UVM-ML Whitepaper

A Modular Approach for Integrating Verification Frameworks

Gabi Leshem, Cadence Design Systems, Inc.
Vitaly Yankelevich, Cadence Design Systems, Inc.

Bryan Sniderman, Advanced Micro Devices, Inc.

12 March, 2013

Notices:

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

Contents

Table of Figures

Terminology

This section presents terminology and concepts that are referenced in this paper.

General Terminology

Framework—An assembly of verification and modeling facilities implemented in a single language. Some example frameworks are UVM-SV, UVM-e, OSCI SystemC, VMM, and so on. Different frameworks can be implemented in the same language (for example, UVM and VMM). Or, a new framework can be created as a composition of a few simpler frameworks (for example, a combination of OSCI with a SystemC UVM library). A framework may be written in a specialized Hardware Verification Language (HVL) such as SystemVerilog, or ***e***, a modeling language (such as SystemC), or a generic programming language (for example, C++).

Backplane—A routing and service layer for interconnecting frameworks by way of framework adapters. The Backplane is only required when multiple frameworks need to inter-operate. The Backplane is framework independent.

Framework Adapters—Form the adaptation layer between the Framework and the Backplane. It provides the API, which is required from each Framework in order to connect to the Backplane, thus allowing the Framework internal code (and user code running on it) to be unaware of the Backplane and UVM-ML requirements. The adapter is the mechanism for abstracting UVM-ML from its associated Framework. See the following figure for an illustration of framework adapters.

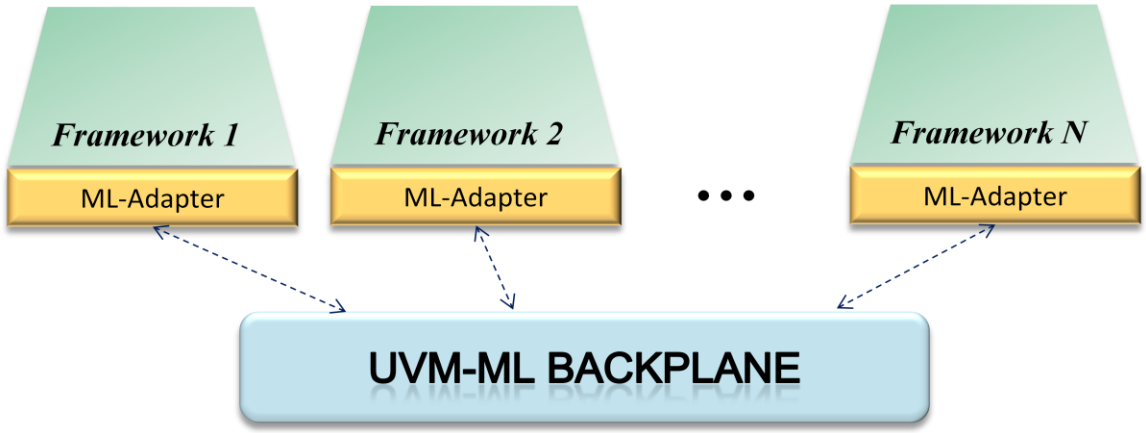


Figure 1: High Level Architecture and Main Entities

Verification IP (VIP)—A re-usable verification component built upon framework facilities. Examples include: monitors, scoreboards, drivers, UVC flavors, libraries, sequences, and so on.

Testbench (TB)—An arrangement of one or more verification IP components (VIPs) used for the purpose of supporting verification goals and strategies. VIP is usually targeted at the verification of a specific device under test (DUT) design. A testbench may be deployed on one or more frameworks, utilizing one or more languages, to achieve the verification goals. The testbench is also known as the *verification environment*.

Multi-Language Specific Terminology

UVM-ML compliant framework—A framework that provides the high-level functionality required by the UVM-ML specification, and meets the standardized UVM-ML interface requirements.

UVM-ML backplane —A supplementary language-neutral library that enables a seamless integration, and facilitates the interoperability, of multiple frameworks.

Mixed-Language call-chain—A sequence of nested procedure calls crossing language boundaries. A call-chain may include blocking & non-blocking procedures, according to programming languages rules.

Atomic Verification IP (VIP) —A single unit verification component that is manifested in only one framework.

Composite Verification IP (VIP) —A VIP composed of two or more atomic VIP parts that utilize one or more frameworks. A Multi-Language Composite VIP utilizes multiple frameworks.

Quasi-Static Framework Component —A VIP user-defined structural unit, derived from a framework component base class (for example, `uvm_component` in UVM SV or `unit` in ***e***). The difference between static and quasi-static components is that the former (for example, Verilog modules) are instantiated before the beginning of the test, while the latter are instantiated after. The time window, when creation of quasi-static components is allowed, may be limited by the framework phasing rules.

Synchronized Build Systems—A system with the build order reflecting the logical parent-child relationship between components across multiple frameworks (top-down, depth first).

Non-synchronized Build Systems —Systems in which the build order between frameworks is not synchronized for each individual build phase (rather all components of one framework are built before another framework's components).

Actor Terminology

Actor —A term that represents the idea of a *role*. The following is a list of actors who are pertinent to establishing a verification testbench. It should be noted that one or more persons may represent one or more actors:

> ***Developer*** —A person who develops an object in a single underlying framework.
>
> ***Integrator***—A person who, at any level, integrates developed collateral from one or more frameworks, and is exposed to multiple frameworks and their integration facilities.
>
> ***User*** —A person who is using the developed and integrated collateral to attain their goals, and is unaware of the composition of the facilities itself.

Actor Examples

Framework Developer —A person who develops one or more framework(s). Each framework is based on a specific choice of language and methodology.

ML Adapter Developer—A person who develops the adapter that enables the framework to interoperate seamlessly with the backplane.

VIP Developer—A person who develops either atomic or composite VIPs.

Composite VIP Integrator —A person who combines VIPs in different frameworks to form composite VIPs.

TB Integrator —A person who composes a testbench from atomic or composite VIPs. The frameworks and backplane interconnectivity is enabled by way of the ML Adapter Interface.

TB User —The person who is using the testbench to meet their needs.

Abstract

This paper explains the need for a solution to the problem of seamlessly integrating verification collateral that was developed using different systems and languages. The paper presents the issues and challenges that need to be overcome when enabling disparate frameworks to operate together. The paper also describes a solution that overcomes these multi-framework issues.

The paper summarizes the features that are available in the Universal Verification Methodology-Multi Language (UVM-ML), which is our released solution to the multi-framework problem. UVM-ML provides an integration solution for seamlessly interconnecting frameworks, and addresses the challenges users face when they need to mix VIPs from different frameworks together in one environment.

The paper concludes with a section describing future enhancements to the solution that will be made available in the coming months following the presentation of this paper.

Introduction

Modern testbenches are based on comprehensive methodology standards, such as UVM for SystemVerilog, or UVM for ***e***, which provide the key facilities to enable the verification of complex designs. While these methodologies are rich and fully featured, their implemented frameworks work best if all the testbench elements are available in their target framework and implementation language. For example, Verification IP (VIP) in the industry is written in several languages and methodologies. Fortunately, the industry at large has started to consolidate around methodologies such as UVM and Transaction-Level Modeling (TLM), which in the long term will help promote re-use and interoperability. However, this needs to be taken a step further by extending consolidated methodologies beyond their native language implementations.

In reality, testbenches are most often composed of a mixture of verification collateral from different frameworks and languages, such as C/C++, SystemC, SystemVerilog, ***e*** and others. For example, high-level design models written in SystemC may be re-used in a UVM based testbench as a reference model in scoreboards. Until now, verification deployment teams had to devise their own methods of incorporating disparate VIP from different frameworks and languages into a testbench, and aligning their methodologies.

These ad hoc solutions are proprietary, time consuming to implement by a few local experts, and may require additional attention as new forms of foreign collateral is introduced. The types of challenges embodied in multi-framework testbenches include, but are not limited to:

- Specifying the construction order of multi-framework components
- Referencing and configuring multi-framework components
- Connecting component ports between frameworks

- Transmitting transactions and other data between multi-framework components
- Specifying equivalent transaction data types between multiple frameworks
- Coordinating and synchronizing multi-framework component activity
- Coordinating reset and other key phases across multi-framework components
- Reporting errors and messages between frameworks
- Initializing the system, and enabling shutdown coordination across frameworks

In addition, many companies need to integrate and reuse verification code written by diverse teams inside the company, using different languages and methodologies.

These are just some of the difficult challenges faced by verification teams who are charged with overcoming multi-framework integration issues.

As a result, there is a growing industry-wide need to provide a standard, well conceived solution for efficiently enabling verification deployment teams to construct multi-framework based testbenches. This industry-wide need also includes the requirement for a solution that enables teams to focus on the verification task at hand, rather than spending time devising proprietary methods to incorporate multi-framework components.

A standard solution will further enable verification engineers to select verification collateral based on its merits and quality, rather than its availability in a given language or framework. This will serve to remove any barriers to developing high quality verification strategies, and promote higher re-use and interoperability between frameworks, while shortening testbench development cycles.

In that spirit, this paper introduces Universal Verification Methodology-Multi Language, a multi-framework solution known as UVM-ML, which is a solution that addresses the verification challenges in a manner that is comprehensive, flexible, extensible, and aligned with standard methodology concepts.

UVM-ML is so named because it is aligned to, and supports the primary concepts described in the UVM methodology, but is not limited to SystemVerilog. In fact, it is deliberately architected to enable connecting any framework that is able to provide the key UVM concepts. This is the powerful nature of the solution that is described in this paper.

Guiding Principles

There were several guiding principles followed in architecting the UVM-ML solution, including:

1. *Preserve the users' native-language experience.*
   It is important in order to increase the portability of VIPs for multi-language environments. This ensures that:
   - VIPs can be reused as is and without modifications with other frameworks.
   - Seamless interoperability of components in different frameworks is preserved.

2. *Keep consistency and alignment across frameworks and languages.*
   These are necessary to:
   - Ensure ease of use, comprehensibility, and maintainability.
   - Provide a single abstracted methodology that can be implemented in multiple frameworks, ensuring alignment in concepts, terminology, features, and APIs.

3. *Emphasize modularity of the solution.*
   Modularity simplifies implementation and maintenance, and enables extensibility. In accordance with this principle:
   - The backplane must allow connecting any number of frameworks.
   - Single frameworks must remain independent and self-sufficient.
   - The solution must not be dependent on the presence of any specific framework.

4. *Maintain openness and interoperability.*
   The integration of different frameworks, present and future, is realized by:
   - Providing a standardized API to help align the various framework adapters to a common platform.
   - Enabling participation of any UVM-ML capable framework so long as it provides a compliant adapter.

UVM-ML Architecture Overview

The UVM-ML solution is based on a central backplane library connecting two or more frameworks in a star topology. The *backplane* acts as the server transmitting the messages, and the *frameworks* are the clients. The backplane is not necessary in the single-framework environment, or if the frameworks do not communicate.

Each framework is connected to the backplane using the same procedural interface (API). This architecture does not rely on the services of any specific framework, and allows

connecting together any arbitrary number of frameworks. It also allows the broadcasting of messages received from one framework to other participating frameworks.

