UVM-ML Adapter developer Guide

For version UVM-ML 1.3

14 October, 2013

Notices

Questions or suggestions relating to this document or product can be sent to:
uvm_contributions@cadence.com

Table of Contents

Table of Diagrams

Overview

UVM-ML is an extension of UVM methodology, intended to provide UVM like capabilities in the multi-language environment. It is implemented in the UVM-ML release package UVM-ML-1.3.tar which includes the implementation of the UVM-ML backplane and several adapters which connect various frameworks through the backplane.

This guide is intended for the adapter developer. It explains the requirements from the adapter and shows examples of how to use it. Additional relevant information can be found in

- "UVM-ML Whitepaper" which explains the general approach of UVM-ML methodology
- "UVM-ML Integrator User Guide" which explains how to use adapters to create a UVM-ML verification environment.

The examples included in this guide are based on the UVM-SV and UVM-SC adapters included in the UVM-ML release package mentioned above.

Main Features of UVM-ML

The adapters feature set is based on the UVM methodology. The adapters extend the UVM capabilities to operate in the multi-language environment by facilitating framework cooperation through the backplane.

The main facilities supported by the adapter:

- Start-up and framework registration
  - Register the framework
  - Publish required API implementation
- TLM communication
  - Register ML connectors
  - Bind ML connectors
  - Implement the serialization mechanism
- Unified hierarchy
  - Create a child component in a remote framework
- Phasing
  - Manage phasing according to the unified hierarchy
- Time management and synchronization

- Report current time
- Synchronize frameworks

Additional features and services may be supported in the future.

Adapters

Every framework that participates in the UVM-ML testbench needs an adapter. Adapters bridge between the user code in the framework and the backplane. The main role of the adapter is to:

- implement user requests related to actions in other frameworks and propagate the necessary information to the backplane
- implement backplane calls and propagate the relevant information to the framework

Internally the adapters contain various capabilities for the use of the integrator and the end users. The integrator uses the adapters to construct the multi-language environment by using the TLM connection and unified hierarchy capabilities. The end user uses the TLM interfaces to transfer transactions between frameworks. The diagram below shows how the adapter provides serialization and other services.

