC module does the following
This example shows a SC producer driving a SV consumer via a TLM connection made with UVMC, including how to derive a SC producer subtype that can control UVM phasing using the UVMC Command API.

48 UVMC Command API Example - Print Topology

This example shows how to print the UVM testbench topology from SC.

First, we wait for the UVM's build phase to start, then print the UVM topology. At this point only the top-level component exists.

Then, we wait for the build phase to end, then print the topology once more. This time, all the UVM components exist and you see

48.1 top

Our top-level SC module does the following

- Creates an instance of a generic consumer. The consumer merely prints the transactions it receives side and sends them out its analysis port.
- Spawn a thread function, *show_uvm_print_topology*
- Register the consumer's ports for UVMC connection.

```
#include "systemc.h"
#include "tlm.h"
#include <string>
#include <iostream>

using namespace std;
using namespace sc_core;
using namespace tlm;

#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;

#include "consumer.cpp"


SC_MODULE(top)
{
  consumer cons;

  SC_CTOR(top) : cons("consumer") {
    SC_THREAD(show_uvm_print_topology);
    uvmc_connect(cons.in,"foo");
    uvmc_connect(cons.analysis_out,"bar");
  }

  void show_uvm_print_topology();

};
```

48.1.1 show_uvm_print_topology

The *show_uvm_print_topology* waits for UVM to start its *build_phase*, then prints UVM topology. At this point only the top-level component exists.

It then waits for the *build_phase* to finish. This time, printing the UVM topology shows our expected testbench topology.

48.1.2 sc_main

Creates an instance of our top module then calls *sc_start* to start SC simulation.

```
int sc_main(int argc, char* argv[])
{
  top t("top");
  sc_start();
  return 0;
}
```

../../../uvmc/examples/commands/ex_print_topology.cpp

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

// (begin inline source)
#include "systemc.h"
#include "tlm.h"
#include
```

Our top-level SC module does the following

49 UVMC Command API Example - Reporting

This code provides an example of issuing UVM reports and setting report verbosity from SC.

uvmc_report	Send a report to UVM for processing
uvmc_set_report_verbosity	Set the verbosity level of messages coming from the given context(s).
uvmc_report_enabled	Return true if a report of the given severity, verbosity, and ID would be issued. Return 0 if it would be filtered.
UVMC_INFO	Send an info report to UVM for processing if *uvmc_report_enabled* returns true. File and line number are provided in the call to *uvmc_report*.
UVMC_WARNING	Send a warning report to UVM for processing if *uvmc_report_enabled* returns true. File and line number are provided in the call to *uvmc_report*.
UVMC_ERROR	Send an error report to UVM for processing if *uvmc_report_enabled* returns true. File and line number are provided in the call to *uvmc_report*.
UVMC_FATAL	Send a fatal report to UVM for processing if *uvmc_report_enabled* returns true. File and line number are provided in the call to *uvmc_report*.

The UVM reporting API provides a means of issuing reports from SC that are filtered and formatted by the standard UVM reporting mechanism in SV. Reports from SC are subject to the same filtering and report catching semantics as native UVM reports.

Reports issued from SC via uvmc_report use the global *uvm_top* as context, but the provided *context* is displayed to screen. To have the actual SC *context* participate in finer-grained report filtering, use the report catching mechanism.

When setting report verbosity from SC, the *context* argument refers to SV context. Use a context of "" when setting verbosity level for reports issued from SC via uvmc_report.

49.1 top

Our top-level SC module does the following

- Creates an instance of a generic consumer. The consumer merely prints the transactions it receives side and sends them out its analysis port.
- Spawn a thread function, *show_uvm_reporting*
- Register the consumer's ports for UVMC connection.

```
#include "systemc.h"
#include "tlm.h"
#include <string>
#include <iostream>

using namespace std;
using namespace sc_core;
using namespace tlm;

#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;


#include "consumer.cpp"

SC_MODULE(top)
```

```
{
  consumer cons;

  SC_CTOR(top) : cons("consumer") {
    SC_THREAD(show_uvm_reporting);
    uvmc_connect(cons.in,"foo");
    uvmc_connect(cons.analysis_out,"bar");
  }

  void show_uvm_reporting();

  private:
  void issue_reports();
};
```

49.1.1 show_uvm_factory

This function sets UVM's report verbosity to various levels and issues several reports to demonstrate the effect.

49.1.2 sc_main

Creates an instance of our top module then calls *sc_start* to start SC simulation.