The general architectural layout is represented in the following diagram.

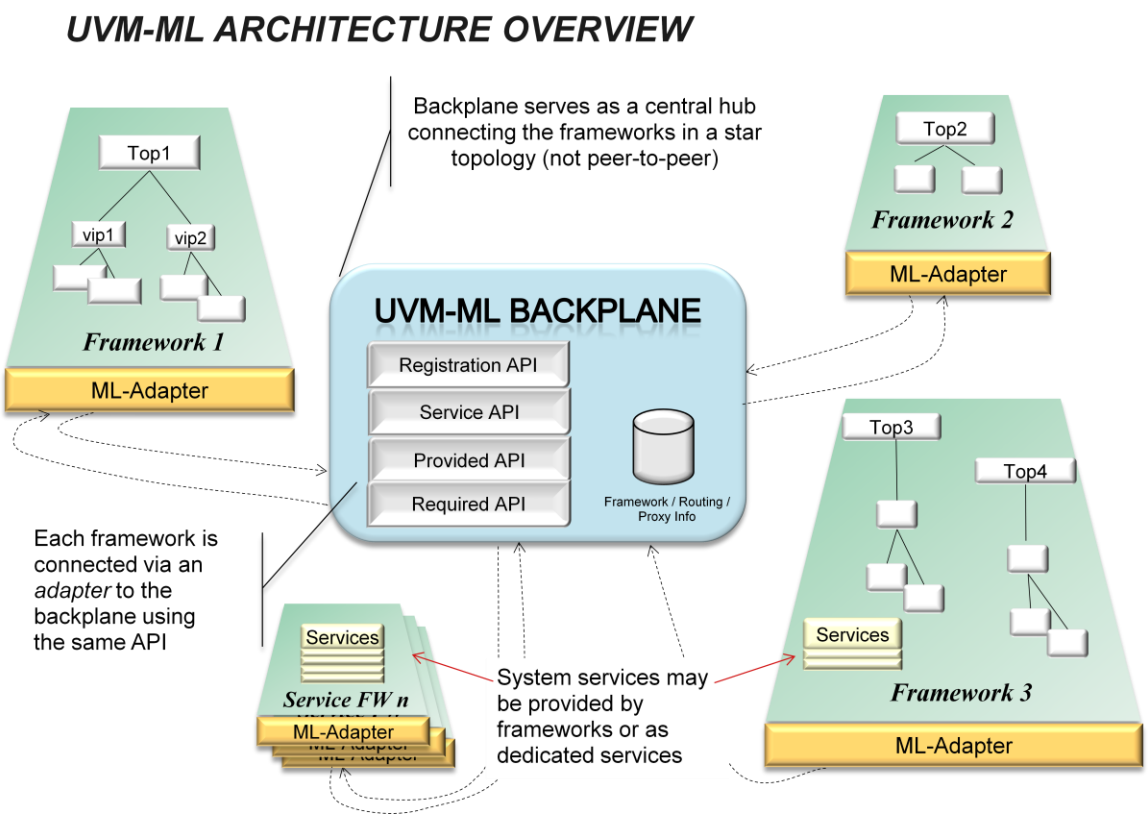


Figure 2: Multi-Framework Architecture Overview

The illustrated architecture creates an extensible solution that enables integrators to interconnect VIP from any framework that can subscribe to the UVM-ML backplane. The backplane itself is generic, and is not tied to any specific framework or vendor in its implementation. The generic API abstracts framework-specific details from the backplane.

The primary concept is that, while each local framework may support a complete or subset of different methodologies, implemented in different ways, they can still participate in connecting to other frameworks by way of the UVM-ML Backplane.

Standardized Parts of the Solution and Key Concepts

In order to make the solution interoperable for different frameworks and simulation engines, the following parts must be standardized:

- Backplane API
- Data transfer protocol
- System initialization procedure

The Backplane API externalizes the key UVM concepts and facilities, including:

- Phased pre-run, runtime, and post-run execution of the corresponding states
- TLM-based communication
- Hierarchical relationships between structural building blocks (components)
- Configuration of components and sharing of the resources
- Coordinated messaging and error handling
- Runtime synchronization between the components

The concepts listed above are generic and should be applicable for different local methodologies and language-specific implementations without enforcing detailed low-level alignment between them. This is useful for supporting disparate frameworks, with varied representations of the key techniques that need to operate together.

For the standardized parts, the data transfer protocol must be standardized in order to enable the deciphering of transactions produced in any other framework. The backplane does not perform any data manipulations and simply re-directs the transactions.

The standardized initialization procedure must be established in order to specify operations such as: *how and when the backplane library will be loaded*, *how the framework adapters will be registered*, and so on.

As a rule, there may be only one instance of the backplane in any integrated multi-framework environment. Consequently, it is important that there be only one open-source implementation of the backplane. Otherwise, it will be impossible to connect frameworks developed by different vendors, or users, if each of them arrives with a proprietary implementation of the backplane.

The native framework facilities do not include a component that connects to the ML Backplane API. Such a component must be added per framework. We call this a *Framework ML Adapter*. The adapter serves as a bridge between the native framework facilities and the ML Backplane.

In this way, VIP in each framework can continue behaving according to how its framework provides the facility. For example, if a framework has its own proprietary method for transaction communication, then its ML Framework adapter would need to translate between the native framework's method for transaction passing and the ML Backplane's TLM oriented APIs. The level of complexity in each framework adapter depends on its framework's abilities. Frameworks conceptually aligned to the UVM methodology may have less logic in their adapters as compared to frameworks which have native semantics different to UVM.

Another degree of freedom is with respect to the *framework capabilities*. Frameworks that are connected to the backplane can specify their capabilities at registration time in order to indicate what activities they are able to subscribe to in the system. For example, the native SystemVerilog UVM can serve only as the master phasing controller, and cannot allow another framework to control its phases.

The notion of capabilities allows frameworks to participate without requiring each framework to be augmented in order to be ML capable. This enables legacy and proprietary frameworks used by the VIP to participate in the system in a limited way. The integrator's role is to be aware of the capability of these types of frameworks and ensure that participating framework VIPs are deployed within their means. A Framework ML adapter can issue an error message if the framework is required to execute some activity that it is not capable of performing.

UVM-ML Backplane API

The Backplane API covers behavioral aspects of interoperability. Communication between a framework and the backplane is bidirectional. One framework can send a transaction, initiate a phase, and so on, while another framework (or frameworks) can process that transaction, or execute the phase. Consequently, the backplane API consists of two parts:

1. The part that contains the functions that the backplane implements, so that a framework can call them. This is called *the backplane provided API*.

2. The part that defines the functions that a framework adapter must implement in order to support each feature. This part is called *the backplane required API*. If a framework is not capable of supporting a certain feature, it can assign *NULL* to the corresponding function pointers.

The totality of the provided and required API functions is called the *Backplane API*.

The Backplane API is based on recognition of the currently available technology limitations in the multi-language programming area. The most important limitation is the absence of an ML object-oriented programming framework for the platforms and languages used in EDA. Technologies that enable passing objects by reference across framework boundaries, invocation of the object methods, or common memory management are not standardized yet. Consequently, the Backplane API is designed as a pragmatic alternative to the lack of the native ML programming solution.

The backplane library is implemented in the C++ language, but the generic API programming language is C. Because the backplane communicates with the frameworks implemented in languages other than C++, C is the natural and most commonly supported choice for language interoperability. As such, the C++ backplane provides a C API wrapper layer.

Since the backplane is implemented as an open source, the framework adapters, which are generally based in C++, can take advantage and directly interact with the C++ backplane implementation layer, bypassing the generic C layer.

The provided and required APIs are two sides of the same collection of the feature-oriented interfaces (clusters of functions), covering the following functionality:

- **Setup** (initialization and registration)
  This part of the API enables connecting the frameworks adapters to the backplane, specifying their capabilities, and identifying a test and top-most hierarchical components. In addition, the selection of ML services, such as master phase controller, message, and error handling are specified during setup.

- **TLM**
  This part defines how the TLM2 sockets or TLM1 ports, belonging to different frameworks, are connected and how the standard TLM interface calls are propagated across language boundaries.

- **ML Phasing**
  This API supports a synchronized phasing notification flow from the phase controller to the rest of the integrated frameworks. There are two kinds of phasing interfaces: framework notification, and the hierarchical component phase notification.

- **ML Hierarchical Composition**
  This interface enables instantiation of a quasi-static hierarchical component in a foreign framework, and could support forwarding of the predefined component

interface functions (such as `print`).

- **Synchronization**
  This interface allows frameworks to explicitly synchronize their simulation times (for example, as an implementation of a time quantum mechanism). This API also supports other synchronization facilities (for example, events, objections, or barriers).

- **Configuration and Resources**
  This interface (yet to be added to the currently available backplane API) will allow propagation of configuration and other resource settings across framework boundaries.

Backplane Library

The backplane is distributed as open source. A distribution such as this enables a wider choice of platforms by preventing the backplane from being bound to specific OS platforms and C++ compiler versions, leaving the choice to the integrator.

One of the integrator's roles is to build and link the backplane. The most useful way of linking the backplane to the executable format is by the creation of a dynamically loadable (shared) library. Every popular simulator supports an ability to link in a dynamic library. Hence, it will be a straightforward task to add the backplane without requiring any built-in support in the simulator when multiple frameworks are used.

In order to avoid potential linking problems between the framework adapters and the backplane (which may, or may not, be included), the Backplane API is available in the form of an array of C function pointers, called the *provided API function tray*. Given the function pointers, the framework adapters do not need to depend on the hard-coded symbols that must be resolved at linking time. In other words, frameworks can register with the backplane at runtime. The only pre-requisite for using this mechanism is that the adapter obtains the API function tray itself during setup. The backplane API provides a special function with the fixed name that returns the tray. The adapters can obtain this function pointer using the dynamic symbol finding function of the standard C library.

UVM-ML Framework Adapter

The framework specific implementation for the ML Backplane *required* APIs and services are deployed by each UVM-ML Framework Adapter. There is only one adapter per framework. These adapters translate between the framework's implementation and the backplane to allow seamless connectivity between framework native facilities and the UVM-ML Backplane's APIs.

Keeping the separation between the framework and its UVM-ML adapter enables a modular and flexible architecture where a standardized framework is not immediately affected by the introduction of the ML layer or changing ML requirements.

The adapter architecture and implementation is not standardized. The only mandatory requirement is that each adapter shall be able to register itself with the backplane, at the initialization stage, and provide the backplane with its required API function tray. It is not required that the adapter implement all the API functions. The *NULL* function pointers assigned to the tray elements guarantee that the backplane will not attempt to call non-implemented functions.

The following diagram shows a sample adapter structure.