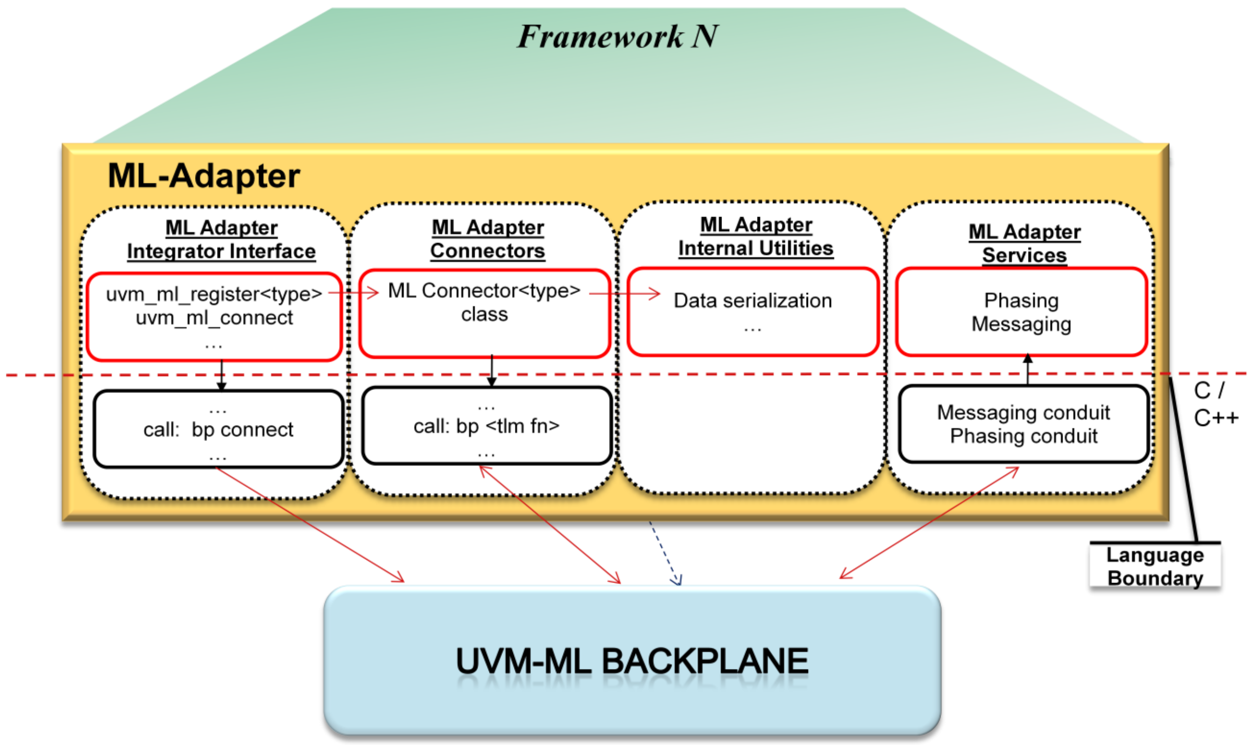

Diagram 1. Adapter structure

The ml/adapters directory in the release package contains two examples, a UVM-SV adapter under ml/adapters/uvm_sv and a UVM-SC adapter under ml/adapters/uvm_sc. These are used below to demonstrate the various capabilities of adapters.

Note: The UVM-***e*** adapter is not used as an example in this document because of the unique capabilities of the ***e*** language which leaves the adapter very small and un-typical.

The UVM-SV adapter implementation is in ml/adapters/uvm_sv/uvm_ml_adapter.sv. It is implemented on top of the UVM-SV framework which is an ML-enabled version of uvm-1.1c, located in ml/frameworks/uvm/sv/uvm-1.1c.

The UVM-SC adapter implementation is in ml/adapters/uvm_sc/common/uvm_ml_adapter.*. It is implemented on top of the UVM-SC framework in ml/frameworks/uvm/sc/base. This framework adds UVM style capabilities on top of SystemC.

Adapter Interfaces

The adapter has two interfaces, the required and provided API to communicate with the backplane and the user facing API to be used by the framework. The end user communicates through the user facing API and the adapter implements these API requests either internally in the adapter or through the backplane API.