```
int sc_main(int argc, char* argv[])
{
  top t("top");
  sc_start();
  return 0;
}
```

../../../uvmc/examples/commands/ex_reporting.cpp

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

// (begin inline source)
#include "systemc.h"
#include "tlm.h"
#include
```

Send a report to UVM for processing, subject to possible filtering by verbosity, action, and active report catchers.
Our top-level SC module does the following

50 UVMC Command API Examples - Common SV Code

This code provides an example of waiting for each UVM phase to reach a specified state and then, if the phase is a task phase, controlling its progression by raising and dropping the objection that governs it.

uvmc_wait_for_phase	block until UVM has reached a certain phase. You may also wait for certain phase state (e.g. started, ended, etc.)
uvmc_raise_objection	prevent a UVM phase from ending
uvmc_drop_objection	remove your objection to ending a UVM phase

See the Phasing command descriptions for more details.

50.1 prod_cfg

The *prod_cfg* class is the configuration object used by our producer below. The UVMC Command API Example - Configuration demonstrates how to set this configuration from the SC side.

The configuration object specifies various constraints on randomization of the generated transactions: the number of transactions, the address range, and limits on the size of the data array.

SV-side conversion is implemented inside the *do_pack* and *do_unpack* methods in the configuration object.

```
class prod_cfg extends uvm_object;

  `uvm_object_utils(prod_cfg);

  function new(string name="producer_config_inst");
    super.new(name);
  endfunction

  int min_addr = 'h00;
  int max_addr = 'hff;
  int min_data_len = 10;
  int max_data_len = 80;
  int max_trans = 5;

  virtual function void do_pack(uvm_packer packer);
    `uvm_pack_int(min_addr)
    `uvm_pack_int(max_addr)
    `uvm_pack_int(min_data_len)
    `uvm_pack_int(max_data_len)
    `uvm_pack_int(max_trans)
  endfunction

  virtual function void do_unpack(uvm_packer packer);
    `uvm_unpack_int(min_addr)
    `uvm_unpack_int(max_addr)
    `uvm_unpack_int(min_data_len)
    `uvm_unpack_int(max_data_len)
    `uvm_unpack_int(max_trans)
  endfunction

  function string convert2string();
    return $sformatf("min_addr:%h max_addr:%h min_data_len:%0d max_data_len:%0d max_tran
```

```
    min_addr, max_addr, min_data_len, max_data_len,max_trans);
  endfunction

endclass
```

50.2 producer

A simple SV producer TLM model that generates a configurable number of *uvm_tlm_generic_payload* transactions and sends them to its *out* port for execution. The transaction is also broadcast to its *ap* analysis port.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM ports to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might be receiving the transactions sent via its *out* and *ap* ports.

Because this producer is used for all the Command API examples, for

50.2.1 Methods

50.2.2 Phases

We implement each phase to simply print a message that the phase has started. The UVMC Command API Example - Phase Control will show that SC can be synchronized to UVM phases and even prevent the task phases from ending.

```
function void build_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","BUILD Started",UVM_NONE);
endfunction

function void connect_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","CONNECT Started",UVM_NONE);
endfunction

function void end_of_elaboration_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","END_OF_ELABORATION Started",UVM_NONE);
endfunction

function void start_of_simulation_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","START_OF_SIMULATION Started",UVM_NONE);
endfunction

task pre_reset_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","PRE_RESET Started",UVM_LOW);
endtask

task reset_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","RESET Started",UVM_LOW);
endtask

task post_reset_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","POST_RESET Started",UVM_LOW);
endtask
```

```
task pre_configure_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","PRE_CONFIGURE Started",UVM_LOW);
endtask

task configure_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","CONFIGURE Started",UVM_LOW);
endtask

task post_configure_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","POST_CONFIGURE Started",UVM_LOW);
endtask

task pre_main_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","PRE_MAIN Started",UVM_LOW);
endtask

task main_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","MAIN Started",UVM_LOW);
endtask

task post_main_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","POST_MAIN Started",UVM_LOW);
endtask

task pre_shutdown_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","PRE_SHUTDOWN Started",UVM_LOW);
endtask

task shutdown_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","SHUTDOWN Started",UVM_LOW);
endtask

task post_shutdown_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","POST_SHUTDOWN Started",UVM_LOW);
endtask

function void extract_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","EXTRACT Started",UVM_LOW);
endfunction

function void check_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","CHECK Started",UVM_LOW);
endfunction

function void report_phase(uvm_phase phase);
  `uvm_info("UVMC_PHASING","REPORT Started",UVM_LOW);
endfunction
```