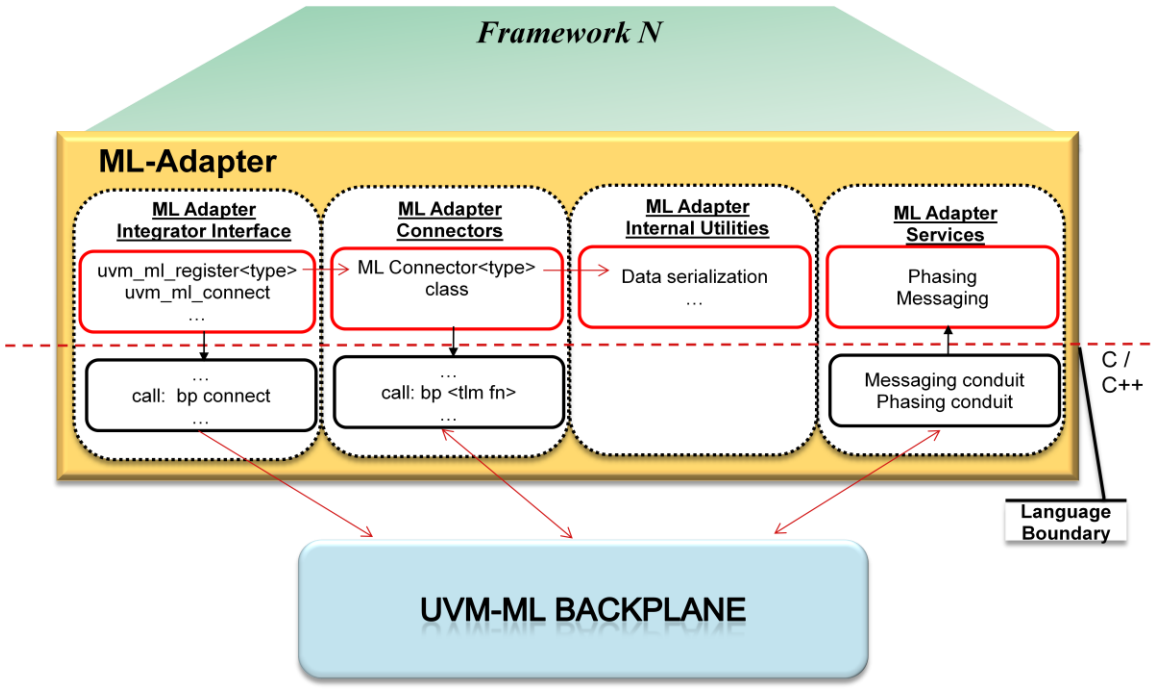


Figure 3: The Framework Adapter Architecture

In the sample adapter structure diagram, the following terminology is used:

- UVM Multi-Language Adapter Library (ML Adapter)
  - This is the full package containing all the facilities required to bind a framework to the backplane, including the integrator's user interface, implementation API interface, internal connectors, framework services, utilities, and any cross-language glue code.
  - The adapter itself may consist of two layers, implemented in two languages, as shown in the diagram above. These are: the native framework language part, and the C/C++ layer. For example, the SystemVerilog DPI-C interface requires the addition of an extra C layer, in addition to the generic backplane code, because of the need to set a scope of the export DPI function before invocation. If the framework is in C/C++ then a language boundary crossing layer is not required.

- ML Integrator API
  - This is the user-facing adapter interface implemented in the framework's native language.
  - This interface covers the functionality that requires explicit specification of the foreign language port's path, component foreign type name, foreign type name for polymorphic transaction types, and so on.

- ML Adapter Connector
  - This is an object that is responsible for connecting an item in the framework to the backplane in order to forward the item's data to the backplane for routing.
  - The integrator API should create the connectors implicitly so that the connectors are transparent (or almost transparent) to the end user.
  - There may be an instance of a connector per item connected.
  - Example connectors are: ports, sockets, pools, and logical hierarchy points.

- ML Adapter Services
  - There are optional services that can be registered to the backplane service callbacks. For example, a framework can provide a service of a master phase controller for the rest of the frameworks.

- ML Adapter Internal Utilities
  - Utilities (such as *pack* and *unpack*) that are required for the connector and services.

UVM-ML Key Concepts

Logical Hierarchy

In a single framework testbench, all the VIP collateral is constructed in a hierarchical manner, from the top down, recursively, within the same framework. In the UVM methodology, the parent-child hierarchy relationship is required to enforce a predictable build and configuration order.

Typically, a single framework has only one framework root node that spawns the construction of all its child nodes. Hence, the hierarchy of a single framework system can always trace its origins back to its root node.

In the multi-framework scenario, VIPs exist in more than one framework. When the integrator wants to mix VIPs from different frameworks together, then conceptually, from a user point of view, the set of VIPs should manifest themselves as though they belong to a single framework. This means that regardless of in which framework the VIP may reside, to the user, it should appear as though the VIP is participating in the same hierarchy as it would in the single framework scenario.

The following diagram illustrates the relationship between a single framework hierarchy and a multi-framework hierarchy.

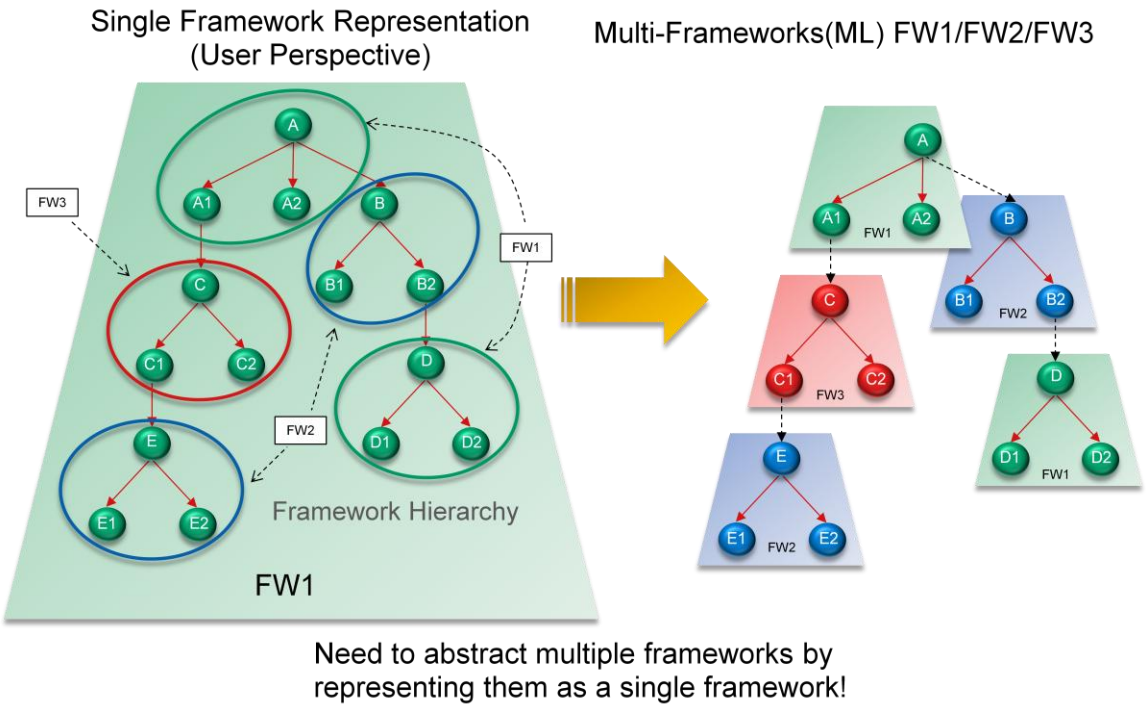


Figure 4: Logical Hierarchy: in one vs. many Frameworks

In the diagram, the left side represents a single framework (*FW1* is represented in green), where all of the nodes, starting from the root node *A*, are related in a parent-child relationship, according to their construction hierarchy.

In reality, the VIPs may be manifested in different frameworks, depicted on the right side of the diagram, where nodes *A* and *D*, reside in *FW1*, nodes *B* and *E* reside in *FW2*, and node *C* resides in *FW3*.

While these VIP components may effectively be in different frameworks, their hierarchical relationship needs to be maintained in order to effectively appear as though they are participating in the same framework. As such, the right side of the diagram can be thought of as the *Logical Hierarchy* representation of the multi-framework system.

It should be noted that the dotted lines between nodes spanning different frameworks represent their logical association. The details of how VIPs are interconnected across frameworks are abstracted by the backplane and its adapters.

The concept of a *Logical Hierarchy* essentially provides the system with a way of representing multi-framework VIPs as a *Single Unified Hierarchy*, as illustrated on the left side of the diagram. This is an important concept that enables all users to reference components in the system in the same way, regardless of where they physically reside. It allows the integrator to substitute for VIP implementation in different frameworks without requiring testbench changes, so long as the VIP implementations are functionally and hierarchically equivalent.

Tree Terminology

The terminology listed in the ML Terminology column of the following illustration is relevant to the concept of *Logical Hierarchy*.

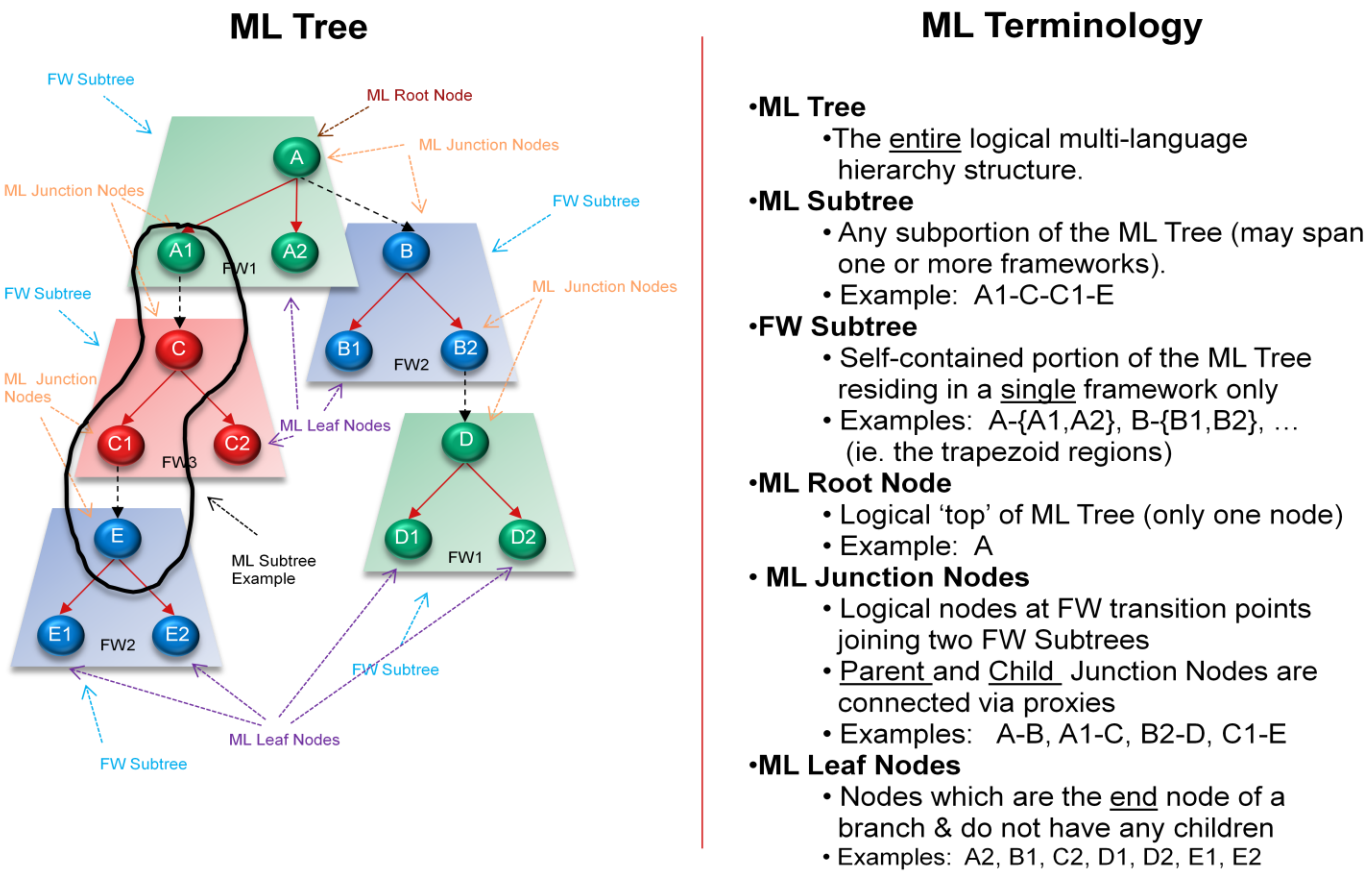


Figure 5: Logical Hierarchy: Tree Terminology

Logical and Physical Representations

The difference between the *Logical Hierarchy* and its *Physical Representation* are illustrated in the following diagram.