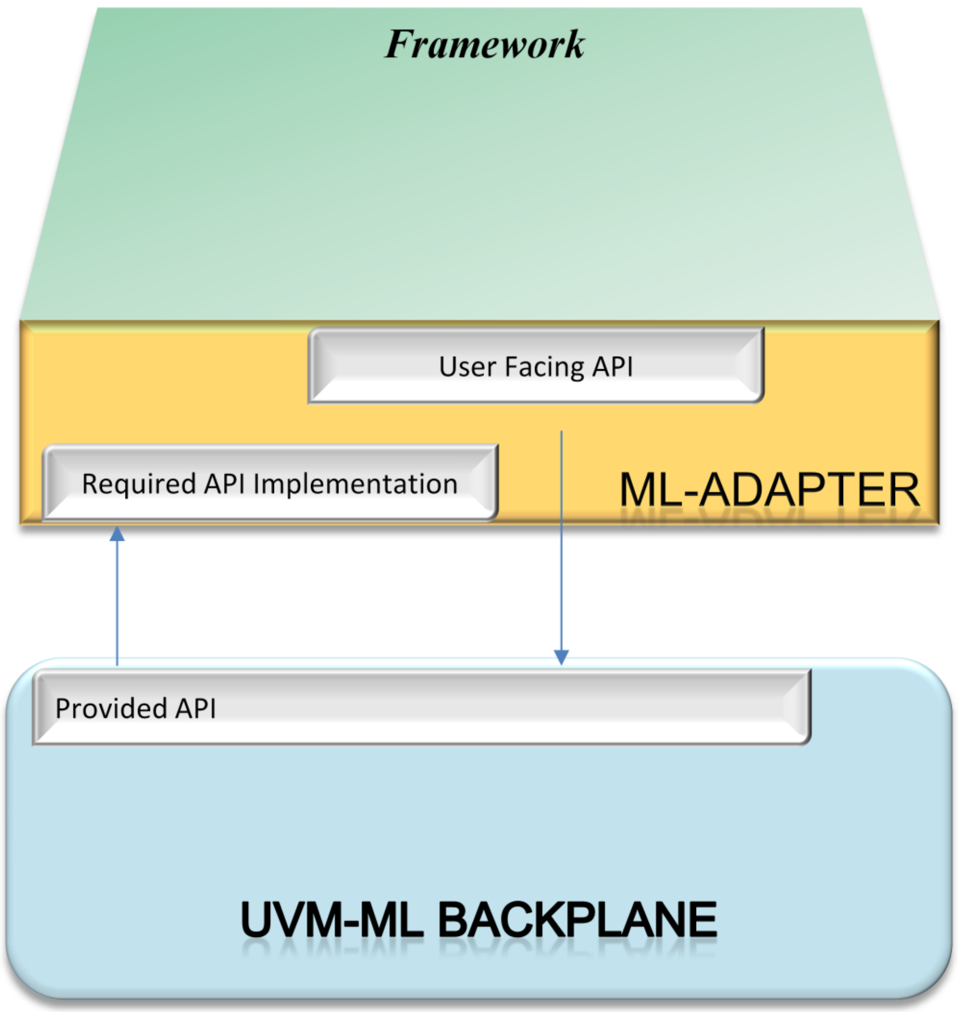

Diagram 2. Adapter interfaces

Required API
The required API is defined in the backplane as a template bp_frmw_c_api_struct in ml/backplane/bp_required.h. This is a struct that contains a list of pointers to the methods and tasks that must be implemented by each adapter. This API is discussed in detail in “Implementing the Required API” below.

Provided API
The provided API is defined in the backplane in bp_api_struct in ml/backplane/bp_provided.h. This is a struct that contains a list of pointers to the methods that constitute the backplane API and are available to be used by the adapters. This API is discussed in detail in “Using the Provided API” below.

User Facing API
The user facing API is specific to each framework. The adapter developer decides which features to implement according to the framework’s capabilities and needs. This API is discussed in detail in “Implementing the User Facing API” below.

Framework Modifications
Occasionally there is need to do some small adjustments in the framework itself to be able to support the required functionality. These changes should be kept minimal and if possible should be implemented in a way that will be forward compatible i.e. will apply to future versions of the framework.

Typical examples of such modifications are gaining control over time management in a simulator that is not designed to run in slave mode, or avoiding undesired race conditions between the testbench construction and other initialization code.

Modifications in the OSCI SystemC simulator
The OSCI SystemC simulator is not designed to run in slave mode under another master simulator. By modifying the time management one can avoid situations where the simulator advances time beyond the master simulator when the OSCI simulator is not the master simulator.

The modifications to the OSCI SystemC simulator are implemented in ml/frameworks/osci/sysc/kernel/sc_simcontext.* labeled “ML enabling additions”. The sc_simcontext module is replaced in the library to create an ML enabled version of the library.

Modifications in the UVM-SV library
In the case of SystemVerilog there is a need to avoid the inherent race between arbitrary initial and always blocks and the construction of the testbench.

The modifications to UVM-SV can be found in ml/frameworks/uvm/sv/uvm-1.1c. In this case the entire UVM-SV package is replaced using the –uvmhome command line flag which points to the desired version of UVM-SV. The ML specific changes are labeled “ML enabling fix”.

Implementing the Required API

The required API is a “tray” containing the list of method pointers that must be implemented in every adapter. The backplane uses these methods to communicate with the adapter. All the methods must be implemented but some may have empty implementation if the corresponding feature is not supported by the framework.

The UVM-SC required API is established when uvm_ml_bootstrap() is called, using the following function

```
 static bp_frmw_c_api_struct* uvm_ml_sc_get_required_api()
```

which creates a new struct of type bp_frmw_c_api_struct and returns its pointer to the calling backplane. This function also sets the pointers in the required_api struct to the appropriate implementation functions.

The UVM-SV required API is implemented as export DPI-C functions defined in ml/adapters/uvm_sv/uvm_ml_export_dpi.svh

The full list of API can be found in ml/backplane/bp_required.h. The implementation in the UVM-SC adapter is in class uvm_ml_tlm_rec in ml/adapters/uvm_sc/common/uvm_ml_adapter.cpp. The implementation in the UVM-SV adapter is in package uvm_ml in ml/adapters/uvm_sv/uvm_ml_adapter.sv.