50.2.3 new

Creates a new producer object. Here, we allocate the *out* port and *ap* analysis port. If +PHASING_ON is not on the command line, we disable the UVMC_PHASING messages that are emitted by each phase callback (see above).

```
function new(string name, uvm_component parent=null);
   super.new(name,parent);
   out = new("out", this);
   analysis_out = new("analysis_out", this);
```

```
   if ($test$plusargs("PHASING_ON"))
     set_report_id_action("UVMC_PHASING",UVM_NO_ACTION);
endfunction : new


// Function: check_config
//
// Enabled only during the <UVMC Command API Example - Configuration>,
// the ~check_config~ function gets the configuration parameters
// that the SC side should have set. It produces ERRORs in cases
// where a get was not successful.

// (begin inline source)
prod_cfg cfg = new();

function void check_config();

  int i;
  string str;
  uvm_object obj;

    if (!uvm_config_db #(uvm_bitstream_t)::get(this,"","some_int",i))
      `uvm_error("NO_INT_CONFIG",{"No configuration for field 'some_int'",
        " found at context '",get_full_name(),"'"})
    else
      `uvm_info("INT_CONFIG",
        $sformatf("Config for field 'some_int' at context '%s' has value %0h",
        get_full_name(),i),UVM_NONE)


    if (!uvm_config_db #(string)::get(this,"","some_string",str))
      `uvm_error("NO_STRING_CONFIG",{"No configuration for field 'some_string'",
        " found at context '",get_full_name(),"'"})
    else
      `uvm_info("STRING_CONFIG",
        {"Config for field 'message' at context '", get_full_name(),
        "' has value '", str,"'"},UVM_NONE)

    if (!uvm_config_db #(uvm_object)::get(this,"","config",obj))
      `uvm_error("NO_OBJECT_CONFIG",{"No configuration for field 'config'",
        " found at context '",get_full_name(),"'"})
    else begin
       prod_cfg c;
       if (!$cast(c,obj))
         `uvm_error("BAD_CONFIG_TYPE",
            {"Object set for configuration field 'config' at context '",
            get_full_name(),"' is not a prod_cfg type"})
       else begin
         cfg = c;
         `uvm_info("OBJECT_CONFIG",
            {"Config for field 'config' at context '",get_full_name(),
            "' has value '", cfg.convert2string(),"'"},UVM_NONE)
       end
    end

endfunction
```

50.2.4 check_config

Enabled only during the <UVMC Command API Example

Configuration>, the *check_config* function gets the configuration parameters that the SC side should have set. It produces ERRORs in cases where a get was not successful.

50.2.5 run_phase

Produces the configured number of transactions, sending each to its *out* and *ap* analysis ports. A prod_cfg configuration object governs how many transactions are produced and constrains the address range and data array length during randomization. Upon return from sending the last transaction, the producer drops its objection to ending the run_phase, thus allowing simulation to proceed to the next phase.

```
task run_phase (uvm_phase phase);

  uvm_tlm_generic_payload pkt = new;
  uvm_tlm_time delay = new;

  bit enable_config_check = $test$plusargs("CONFIG_ON");
  bit enable_stimulus = $test$plusargs("TRANS_ON");
  pkt.m_streaming_width.rand_mode(0);
  pkt.m_byte_enable_length.rand_mode(0);
  pkt.m_byte_enable.rand_mode(0);
  pkt.m_data = new[1];

  `uvm_info("UVMC_PHASING","RUN Started",UVM_LOW);

  phase.raise_objection(this);

  if (enable_config_check)
    check_config();

  if (enable_stimulus && cfg != null) begin

    for (int i = 1; i <= cfg.max_trans; i++) begin

      if (!pkt.randomize() with {
         m_address inside { [cfg.min_addr:cfg.max_addr] };
         m_data.size() inside { [cfg.min_data_len:cfg.max_data_len] }; })
        `uvm_error("RAND_FAILED",  "Randomization of tlm_gp failed")

      pkt.set_data_length(pkt.m_data.size());

      delay.set_abstime(11,1e-9);

      $display();
      `uvm_info("PRODUCER/SEND_PKT",
         $sformatf("SV producer sending packet #%0d\n  %s",i,
                   pkt.sprint(uvm_default_line_printer)),UVM_MEDIUM)

      analysis_out.write(pkt);
      out.b_transport(pkt,delay);
    end

  end

  #1000;

  `uvm_info("PRODUCER/STOP_TEST","Stopping the test",UVM_LOW)
```

```
      phase.drop_objection(this);

   endtask

endclass
```

50.3 producer_ext

This trivial extension of our producer class is used to demonstrate factory overrides from SC using the UVMC Command API.

```
class producer_ext extends producer;

   `uvm_component_utils(producer_ext)

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      `uvm_info("PRODUCER_EXTENSION","Derived producer created!",UVM_NONE);
   endfunction

endclass
```

50.4 scoreboard

A simple SV consumer TLM model that prints received transactions (of type ~tlm_generic_payload) and sends them out its *ap* analysis port.

While trivial in functionality, the model demonstrates use of TLM ports to facilitate external communication.

- Users of the model are not coupled to its internal implementation, using only the provided TLM exports to communicate.
- The model itself does not refer to anything outside its encapsulated implementation. It does not know nor care about what might be driving its analysis exports.