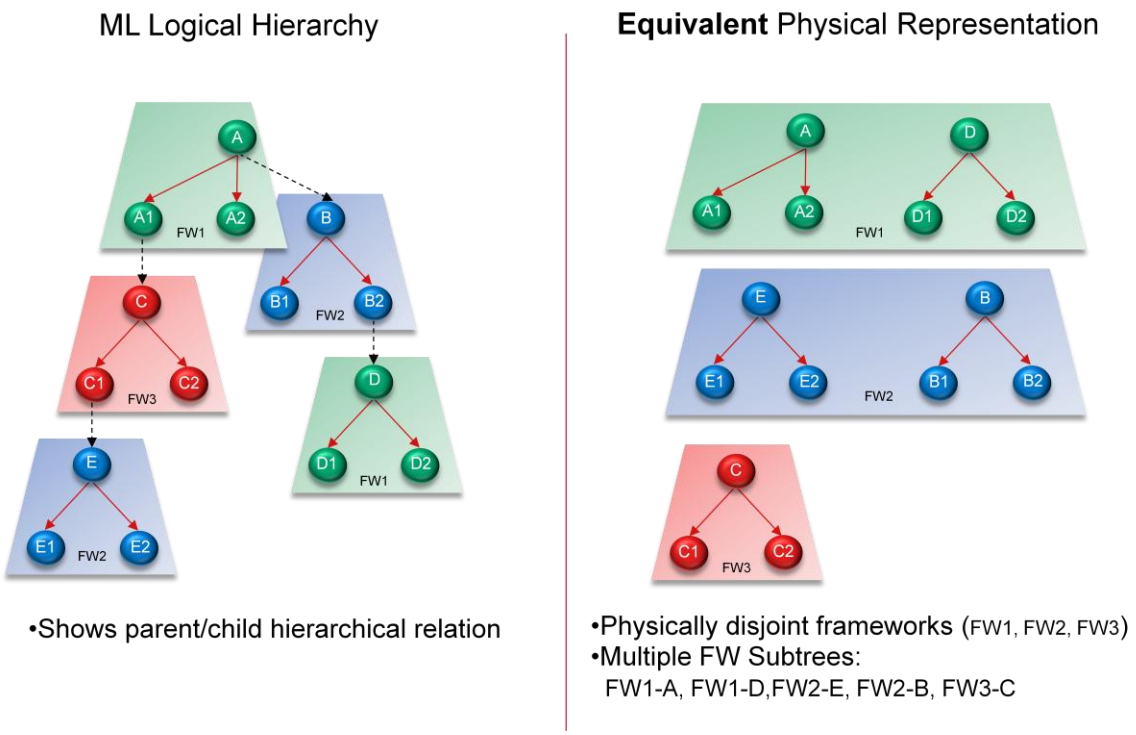


Figure 6: Logical Hierarchy and its Physical Representation

The left side of the illustration shows the logical relationship of the nodes, as indicated in previous diagram. Logically, the nodes appear as though they participate in a *Single Unified Hierarchy*. However, they actually reside in different frameworks.

The right side of the diagram depicts the actual physical representation of the system.

In the physical representation, some of the frameworks (*FW1*, *FW2*) contain multiple sub-trees, which are physically disjointed from one another. Each sub-tree actually represents a portion of the logical hierarchy. The relationship between the various sub-trees is dictated by the logical hierarchy. As such, a single framework needs to manifest more than one independent sub-tree at a time, in order to support any arbitrary logical hierarchy specification. This is an important concept that extends the simple scenario of a single framework that assumes it only needs one root node to support the construction of all its children.

Multi-entry Frameworks (multi-top)

In addition to needing to support more than one concurrent sub-tree in the system, participating frameworks need to be re-entrant. Since the construction of sub-trees follow the logical order of construction, a framework may be entered more than once at disjointed times during the constructions. The same is true for any other phases that follow the logical hierarchy, where nodes are traversed according to the system's logical hierarchy, and hence may result in a framework being visited more than once. This is known as the *Multi-Entry Framework* concept.

Multi-entry Frameworks are frameworks that are capable of supporting more than one sub-tree and are able to be entered more than once during the same phase. Upon each entry only the specified sub-tree is notified for the current phase.

In some scenarios, there are cases where multiple independent logical hierarchies are desirable. This means that more than one root node is specified at load time, and they can co-exist independent of each other in the system, though each of them may still have multiple sub-trees in more than one framework. This allows for concurrent independent systems to be constructed which may only be bound to each other through non-hierarchical methods (for example, TLM communication, synchronization facilities, and so on).

Multiple Logical Hierarchies refers to the ability to specify one or more disjoint ML Root Nodes that spawn independent logical hierarchies in one or more frameworks. This would be manifested in systems where loading independent ML Root Nodes is required (for example, in systems that require independent local hierarchies, such as the separation of environment and test hierarchies).

Framework Capabilities

Frameworks may have different capabilities and may be based in different methodologies. Although the spirit of the UVM-ML solution is to provide a conduit for coordinating the core UVM methodology across multiple participating frameworks, it is recognized that there are frameworks that preceded the advent of UVM and may only possess a subset of the UVM methodology, and so, do not have the same capabilities in all areas.

As users may have significant collateral based on older or legacy framework methodologies, it is not realistic to expect that all of the participating frameworks will be upgraded to subscribe to the complete set of UVM methodologies. Therefore, different frameworks may each have a different subset of capabilities.

As such, in order to accommodate the incorporation of frameworks with different capabilities, the UVM-ML solution provides a method for frameworks to report their capabilities at initialization time. This allows the integrator to identify which frameworks may participate in which aspects of UVM-ML eco-system. For example, if one framework does not have a notion of *resources* or *configuration*, it can report that it does not support this feature. This means that this particular framework would not participate in resource sharing and hierarchical configuration, while other frameworks, which subscribe to this capability, would participate. It would also mean that the integrator is aware of any system limitations and may have to consider how to address any areas which may require some form of participation.

Without this concept, frameworks that are not able to provide all the equivalent UVM capabilities would not be able to participate in the UVM-ML system. It should be noted, that indicating that a framework is capable of a particular UVM feature, does not imply that the framework manifests the feature in the same way as other UVM capable frameworks. In fact, each framework implementation may be quite different from each other. This is where the adapters act to both normalize and abstract the way their associated frameworks manifests each capability, with respect to the Backplane. In this way, frameworks which report they are capable of manifesting various UVM features but have completely different implementations can still contribute to the UVM-ML system.

UVM-ML Features Overview

The following areas are addressed by the UVM-ML solution:

- Overall UVM-ML Flow: From Initialization to Shutdown
- Hierarchical Construction
- Resource and Configuration
- TLM Communication
- Pre-/Post-/Run Phase Synchronization
- Message Reporting and Control
- Error Handling and Reporting
- Time Management

The following sections explain these areas of the UVM-ML solution.

Overall UVM-ML Flow: From Initialization to Shutdown

In order to integrate frameworks, the integrator will include the backplane shared library and the relevant framework ML adapters in the simulation environment. The mechanism of bringing in the adapter is framework- and implementation-specific. The example UVM SystemVerilog ML adapter, which is incorporated in the UVM-ML package, is implemented as a combination of a SystemVerilog source package *uvm_ml* and "DPI" C code. Hence, the integrator must import the package in the appropriate place in the SystemVerilog source code.

Another example adapter in the UVM-ML package is the UVM SystemC ML adapter. It is written in SystemC. The integrator will first compile it and probably link it together with the user code. In order to use the SystemC adapter API the integrator needs to include the *uvm_ml.h* header file.

Once an ML adapter is added in the simulation environment, it automatically registers itself at the backplane library. This registration mechanism is internal to the adapter/backplane communication and is transparent to the integrator. Essentially, the

adapter developer is responsible for invoking the backplane function that returns the provided API tray, and for calling the framework registration function, while passing the framework-specific information as the arguments.

The framework-specific information includes the unique framework adapter name and the so-called framework indicator names. The unique adapter name is used by the backplane only for the tracing and messaging purposes. The framework indicators are needed in order to identify top and ML junction hierarchical components.