Below is a sample of interfaces to demonstrate the various capabilities.

Unified Hierarchy and Phasing

- Create a child component for a foreign parent component in another framework.

The adapter instantiates a parent proxy. The constructor of the parent proxy creates the requested child component with the provided arguments.

The implementation in the UVM-SC adapter is as follows:

```
int uvm_ml_tlm_rec::create_child_junction_node(
  const char * component_type_name,    // type of the child component
  const char * instance_name,          // instance name of the child component
  const char * parent_full_name,       // full name of the remote parent component
  int          parent_framework_id,    // unique ID of the initiator framework
  int          parent_junction_node_id // unique ID of the parent component
);
```

The implementation in the UVM-SV adapter is as follows:

```
function int export_create_child_junction_node(
  string component_type_name,    // type of the child component
  string instance_name,          // instance name of the child component
  string parent_full_name,       // full name of the remote parent component
  int    parent_framework_id,    // unique ID of the initiator framework
  int    parent_junction_node_id // unique ID of the parent component
 );
```

- Propagate the phase from a remote parent component.

Pre, post and runtime phases must be propagated top down on the unified hierarchy tree. The parent proxy indicates the start of a new phase through the appropriate API in the backplane. The backplane propagates the phase change to the appropriate parent proxy. The adapter is responsible for triggering the phase in the appropriate parent proxy.

The implementation in the UVM-SC adapter is as follows (in uvm_ml_phase.cpp):

```
int uvm_ml_phase::transmit_phase(
    int                  target_id,   // unique ID of the parent proxy
    const char *         phase_group, // name of the phase group
    const char *         phase_name,  // name of the phase
    uvm_ml_phase_action phase_action // STARTED, EXECUTING or ENDED
);
```

The implementation in the UVM-SV adapter is as follows:

```
function int export_transmit_phase(
    int unsigned           target_id,   // unique ID of the parent proxy
    string                 phase_group, // name of the phase group
    string                 phase_name,  // name of the phase
    uvm_ml_phase_action_e phase_action // STARTED, EXECUTING or ENDED
);
```

TLM Communication

- Implementation of non-blocking put from a remote port in some other framework.

The adapter unpacks the bit-stream into the appropriate type and then sends it using nb_put of the appropriate interface to the target export in the user code.

The implementation in the UVM-SC adapter is as follows:

```
int uvm_ml_tlm_rec::nb_put(
  unsigned          connector_id, // unique ID of the connector
  unsigned          stream_size,  // size of data stream
  uvm_ml_stream_t   stream,       // data stream sent to initiator
  uvm_ml_time_unit time_unit,    // current time unit
  double            time_value    // current time value
);
```

The implementation in the UVM-SV adapter is as follows:

```
function bit export_try_put(
  int unsigned connector_id, // unique ID of the connector
  int unsigned stream_size,  // size of data stream
  `STREAM_T    stream        // data stream sent to initiator
);
```

Note: In the UVM-SV adapter there is no use for the time stamp, so the time parameters are discarded when the API is called.

Framework Synchronization

- ➢ Synchronize the framework to the time broadcasted from the master simulator.

The implementation in the UVM-SC adapter is as follows:

```
void uvm_ml_tlm_rec::synchronize(
  uvm_ml_time_unit time_unit, // current time unit
  double           time_value // current time value
);
```

Using the Provided API

The adapter uses the API provided by the backplane to communicate with other frameworks and to request services.

The full list of API can be found in ml/backplane/bp_provided.h. Below is a sample of interfaces to demonstrate the various capabilities.

Check Backplane version number

The backplane has a version number which identifies the provided and required API. Whenever the API is modified, the version number is advanced. The adapter developer must check the backplane version number and take the appropriate action if the version number does not match the expected version.

The backplane API is as follows:

```
typedef char* (*bp_get_version_type) ();
```

The version is provided as a string, e.g.:

```
#define UVM_ML_VERSION "1.2.2"
```

Initialize the Framework Backplane Channel

- ➢ Register the framework at the backplane.

Each framework must register and receive a unique ID which is used later on to identify this framework. While registering, the adapter publishes its implementation of required API in the form of the pointer to the “tray” of API pointers.