50.5 scoreboard_ext

This trivial extension of our scoreboard class is used to demonstrate factory overrides from SC using the UVMC Command API.

```
class scoreboard_ext extends scoreboard;

   `uvm_component_utils(scoreboard_ext)

   function new(string name, uvm_component parent=null);
      super.new(name,parent);
      `uvm_info("SCOREBOARD_EXTENSION","Derived scoreboard created!",UVM_NONE);
   endfunction

endclass
```

50.6 env

Our SV *env* contains an instance of our producer and scoreboard, above.

```
class env extends uvm_env;

   `uvm_component_utils(env)

   uvm_tlm_b_transport_port #(uvm_tlm_generic_payload) prod_out;
   uvm_analysis_export #(uvm_tlm_generic_payload) sb_actual_in;

   producer prod;
   scoreboard sb;

   function new(string name, uvm_component parent=null);
     super.new(name,parent);
     prod_out = new("prod_out",this);
     sb_actual_in = new("sb_actual_in",this);
   endfunction

   function void build();
     prod = producer::type_id::create("prod",this);
     sb = scoreboard::type_id::create("sb",this);
   endfunction

   function void connect();
     prod.analysis_out.connect(sb.expect_in);
     prod.out.connect(prod_out);
     sb_actual_in.connect(sb.actual_in);
   endfunction

endclass
```

50.7 sv_main

module sv_main

This is the top-level module for the SV side of each command API example.

This top-level SV module does the following

- Initializes the UVMC Command API layer by calling *uvmc_init*. This is required. You can also relegate the init call to a separate module that is compiled separately or on the same command line as this file.
- Registers the env's *prod_out* and *sb_actual_in* ports for UVMC communication.
- Calls *run_test* to start the SV portion of the simulation.

We could have registered the UVMC connections in the env's *connect* method, but that would have forced the *env* to only work with UVMC. If you prefer to relagate UVMC registration to the env or lower, you should not mix it in with the *env's* main purpose. Instead, try to add the UVMC connection code to a simple extension/wrapper around your original model. This technique is demonstrated in <SC to SV Connection-SC side).

```
module sv_main;

  import uvmc_pkg::*;

  env e;

  // Must initialize
```

```
  initial
    uvmc_init();

  initial begin
    e = new("e");

    $timeformat(-9,0," ns");

    // actual path - SC-side consumer to SV-side scoreboard
    uvmc_tlm #(uvm_tlm_generic_payload)::connect(e.prod_out,"foo");
    uvmc_tlm1 #(uvm_tlm_generic_payload)::connect(e.sb_actual_in,"bar");

    run_test();

  end

endmodule
```

class prod_cfg extends uvm_object

The *prod_cfg* class is the configuration object used by our producer below.

class producer extends uvm_component

A simple SV producer TLM model that generates a configurable number of *uvm_tlm_generic_payload* transactions and sends them to its *out* port for execution.

class producer_ext extends producer

This trivial extension of our producer class is used to demonstrate factory overrides from SC using the UVMC Command API.

class scoreboard extends uvm_component

A simple SV consumer TLM model that prints received transactions (of type ~tlm_generic_payload) and sends them out its *ap* analysis port.

class scoreboard_ext extends scoreboard

This trivial extension of our scoreboard class is used to demonstrate factory overrides from SC using the UVMC Command API.

class env extends uvm_env

Our SV *env* contains an instance of our producer and scoreboard, above.

module sv_main

This is the top-level module for the SV side of each command API example.
An API that provides access UVM's phase state and the objection objects used to control phase progression.
This example demonstrates usage the set_config and get_config portion of the UVMC Command API.
This code provides an example of waiting for each UVM phase to reach a specified state and then, if the phase is a task phase, controlling its progression by raising and dropping the objection that governs it.

51 UVMC Command API Examples - SC Consumer

51.1 Description

A generic consumer that receives and processes transactions coming from its blocking-transport *in* export. It performs no meaningful functionality--it prints the transaction, waits the specified delay, then sends it out its analysis port.

../../../uvmc/examples/commands/consumer.cpp

```
//
//------------------------------------------------------------//
//   Copyright 2009-2012 Mentor Graphics Corporation          //
//   All Rights Reserved Worldwid                             //
//                                                            //
//   Licensed under the Apache License, Version 2.0 (the      //
//   "License"); you may not use this file except in          //
//   compliance with the License.  You may obtain a copy of   //
//   the License at                                           //
//                                                            //
//       http://www.apache.org/licenses/LICENSE-2.0           //
//                                                            //
//   Unless required by applicable law or agreed to in        //
//   writing, software distributed under the License is       //
//   distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR   //
//   CONDITIONS OF ANY KIND, either express or implied.  See  //
//   the License for the specific language governing          //
//   permissions and limitations under the License.           //
//------------------------------------------------------------//

#include "systemc.h"
#include "tlm.h"
#include
```