The diagram below shows an example of using the framework indicators as the arguments to the SystemVerilog ML test initialization function:

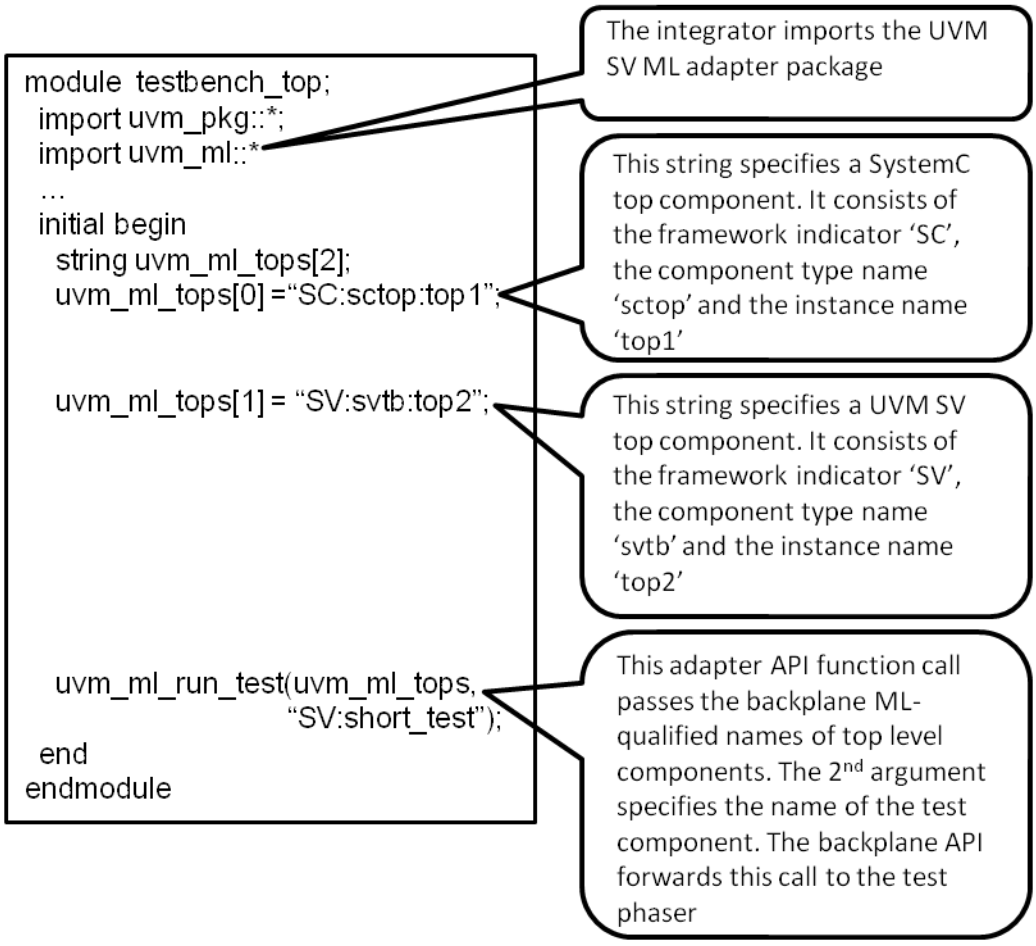
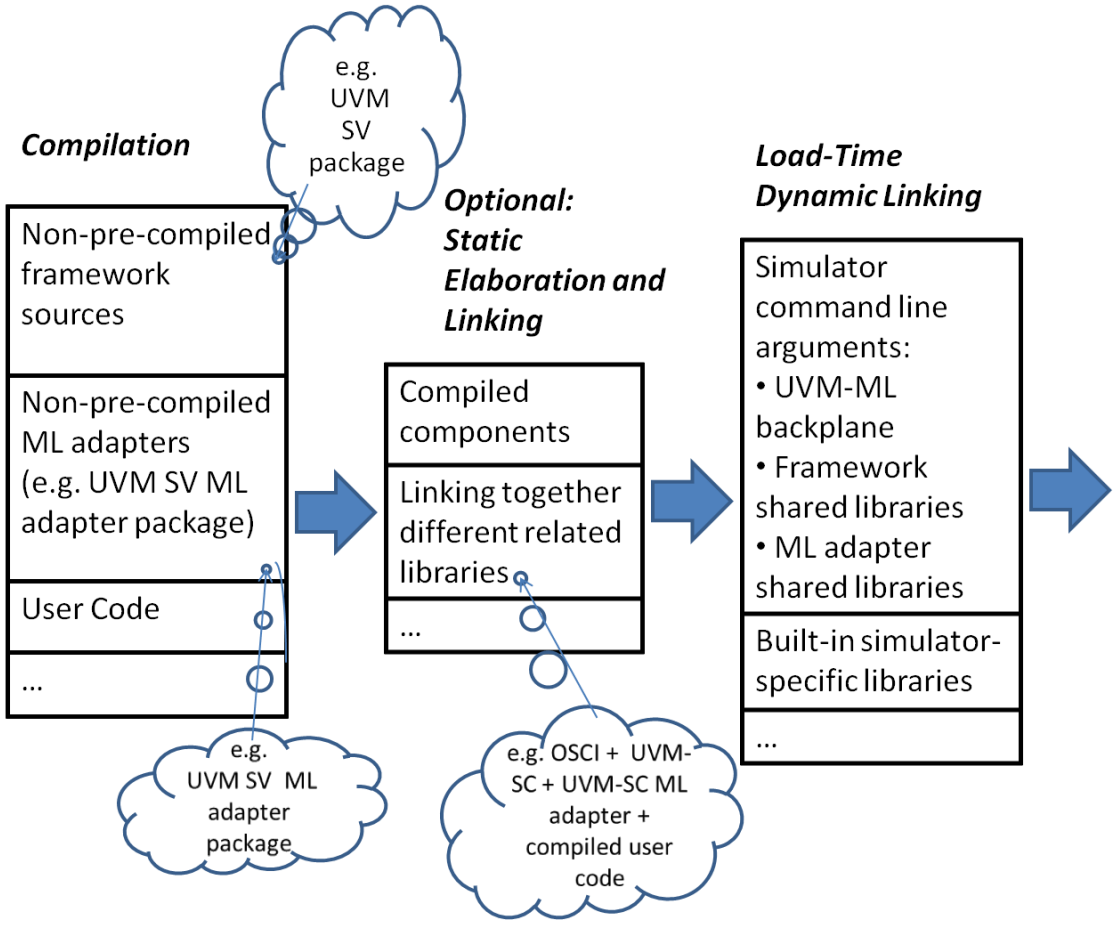


Figure 7: Sample SV Initialization Implementation

One adapter may register several framework indicators. For example, in the above diagram, the integrator uses short indicators 'SC' and 'SV' in order to specify the frameworks. Such short indicators are sufficient when the integrated environment contains one framework per language. Assuming that there may be a need to integrate more than one framework implemented in the same language, the integrator may be required to use longer, more detailed, indicators (for example, 'UVM-SV' rather than just 'SV', or 'UVM-SC' rather than just 'SC', and so on). The adapter documentation will specify in detail what indicators it registers. The backplane library issues an error if the integrator uses an ambiguous indicator (associated with two or more framework adapters).

The diagram below schematically shows main steps of the ML test flow from initialization until shutdown.



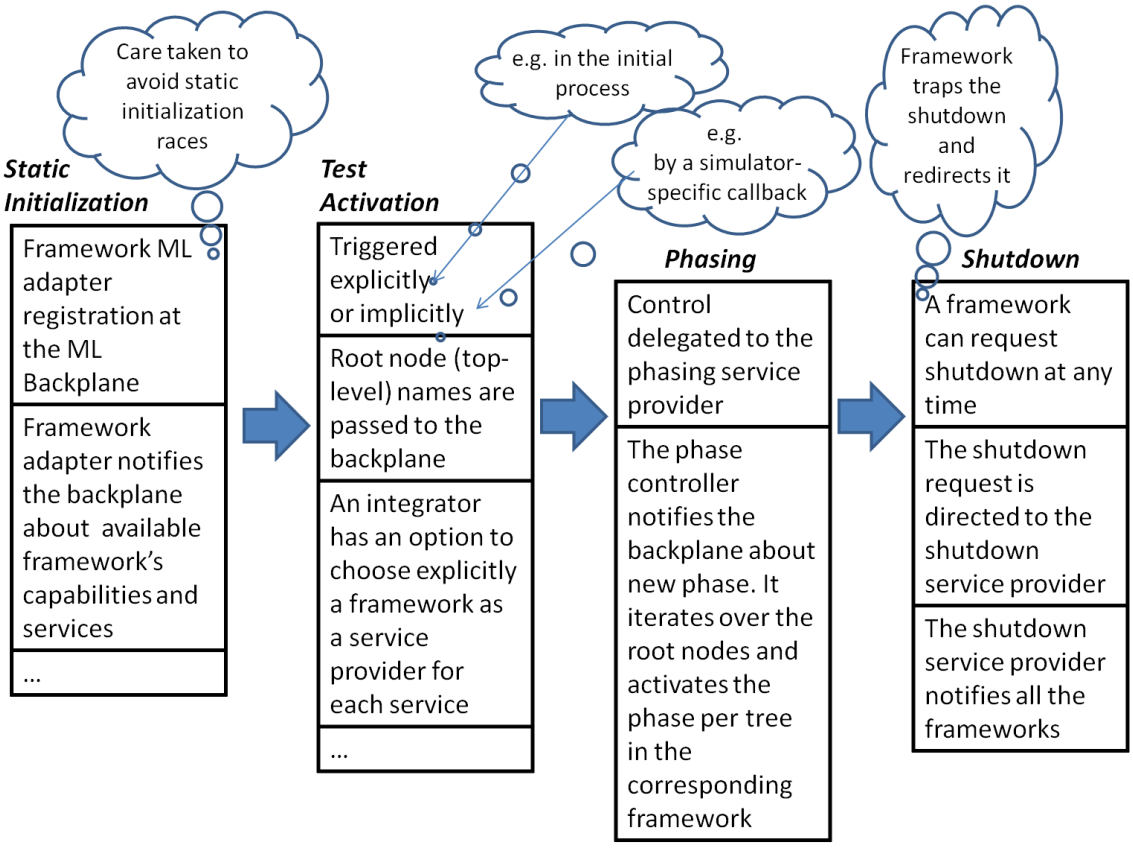


Figure 8: UVM-ML Flow from Initialization to Shutdown

The backplane API supports a centralized mechanism for shutting down the simulation. Note that the term *shutdown* is unrelated to the process of the synchronized phased completion of a test. The backplane provided API allows a framework to register itself as a potential shutdown service provider. Another provided API function allows a framework to request the shutdown. The backplane would re-direct shutdown requests to the service provider framework. The service provider notifies all frameworks about the shutdown request giving them an opportunity to execute the local shutdown. This feature will be supported in a future release of the backplane.

Hierarchical Construction

The need for logical hierarchy in ML environments was discussed the UVM-ML Key Concepts section. Logical hierarchy is made possible by hierarchical construction which ensures that the entire ML environment is constructed in a proper way. To make this possible, each framework should support piecewise construction, allowing construction of each sub-tree independently.

Hierarchical Proxies

To implement logical hierarchy, some special components are introduced to serve as bridges between the frameworks.

An **ML child proxy** is a special component that plays the role of the root of the sub-tree in a foreign framework (child junction node). A unique child proxy is needed to represent each root of a sub-tree.

Similarly, an **ML parent proxy** is a special component playing the role of the logical parent, representing some node in a foreign framework (parent junction node). As such, the logical path of the parent proxy is identical to the logical path of the foreign parent node, and the child junction node created under it will use it as its parent path.

The child proxy is being used both to create and connect to its corresponding parent proxy, which in turn creates the corresponding sub-tree. In addition, the child proxy propagates the phases to the parent proxy, thus maintaining proper order of phasing in an ML environment.

A single parent proxy can represent multiple sub-trees, so multiple child proxies may be connected to the same parent proxy. This is illustrated in the following diagram.