The call in the UVM-SC adapter is as follows:

```
#define BP(f) (*bpProvidedAPI->f##_ptr)
unsigned uvm_ml_utils::initialize_adapter()
{
    ...
    char *frmw_ids[3] = {(char*)"UVMSC", (char*)"SC",(char*)""};
    return BP(register_framework)((char*)"SystemC",
                                  frmw_ids,
                                  uvm_ml_sc_get_required_api());
}
```

Note: The framework may have multiple IDs. If it is the only SystemC framework it can be identified simply as "SC", but if there are multiple SystemC frameworks than a more specific name can be used like "UVMSC", to differentiate from another framework that is not UVM-SC based.

The call in the UVM-SV adapter is as follows:

```
int uvm_ml_sv_register_framework()
{
    ...
    char * frmw_ids[3] = {(char *)"UVMSV", (char *)"SV", (char *)""};
    uvm_sv_framework_id = BP(register_framework)((char *) "UVM SV",
                                                 frmw_ids,
                                                 sv_get_required_api());
    ...
    return uvm_sv_framework_id;
}
```

Unified Hierarchy and Phasing

- Create a remote child component in another framework.

The adapter instantiates a child proxy when the user facing API requests a remote child created. The constructor of the child proxy calls this API in the backplane, passing the necessary parameters to the target framework where the adapter creates the matching parent proxy.

The UVM-SC adapter calls this API in the constructor of the child proxy as follows:

```
child_component_proxy::child_component_proxy(sc_module_name nm): uvm_component(nm)
{
    get_config_string("uvm_ml_parent_name", m_parent_name);
    get_config_string("uvm_ml_type_name",   m_component_type_name);
    get_config_string("uvm_ml_inst_name",   m_instance_name);
    get_config_string("uvm_ml_frmw_id",     m_target_frmw_ind);
    uvm_component *p = get_parent();
    m_parent_id = get_hierarchical_node_id(p);
    m_child_junction_node_id = BP(create_child_junction_node)(
                                 framework_id,
                                 m_target_frmw_ind.c_str(),
                                 m_component_type_name.c_str(),
                                 m_instance_name.c_str(),
                                 m_parent_name.c_str(),
                                 m_parent_id);
}
```

The UVM-SV adapter calls this API in a function of the child_component_proxy as follows:

```
    function int create_foreign_child_junction_node(string target_frmw_ind,
                                                    string component_type_name);
        uvm_component p;
        p = get_parent();
        assert (p != null);
        m_parent_id = get_hierarchical_node_id(p);
        m_target_frmw_ind = target_frmw_ind;
        m_component_type_name = component_type_name;
        m_child_junction_node_id  = uvm_ml_create_child_junction_node(
                                          m_target_frmw_ind,
                                          m_component_type_name,
                                          get_name(),
                                          p.get_full_name(),
                                          m_parent_id);
        return m_child_junction_node_id; // (-1) if failed
    endfunction : create_foreign_child_junction_node
```

➢ Propagate a phase to a remote child component in another framework.

Pre, post and runtime phases must be propagated top down on the unified hierarchy tree. The child proxy indicates the start of a new phase through the appropriate API in the backplane. The backplane propagates the phase change to the appropriate parent proxy.

The UVM-SC adapter propagates the build phase from the child proxy as follows:

```
void child_component_proxy::build_phase(uvm_phase *phase) {
  int res = BP(transmit_phase)(framework_id,
                               m_target_frmw_ind.c_str(),
                               m_child_junction_node_id,
                               "common",
                               "build",
                               UVM_ML_PHASE_EXECUTING);
}
```

The UVM-SV adapter propagates the build phase from the child proxy as follows:

```
    function void build_phase(uvm_phase phase);
        int res = uvm_ml_transmit_phase(m_target_frmw_ind,
                                        m_child_junction_node_id,
                                        "common",
                                        "build",
                                        UVM_ML_PHASE_EXECUTING);
    endfunction
```

TLM Communication

- Connect port to export or initiator to target socket.

This call to the backplane registers the connection in the backplane repository. The ports and sockets are identified by their unified hierarchy path.