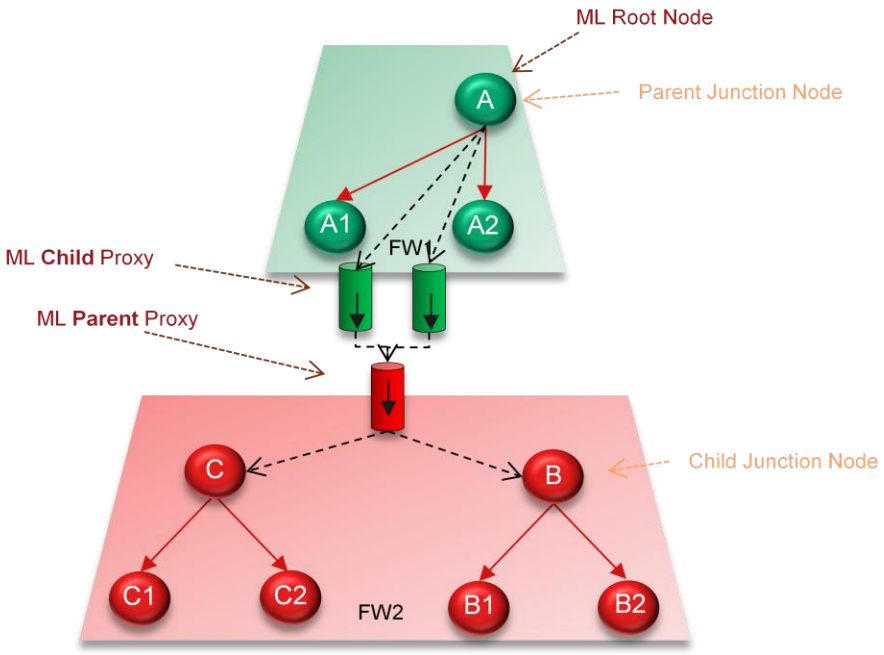


Figure 9: Proxies Enabling Logical Hierarchy

Resource and Configuration

Resource and Configuration Relevant Terminology

The following terminology and concepts are relevant to the UVM-ML solution:

Resource element—Refers to an item that specifies a set, such as *{scope, name, type, value, attributes}*, which is stored along with other resource elements in a storage container.

The *scope*, when combined with its *name* provides a unique index into the resource container. The *value* of the item represents its assigned data, which may be any object that can be stored or referenced (for example: integer, string, class, array, and so on). The resource names are not necessarily unique. The combination of the name and scope allows the specific resource value to be chosen, among the resources with the same name.

Configuration element—Is a resource element that is stored in a configuration container and has a *scope* that represents its hierarchy. Hierarchical scope is required for configuration items and is used for value resolution.

Hierarchical Configuration—Is the methodology for determining the value of a configuration element based on its hierarchy, and the hierarchy of its parental lineage. The highest hierarchical ancestor of a configuration element may override the provided value for the element. The resolved value, based on ancestral evaluation, would be the value returned as a result of accessing a configuration element.

Distributed Storage—Refers to storage for the resource or configuration elements that are distributed across different frameworks. Each framework will have a local copy of the elements for its corresponding table.

Centralized Storage—Refers to storage for resource or configuration elements that are stored in a single location for all frameworks. A single copy of all items for the system would reside in the common storage area.

UVM-ML Resource and Configuration Solution Overview

Given the similarity between resources and configurations and layering of configuration on top of the resource database in UVM, a common resource-centric solution will be provided. For the purpose of the following discussion, resources are only distinguished from configurations based on the type of *scope* (resources may have any scope, while the configuration's *scope* is always pre-pended with its hierarchy context).

The ML Hierarchical Configuration and Resource solution may be approached in one of two major ways:

1. Centralized Solution
2. Distributed Solution

For an ML Configuration in particular, either case requires that the system be constructed in a true ML Logical Hierarchical manner, such that the relative relationship of components in the system, from any framework, is governed by its logical hierarchical path in the ML System. This requirement is fundamental to ensuring the correct parent-child relationship of every component in the system, in order to enable configuring components according to the parent-child rules of configuration.

The Centralized Solution
The centralized solution revolves around storing each resource or configuration item declared in any framework into a centralized configuration storage facility owned by the backplane (either directly or through a backplane service). In this solution, each item registered from any framework is routed to the backplane and stored.

In addition, all read requests are also routed to the backplane for retrieval. This solution implies that all native framework configuration mechanisms need to route both their read and write requests to the backplane, and bypass their local storage facilities. Given read requests may outnumber write requests, this solution may introduce some performance inefficiencies. In addition, frameworks that are deployed in languages different from the backplane need to cross a language boundary for any resource or configuration access.

Due to potential language boundary crossings for both local and non-local accesses, the centralized storage solution is not able to support the use of reference data pointers in its storage. Frameworks which currently allow configuration items to contain reference data need to adapt their behavior and are not backwards compatible. A centralized solution could be suitable in a future scenario where participating frameworks are updated, or written to rely on the central resource and configuration facility. This scenario would enforce a consistent methodology for all participating frameworks and would eliminate potential inconsistencies. Such a solution however, would disallow storing data by reference. An efficient read mechanism for maximum performance would also be required in this scenario.

The Distributed Solution
Conversely, the distributed solution relies upon each framework manifesting its own concept of resource and configuration storage. A framework would interact with its native facility as it would in a single framework scenario. Only *create* or *update* activity would generate notifications to the backplane. In this scenario, once an item is created or updated in the local framework facility, a copy is propagated to the backplane for notification to other frameworks. Resource and configuration entries that store data object pointers could be permitted as long as such items can be copied and serialized for

broadcast. If object pointers cannot be copied and serialized, they can still be stored locally in a distributed solution, but not broadcast. Consuming frameworks would receive updates from the backplane and store the new item or updates locally. This distributed method would effectively ensure that all frameworks are synchronized with respect to copy-enabled resources and configurations. In this scenario, only *creates/updates* would need to propagate across the backplane. Given that each participating framework has a local copy of the item, all read operations can be performed locally and efficiently.

The main drawback of the distributed solution is that every item written or updated would need to be routed to every framework, regardless if it needs the item, or not. This could lead to performance issues during the configuration phase, in particular. In the centralized solution, each framework would only need to access the items they require. This would minimize the overall flow of resource and configuration notifications to frameworks. In this respect, the centralized solution provides better write efficiency.

On the other hand, it is expected that read operations will be pervasive among frameworks during both the pre-run and run phases, while write operations normally will typically occur in the build phase for configuration. It is also expected that there will normally be more reads than writes by multiple components per resource or configuration item. As such, a solution which optimizes read performance over write performance may be more desirable for overall system performance—in particular runtime performance. This would be an advantage for the distributed approach.

The distributed solution preserves the local framework's resource and configuration management, while the centralized solution requires that all frameworks follow a unified approach to configuration management. This allows legacy assumptions and behavior to continue locally, with minimal modifications required to support an ML system.

It should be noted that the centralized solution also eliminates per framework implementations of the configuration facility, but would require frameworks to always reference configuration items through the backplane, and potentially, by way of a language boundary crossing, for both reads and writes.

Both scenarios are desirable. However, one can view the distributed approach as being a less intrusive initial solution since it can bring together both new and legacy frameworks with little modification.

The centralized solution could be a better long term approach, by providing conformity and standardization of how resource configuration management is deployed and enforced. However, this solution would require re-architecture of existing frameworks to achieve this and require that at least the part of the backplane always be present for standalone frameworks.

The Preferred Solution

Given the near-term requirements to seamlessly support legacy frameworks, allow frameworks to operate standalone without requiring a backplane, and minimize the impact of instantiating a backplane in the system, the distributed approach was preferred.

The distributed solution includes:

1. For Hierarchical Configuration support, participating ML frameworks must also participate in the ML Logical Hierarchical build and construct their components according to the logical hierarchy.
2. Participating ML frameworks provide local resource and configuration storage:
    a. Configuration items provide a logical hierarchy, name, and value (data).
    b. Resource items provide a scope, name, and value (data)
        i. The provided scope may contain wildcards that are stored in the database as is (without expansion). As such, the wildcards in a scope are propagated across the backplane as is (unexpanded). Each local framework resource database is responsible for wildcard evaluation based on regular expressions upon local retrieval operations.
    c. All items must store associated attributes (for example, access, precedence, and so on).

    Notes:
    - A framework may opt to not participate in configuration if it does not require build configuration or is not capable of supporting configuration. This would reduce the number of frameworks that the backplane needs to route configuration notifications.
    - For resources, if a framework does not have the concept of resource storage it may also opt to not participate.
    - Any method of local storage may be supported as long as the Framework Adapter can use its *{scope, name}* set to determine its local target destination for the item.
3. Local ML framework resource and configuration facilities must provide a notification or callback mechanism in order to notify their respective adapters of create and update operations if the resource value or attributes change, or if it is operated upon.
4. ML Framework adapters may be capable of distinguishing copy enabled and serialize-able resource and configuration requests from objects unable to be serialized before propagating them to the backplane. Items which cannot be copied and serialized cannot be leveraged in remote frameworks and cannot be notified.
    a. Users may define the serialization and de-serialization method for a data type.
    b. By default, non-serializable items are not propagated and are not reported as a warning.
    c. If warning levels are enabled (based on verbosity settings):

        i. Sender warns if it cannot serialize an item.
        ii. Receiver warns if it cannot de-serialize an item.
    d. ML-safe usage of resources requires that any update to a resource flow through the resource database for resources targeted for multi-language usage.
5. ML Framework Adapters receive resource broadcasts from the backplane and translate the requests into newly created local resource items, or updates to existing local configuration items.
6. Additional broadcast filters in both the ML Framework Adapter and backplane can be added to reduce what is broadcast. For example, a framework may choose to subscribe to one or more resource scopes or unsubscribe from particular scopes.

TLM Communication

The UVM-ML solution provides a built-in support for the standardized TLM1 and TLM2 interfaces. With UVM-ML, it is possible to specify connections of various TLM ports or sockets in any of the integrated frameworks.

A native language connection function may not be useful for ML if it operates with direct references (such as pointers or handles) to the connected ports. Hence, a framework ML adapter, supporting TLM, must provide a special ML connectivity API. Each adapter API is expected to be framework-specific but it shall eventually communicate with other adapters by way of the pre-defined Backplane API.

In an integrated multi-framework environment you can monitor transactions in one framework, and broadcast the monitored transactions to other frameworks for checking or coverage collection (for example, using ML analysis ports).

The Backplane API allows connectivity of TLM ports or sockets identified by their native language names. The primary task of the ML solution is to enable integration of complex environments. Hence, the port naming scheme must scale to the task of extendible composition of multiple VIP's.

UVM already has a mechanism that assigns each TLM port a unique native hierarchical name. The UVM-ML solution embraces this scheme and supports TLM connectivity based on the ports' hierarchical names.

The diagram below demonstrates, for example, a SystemVerilog integrator's view.