The UVM-SC adapter propagates the connect command from the user facing API as follows:

```
unsigned uvm_ml_utils::do_connect (const std::string & port_name,
                                   const std::string & export_name) {
    return BP(connect)(framework_id,port_name.c_str(),export_name.c_str());
}
```

The UVM-SV adapter propagates the connect command from the user facing API as follows:

```
function bit connect(string path1, string path2, bit map_transactions);
    bit connection_result;
    connection_result = uvm_ml_connect(path1,path2);
    ...
    return connection_result;
endfunction : connect
```

- Initiate TLM1 non-blocking put transaction through an ML interface.

The transaction data initiated in the framework is serialized to a bit-stream and then sent to the backplane. The backplane uses its repository to find the target of this interface and deliver the bit-stream. In addition, every data transaction carries also the current time to facilitate time synchronization between the frameworks.

The UVM-SC adapter forwards the call from the user facing API as follows:

```
int uvm_ml_tlm_trans::nb_put(unsigned local_port_id, MLUPO* val) {
    return BP(nb_put)(framework_id,
                      local_port_id,
                      val->nblocks,
                      val->val,
                      m_time_unit,
                      m_time_value
                     );
}
```

The UVM-SV adapter forwards the call from the user facing API as follows:

```
virtual function bit try_put(input T1 t);
    int       packed_size;
    `STREAM_T packed_stream;
    packed_size = uvm_ml_serialization_kit::pack_cur_stream(t, packed_stream);
    return uvm_ml_nb_put(m_conn_id, packed_size, packed_stream);
endfunction : try_put
```

Framework Synchronization

- Broadcast time advancement through the backplane to slave simulators.

For example when the OSCI SystemC simulator is running in slave mode under some other master simulator, the master simulator must synchronize the slave simulator using this API.

The UVM-SV adapter initiates synchronization through the backplane as follows:

```
task synchronize();
    uvm_ml_adapter_imp::uvm_ml_synchronize();
endtask : synchronize
```

Implementing the User Facing API

The user facing API is defined and implemented by the adapter developer according to the capabilities of the framework. Some of the adapter features are necessary and some are optional.

This section shows examples of the main features that may be provided by the adapter.

- Support TLM communication
- Support unified hierarchy and phasing

Support TLM communication

TLM communication is probably the most important feature to be supported by an adapter. The full support covers TLM1, analysis ports and TLM2, both blocking and non-blocking interfaces. In some cases partial support may be sufficient.

Since each framework has rules about what must be connected, the adapter must obey these rules. For example if a TLM1 put port is intended to be connected to a put export in another framework, the adapter must provide a local “proxy” export to satisfy the need for matching export, and implement the proxy export to perform the serializing of the transaction and forwarding it through the backplane API. Similarly the put export needs a matching “proxy” put port instantiated in the adapter, which will get the bit-stream from the backplane, de-serialize it and send it to the user defined export.

Port Registration

The “proxy” ports are created when the user registers a port as an ML port. In the UVM-SV adapter this is done by a templated method for example for analysis port of T:

```
function void register(uvm_port_base #(uvm_tlm_if_base #(T1,T2)) port, string T1_name,
string T2_name);
      ml_tlm1_connector #(T1, T2) conn;
      ...
      conn = new (port, UNSPECIFIED_ML_IMP_DIRECTION, T1_name, T2_name);
      add_connector(conn);
  endfunction:
```

The implementation in the UVM-SC adapter is as follows:

```
template <typename T>
void uvm_ml_register_internal(
  tlm::tlm_analysis_port<T>* p
) {
  std::string s = std::string(p->basename()) + "_trans";
  uvm_ml_tlm_transmitter_tlm<T,T>* trans =
    new uvm_ml_tlm_transmitter_tlm<T,T>(s.c_str());
  p->bind(*trans);
  trans->object(p);
  trans->set_intf_name("tlm_analysis_if");
  trans->set_REQ_name(T().get_type_name());
  trans->set_RSP_name(T().get_type_name());
}
```

Implement the Port Proxy

When the framework calls the write method of an analysis port that was declared as an ML port, the “proxy” implementation forwards the transaction through the backplane API.

The implementation in the UVM-SC adapter is as follows:

```
void uvm_ml_tlm_trans::write(
  unsigned local_port_id,
  MLUPO *  val
) {
    BP(write)(
      framework_id,
      local_port_id,
      val->nblocks,
      val->val,
      m_time_unit,
      m_time_value
  );
}
```

Where BP is the convenience macro:

```
#define BP(f) (*bpProvidedAPI->f##_ptr)
```

The implementation in the UVM-SV adapter is as follows:

```
  virtual function void write(input T1 t);
      int                 packed_size;
      `STREAM_T           packed_stream;
      packed_size = uvm_ml_serialization_kit::pack_cur_stream(t, packed_stream);
      uvm_ml_write(m_conn_id, packed_size, packed_stream);
  endfunction : write
```

Implement the Export Proxy

When the backplane calls the write API indicating that a remote analysis port sent a transaction, the adapter identifies the corresponding export and forwards the transaction to it.

The implementation in the UVM-SC adapter is as follows:

```
void uvm_ml_tlm_rec::write(
  unsigned          connector_id,
  unsigned          stream_size,
  uvm_ml_stream_t   stream,
  uvm_ml_time_unit time_unit,
  double            time_value
) {
  uvm_ml_tlm_receiver_base* rec = 0;
  try {
    ENTER_CO_SIMULATION_CONTEXT();
    rec = get_receiver_base(connector_id);
    ...
    rec->write_bitstream(stream_size,stream);
    EXIT_CO_SIMULATION_CONTEXT();
  }
  CATCH_KERNEL_EXCEPTION_IN_RECEIVER_2_VOID
}
```

The implementation in the UVM-SV adapter is as follows:

```
  function void export_write(int unsigned stream_size, `STREAM_T stream);
      T1 arg;
      $cast(arg, uvm_ml_serialization_kit::create_and_unpack_cur_stream(
                                                                          stream_size,
                                                                          stream));
      m_actual_port.write(arg);
  endfunction : export_write
```

Blocking Interface

Supporting blocking calls between frameworks require special consideration because different frameworks may have different stacks. The UVM-SC and UVM-SV adapters implement "fake" blocking by separating the call to initiating, waiting and completion steps.

Initiating a blocking call is done by sending the transaction to the target framework using the request interface. For example the TLM1 blocking get uses the following backplane interface:

The implementation in the UVM-SC adapter is as follows:

```
  int disable = BP(request_get)(
    framework_id,
    port_connector_id,
    call_id,
    &(arg->nblocks),
    arg->val,
    &done,
    &m_time_unit,
    &m_time_value
  );
```

Next the calling adapter registers an event to be triggered when the call is done and waits. When the target framework implementation is done, the backplane indicates "notify_end_blocking" which triggers the appropriate event to release the calling adapter. Then the result is obtained through the backplane by the get_requested API.

```
    arg->nblocks = BP(get_requested)(
      framework_id,
      port_connector_id,
      call_id,
      arg->val
    );
```

The implementation in the UVM-SV adapter is as follows:

```
  virtual task get(output T2 t);
      int                     packed_size;
      `STREAM_T               packed_stream;
      blocking_call_id_class call_id_object;
      int unsigned           j;
      int unsigned           done;
      int                    request_result;
      call_id_object = add_blocking_call_id();
      uvm_ml_get_request(m_conn_id,
                         call_id_object.call_id,
                         packed_size,
                         packed_stream, done);
      if (!done) begin
          call_id_object.wait_end_blocking();
          packed_size = uvm_ml_get_requested(m_conn_id,
                                             call_id_object.call_id,
                                             packed_stream);
      end
      else remove_blocking_call_id(call_id_object.call_id);
      ...
  endtask
```

Serialization

Data transferred between frameworks is passed through the backplane as a bit-stream. The adapters are responsible for serializing and de-serializing the data based on the matching type definitions in the frameworks. The serialization algorithm is aligned with the existing UVM mechanisms, so both the UVM-SV and UVM-SC adapters make use the existing mechanism.

The UVM-SC adapter uses the uvm_packer provided in UVM-SC. The code for the packer can be found in ml/frameworks/uvm/sc/base/uvm_packer.*. The code for the UVM-SC serializer is in ml/adapters/uvm_sc/common/uvm_ml_packer.*.

The UVM-SV adapter uses the built in mechanism in UVM-SV that provides packing for uvm_object and uvm_component when declared by the uvm_object_utils macro. The code for the UVM-SV adapter serializer is in ml/adapters/uvm_sv/uvm_ml_serializer.*.