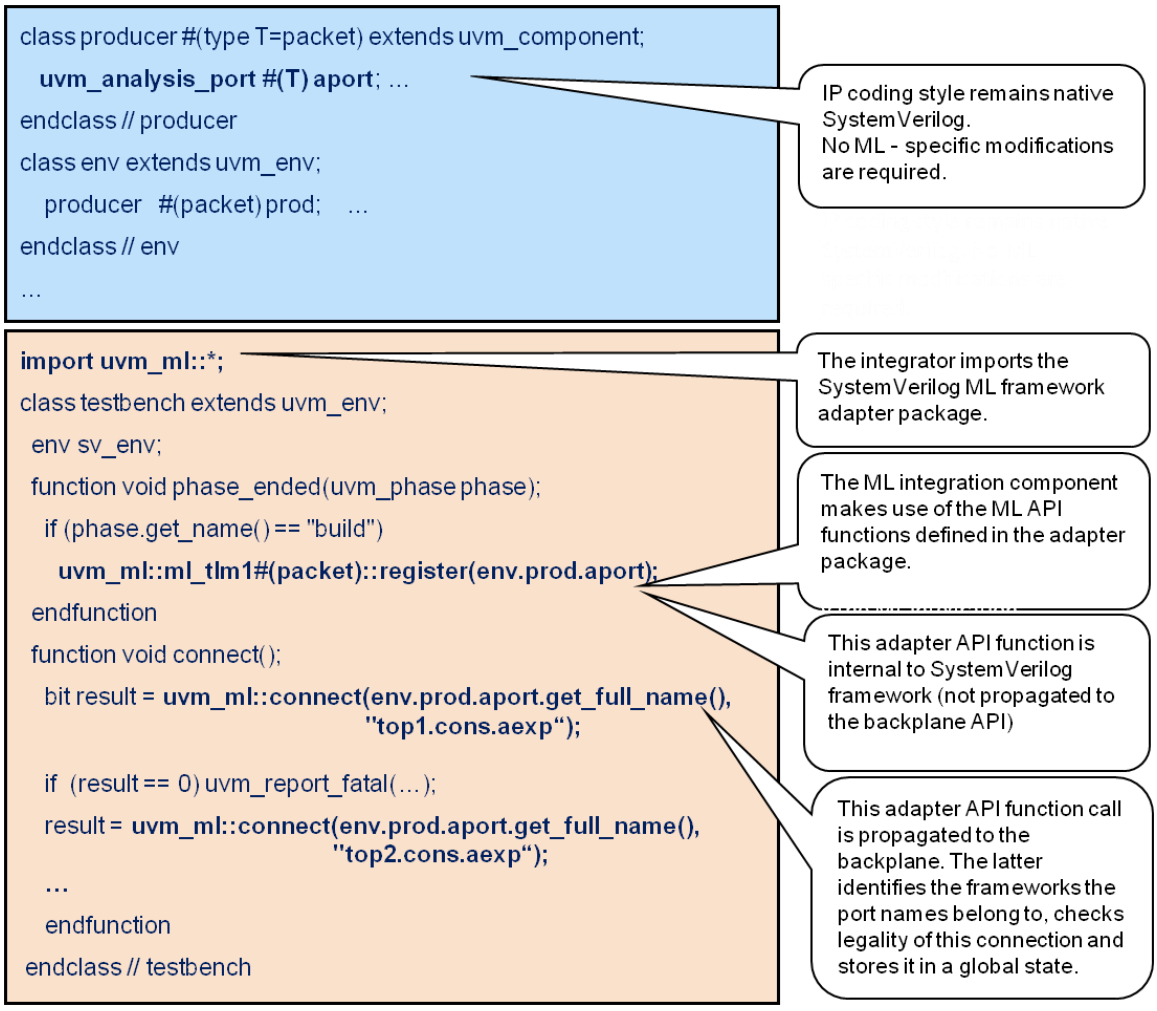


Figure 10: Example Connectivity with Integrator's View

Each ML connection must be established only once. The integrator does not need to invoke `UVM-ML connect` on each framework's side. Note that the SystemVerilog integrator is not required to explicitly identify the other frameworks to which the ports are connected. In its global state, the backplane saves the instance names of all framework root components. With that knowledge, the backplane detects the connected frameworks according to the ports full hierarchical names. This mechanism enables refinement of the integrated environment. For example, before receiving an IP component that is implemented in another language, the integrator can connect its high-level model, implemented in the same language, with the TLM ports connected by way of UVM-ML. Later, the integrator can replace the model with the actual IP, without the need to change the connections. This also simplifies migration from one framework to another.

Passing TLM transactions across framework boundaries adheres to the following principles:

- The backplane API passes opaque transaction objects. The backplane itself does not analyze or modify the transaction contents.
- The UVM-ML solution does not automate exporting or importing of type definitions between the languages. That means the transaction types must be explicitly defined in each participating framework language. For example, in the figure above, the type `packet` must be explicitly defined in SystemVerilog and in the foreign language(s).
- The transaction data is passed by copy. In order to pass a composite transaction, the producing side serializes it according to the established protocol. The receiving side consequently de-serializes the data into a container (object, array, and so on) of the corresponding type.

  Passing transactions by copy imposes limitations. In particular, the changes made to the data on one framework side do not become immediately visible on the other framework side. The data is synchronized only at the interface call boundaries—in accordance with the corresponding TLM standard semantics. For example, TLM2 clearly defines modifiable generic payload attributes. Following those requirements, only the modifiable data fields may be updated on the ML return path.

Serialization and De-serialization of Transactions in UVM-ML

The UVM-ML serialization protocol defines the following generic rules:

- The sizes of serialized and de-serialized transactions must match. Otherwise, the de-serializing framework adapter issues an error.
- A composite transaction may contain fields or elements of primitive and user-defined types. Nested composite user-defined objects are also supported.
- UVM-ML does not define a global type-map covering all languages and all legal type combinations. Instead, each specific framework adapter must document supported primitive types and their sizes in a serialized stream.
- UVM-ML supports object-oriented programming paradigm of passing subtype polymorphic transactions. For example, if a port interface is parameterized by a base-class transaction type, then an actual transaction can be of a derived type (sub-class). In order to enable identification of the actual transaction type, such a transaction is accompanied by a runtime type identifier in the serialized representation.
- Empty (*NULL*) objects can be passed.

- Dynamic-size containers, such as C++ vectors, *e* lists, SystemVerilog dynamic arrays, and so on are supported. Each instance of the dynamic-size container is prefixed with the element's number specifier.
- In general, any primitive type *LanguageN:TypeN* can be serialized and de-serialized to *LanguageM:TypeM* if their sizes match. Although the definition of the primitive type sizes is left for the specific adapters, UVM-ML defines the following basic guidance concerning the data sizes in the serialized stream:
    - The *word* size is 32 bits.
    - Each primitive type, in which the native-language size is smaller than a *word* size, must be aligned to the *word* size and padded (extended unsigned scalars must be zero-padded, while signed scalars must use sign extension)
    - Each primitive type, in which the size is larger than a word size, must be aligned to the size of multiple words.
    - Containers of elements that can be represented by a binary value (such as Verilog *array of bit*, SystemC *sc_bit* or *bool*) are an exception to the above rule, and must be tightly packed (which means *bit-by-bit*).
    - Extra specifiers (such as *type-id*, element number, or a null-object indicator) each occupy one word.
- Though UVM-ML does not operate with any language-specific types, some language-independent meta-type definitions are necessary in order to enable interoperability between the languages. Possible meta-types include:
    - *string*—Strings are represented differently in different languages and they may be even represented by multiple types in one language, as for example *std::string* and *char* * in C++. The UVM-ML serialization protocol defines that a string must be passed as a sequence of bytes (not aligning each byte to the *word* boundary), terminated with the *null* byte.
    - *enumeration*—An enumerated type (enumeration) must be passed as an unsigned numeric value aligned to the *word* size.
    - *4-value logic*—Is another potential meta-type.

The data alignment rules are motivated by runtime performance considerations (word-by-word serialization is faster than serialization adhering to the native data type sizes) and partially by the memory-consumption considerations (the *tight* packing of bit arrays enables the passing of large transactions without memory explosion). Those generic rules also simplify establishing the *contracts* between the framework adapters concerning the recommended assignment-compatible types in different frameworks.

Each framework ML adapter must provide facilities that automate implementation of the serialization and de-serialization procedures for the end-user. Examples of such automation facilities include built-in base UVM-ML packer classes, streaming operators, serialization automation macros, and so on. In particular, it is important for the adapters to provide built-in facilities for the most common base types, such as `uvm_object` and `tlm_generic_payload`.

Subtype Polymorphism in TLM Transaction Passing

UVM-ML supports subtype polymorphism of transaction objects. A port (or a socket) can be parameterized with a transaction base class. The calling code does not have to specify the actual runtime class of a transaction that can be a sub-class of the class used in the definition. The actual class can also differ from the defined one in the class fields, array elements or *`tlm_generic_payload`* extensions.

On the ML adapter side, this feature can be enabled by any framework that supports some native runtime type identification (RTTI). For example, C++ supports RTTI for any class that has a virtual table, while SystemVerilog UVM supports RTTI for sub-classes of `uvm_object`.

The UVM-ML backplane library automates RTTI on the ML level and eliminates the need for manual implementation of such mechanism by every ML integrator.

The backplane has a special facility that associates matching type names with unique numeric *id*'s. A producer framework adapter obtains the *id* from the backplane when it meets a new transaction type for the first time. The corresponding provided API function returns the *id* given the actual type name. The adapter's component that serializes the transaction then includes it in the serialized data stream.

The types with exactly equivalent short (unqualified) names are considered matching by default, and are assigned the same *id*. An additionally provided API function allows explicit overriding of the default match for the specified types' pair.

The *de-serializer* on the recipient framework adapter side allocates the transaction object according to the *id*. Hence, when the de-serializer meets an *id* for the first time, it obtains a type name from the corresponding backplane API function. Then the *de-serializer* allocates the object according to the type name.

Serialization Example for a TLM2 Interface

The following diagram shows equivalent types declared in three different languages: SystemVerilog, SystemC and ***e***, as well as the layout of the corresponding serialized transaction.

In this example, the actual transaction class type is a sub-class of (inheriting from) the base generic payload class in each of the above languages. For simplicity, the class name (`derived_gp`) is the same in all three languages. In addition, the actual transaction object has extensions—in this example only one sample extension type *extension1* is shown, but there may be many. The bottom part of the diagram shows the language-neutral layout of the corresponding serialized transaction. It includes the actual type identifiers of the transaction and of each of the extensions that are assigned.

SystemVerilog

```
import uvm_pkg::*;
import uvm_ml::*;

typedef enum {A_KIND,B_KIND,C_KIND} kind_t;

class extension1 extends uvm_tlm_extension;
  kind_t  m_kind;
  int     m_id;
  …
endclass

class derived_gp extends uvm_tlm_generic_payload;
  string m_txt;
  …
endclass
// SystemVerilog Initiator Socket
class producer extends uvm_component;
  …
  uvm_tlm_b_initiator_socket
                  #(uvm_tlm_generic_payload) isocket;
  …
  // passing an actual transaction object
  derived_gp  t;
  extension1 ext1;
  …
  t.set_extension (ext1);
  isocket.b_transport (t, delay);…
```

SystemC

```
#include "uvm_ml.h"

enum kind_t {A_KIND,B_KIND,C_KIND};

struct extension1: tlm::tlm_extension<extension1> {
public:
  kind_t mKind;
  int    mId;
};

class derived_gp: public tlm_generic_payload
{
public:
  …
  std::string mTxt;
}
// SystemC Target Socket
class receiver: public sc_module,
                tlm::tlm_blocking_transport_if {
public :
  tlm::tlm_target_socket<32,PROTOCOL_TYPES> tsocket;
  …
};
```

e

```
type kind_t : [AKIND, BKIND, CKIND];

struct extension1 like tlm_extension {
  % m_kind: kind_t;
  % m_id: int;
};
…
struct derived_gp like tlm_generic_payload {
    %m_txt : string;
};
// e receiver
unit receiver {
  …
  tsocket: tlm_target_socket of tlm_generic_payload
           is instance;
};
```

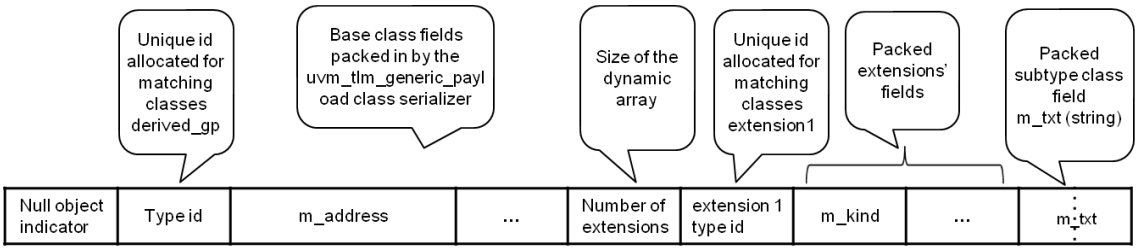


Figure 11: Serialization Example

Pre-/Post-/Run Phase Synchronization

Phasing Terminology

Pre-run phases—Initialize the system and consist of phases that are responsible for configuring, constructing, and connecting verification components in an orderly manner, prior to initiating environment execution. The pre-run phases are executed in a serial manner, and do not require inter-phase barrier control.