Support for Unified Hierarchy and Phasing

This feature enables the framework to participate in the construction of an ML logical hierarchy created from sub-trees implemented in different frameworks. If this feature is not supported by the adapter, one can still create stand-alone sub-trees that are not connected to the unified hierarchy.

To support unified hierarchy the adapter must define and implement the parent_proxy and child_proxy classes. An example can be found in ml/adapters/uvm_sc/common/uvm_ml_hierarchy.*.

Child Proxy

The child proxy represents a remote child in a foreign framework. It is implemented as a uvm_component created by the parent. The constructor of the class sends the request to create the corresponding child component through the backplane API. In addition, the phases detected by this child component are propagated to the remote child through the backplane API.

The implementation in the UVM-SC adapter is as follows:

```
class child_component_proxy: public uvm_component {
private:
  string       m_component_type_name;
  string       m_instance_name;
  string       m_parent_name;
  string       m_target_frmw_ind;
  int          m_child_junction_node_id;
  int          m_parent_id;
public:
  child_component_proxy(sc_module_name);
  virtual ~child_component_proxy() {};
  void phase_started(uvm_phase *phase);
  ...
  UVM_COMPONENT_UTILS(child_component_proxy)
};
UVM_COMPONENT_REGISTER(child_component_proxy)
```

The implementation in the UVM-SV adapter is as follows:

```
class child_component_proxy extends uvm_component;
    int    m_parent_id;
    string m_target_frmw_ind;
    string m_component_type_name;
    int    m_child_junction_node_id;
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction : new
    function int create_foreign_child_junction_node(string target_frmw_ind,
                                                     string component_type_name);
       ...
        m_child_junction_node_id  = uvm_ml_create_child_junction_node(
                                    m_target_frmw_ind,
                                    m_component_type_name,
                                    get_name(),
                                   p.get_full_name(),
                                   m_parent_id);
        return m_child_junction_node_id; // (-1) if failed
    endfunction : create_foreign_child_junction_node
    function void phase_started(uvm_phase phase);
        ...
    endfunction
    ...
endclass : child_component_proxy
```

Parent Proxy

The parent proxy is implemented as a uvm_component created by the required API to create a sub-tree. Its create method instantiates the requested child component under the parent proxy.

The implementation in the UVM-SC adapter is as follows:

```
class parent_component_proxy: public uvm_component {
private:
  int  m_proxy_id;
  int  m_parent_framework_id;
public:
  parent_component_proxy(const sc_module_name & nm);
  virtual ~parent_component_proxy() {};
  UVM_COMPONENT_UTILS(parent_component_proxy)
static parent_component_proxy * create(const string & parent_full_name,
                                       int            parent_framework_id,
                                       int            parent_junction_node_id);
  int Id() { return m_proxy_id; }
  void SetId(int id) { m_proxy_id = id; }
  int ParentFrameworkId() { return m_parent_framework_id; }
  void SetParentFrameworkId(int id) { m_parent_framework_id = id; }
  int  add_node(const char * child_type_name, const char * child_instance_name);
  class FullNameComparer { ... };
};
UVM_COMPONENT_REGISTER(parent_component_proxy)
```

The implementation in the UVM-SV adapter is as follows:

```
class parent_proxy extends uvm_component;
    uvm_component child_list[$];
    uvm_component m_proxy;
    int           m_numChildren; // Number of Child Junction Nodes
    bit           i;
    string        component_type_name;
    string        instance_name;
    string        parent_full_name;
    function new(string name, uvm_component parent);
      super.new(name, parent);
      i = get_config_string("ml_uvm_type_name",   component_type_name);
      i = get_config_string("ml_uvm_inst_name",   instance_name);
      i = get_config_string("ml_uvm_parent_name", parent_full_name);
      m_proxy = factory.create_component_by_name(component_type_name, "",
instance_name, this);
      child_list.push_back(m_proxy);
      m_numChildren = 1;
    endfunction // new
    function void add_node(int proxyId,
                           string component_type_name,
                           string instance_name,
                           string parent_full_name);
       ...
    endfunction // add_node
    `uvm_component_utils(parent_proxy)
endclass
```