Post-run phases—Are responsible for extracting post-run information, checking the extracted data, reporting results, and shutting down the system. Post-run phases are executed in a serial manner after the run phases are complete.

Run phase—Are dynamic in nature. The run phases represent system execution of scheduled tasks in an orderly manner. Run phases require coordination by all participating components to enable an ordered and phased approach to running. Run phases include, but are not limited to: *reset*, *runtime configuration*, a *primary run phase*, and a *runtime shutdown* phase.

Phase actions—Refer to the notified phase activity during the life-cycle of a phase. Typically, the actions include: *PHASE_STARTED*, *PHASE_EXECUTING*, and *PHASE_ENDED*. These actions are notified as part of the run-phase lifecycle only.

Synchronized phases (fine-grained)—Are phases that can be synchronized across multiple frameworks. For pre- and post-run synchronized phases, phases are run serially and according to the prescribed hierarchical order.

Non-synchronized phases (coarse-grained)—Are phases which are not synchronized with respect to other frameworks. Pre- and post-run non-synchronized phases are not synchronized per phase, but rather at the start of each of the major areas (pre-run, run, and post-run).

Hierarchical Phase Propagation—Refers to pre- and post-run phase notifications which are propagated from the Child Hierarchical Proxy to its corresponding Parent Hierarchical Proxy in another framework, following the logical hierarchy established through inter-framework hierarchical build.

UVM-ML Phasing Overview

Phasing can be subdivided into the following three major areas:

1. Pre-run phase area (*build* occurs before *run*)
2. Run phase area (*run* occurs before *post-run*)
3. Post-run phase area (*extract* occurs before *final*)

At a minimum, each of these major phase areas needs to be synchronized at its start and end for all participating frameworks in the system, regardless of the phase synchronization granularity.

The pre- and post-run phase areas have similar characteristics and can be handled in the same general manner. The pre- and post-run phase propagations are required by the ML Logical Hierarchy in build-synchronous systems to propagate phase notifications to individual components across frameworks. Hierarchical phase propagation is not available to non-synchronous systems. It should be noted that frameworks in non-synchronous build systems manage the construction and phasing of their own local framework hierarchy.

Synchronous build requires a top-down hierarchy construction methodology. However, for all other pre- and post-run phases, a bottom-up traversal algorithm is normally followed. If systems do not require ML hierarchical facilities, then hierarchical phase propagation is not required.

Non-hierarchical systems may still employ synchronized phasing. If only a subset of hierarchy facility support is required, then only a subset of synchronous phasing is required in the system (refer to the Framework Feature Table). In either case, the phase solution needs to support these scenarios in a general and consistent manner.

The run phase area characteristics require participation acknowledgement for any framework component that wishes to actively participate in a notified run phase. Run phases require coordination between the participating components across frameworks and the phase controller which is driving the run phases forward. This is different from the pre- and post-run phases, where phases are driven forward sequentially without regard to framework participation.

The concept of ML Logical Root Nodes is used primarily in non-synchronous systems, where participating frameworks may need to instantiate their own sub-tree, due to build synchronization capability limitations. The usage model for this scenario is primarily to enable frameworks that need to have independent sub-trees constructed. As a result of incorporating this concept, the phase solution is generic in nature and can handle the key system scenarios.

In order to satisfy the previously stated requirements and to account for different system scenarios, the phase solution is composed of the following major component areas which

interoperate to provide a consistent solution, regardless of the type of system being described.

The major components are:

- Phase service: master phase controller and scheduling
- Master Phase Controller
- Backplane Phase Routing
- Framework Phase API
- Child and Parent Hierarchical Proxies

The following diagram illustrates the concept of a fully synchronized phased system:

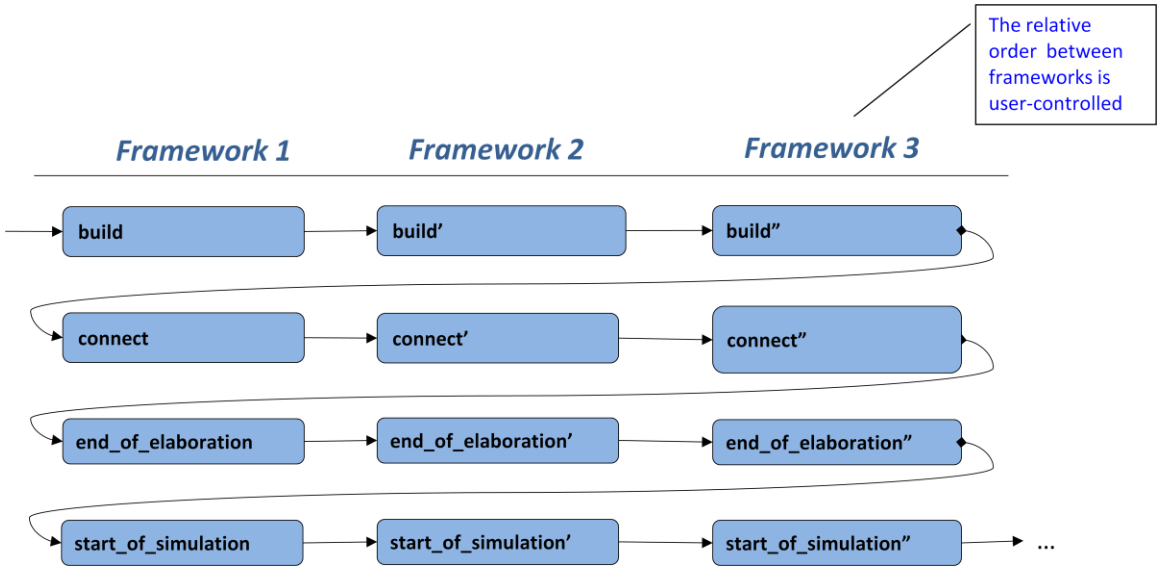


Figure 12: Simple ML Synchronization Concept

The diagram illustrates three participating frameworks that participate in the same phase in a synchronous manner. This represents a simple scenario, where participation for each phase is serially passed from one framework to another.

In reality, the order of pre-run participation highly depends on the sub-tree build order between the frameworks. While it is true that all frameworks participate in the same phase, it is in fact possible, that the framework ordering may include multiple re-entries into the same framework, where a single framework contains multiple sub-trees. Hence, the pre- and post-run phases, in following the build order (top-down or bottom-up) may oscillate between the participating frameworks in a non-trivial manner.

The relative phasing order for the pre- and post-run phases depends on:

1. The number of disjoint trees being created and their initial roots (user specified).
2. The build relationship between the sub-trees in a fully synchronous logical hierarchy enabled system, with re-entrant frameworks.

Framework Feature Table

In terms of methodology aspects that depend on synchronous phasing, the following table highlights a subset of framework features, with a focus on phasing and how it relates to enabling system capabilities.

	Phase			Other	
Framework Features	**Sync Build**	**Sync Pre-run**	**Sync Post-run**	**Multi-tops**	**Reverse Traversal**
Initialization	-	-	-	-	-
Logical Hierarchy	Yes	-	-	*	-
Multi-entry Frameworks	Yes	Yes	Yes	Yes	-
Hierarchical Configuration	Yes	-	-	-	-
Hierarchical Objections	Yes	-	-	-	-
Bottoms-up analysis/checking	Yes	-	Yes	*	Yes
TLM communication	-	Yes	-	-	-
Synchronize Facilities	-	-	-	-	-
Run phase notifications	-	-	-	-	-

Yes = required feature, - = not required feature, * = conditionally required

The general architecture of the UVM-ML phase system is as follows:

1. The list of independent ML Logical Root Nodes in the system is specified by the integrator. For non-synchronous systems, there is at least one Root Node per participating framework. For synchronous systems, the number of Root Nodes depends on the testbench hierarchy.
2. A single master phase controller, registered to the backplane, notifies phase transitions based on the common set of phases.
3. Multiple child hierarchical proxies act as hierarchical phase propagators to the target framework sub-trees (for pre- and post-run phases).
4. A single backplane routes phase notification requests, either from the master controller or from child hierarchy proxies to target frameworks. In the first case, the backplane routes notifications to the set of registered ML Logical Roots targets. For the latter case, known as *Hierarchical Phase Propagation*, the backplane routes phase requests to the target framework parent hierarchical proxy.

5. Framework Adapter Phase APIs receive notifications from the backplane (regardless of the source) to targeted sub-tree roots in their frameworks. The adapters translate the phase notification into one or more local framework phases, based on local phase mapping requirements.

Framework Phase Capability Scenarios

The following table outlines possible framework capability scenarios for supporting phase synchronization in an ML system. It is important to note that not all frameworks are able to be phase synchronous (for some or all phases).

The table provides a list of scenarios that can be supported in UVM-ML.

System Scenarios	Description
Case 1: Fully Synchronized, Multi-Entrant Frameworks	**Requirements**: Requires synchronization of all pre-run phase and post-run phases, with reverse traversal available between frameworks in order to support both top-down and bottom-up phase ordering. This is the most complete support for phasing between frameworks and logical hierarchical scenarios. **Scenario**: The system is capable of supporting full UVM phasing and full logical hierarchical support across frameworks per hierarchy. As each framework is fully multi-entry capable, any form of top-down logical hierarchy can be supported.
Case 2: Build Synchronized Only, Multi-Entry Frameworks	**Requirements**: Requires at a minimum, that synchronized build phase is supported between frameworks and multi-sub-tree is available for any multi-entry frameworks in the hierarchy. Attempting to re-enter a non-multi-entry framework results in an error, or undetermined behavior. Any form of top-down logical hierarchy can be supported. *Note*: The top framework must at least allow the build to be propagated to other frameworks (even if it is not fully synchronous). **Scenario**: The integrator is aware of which framework(s) support re-entrance and ensures that composition is limited to those multi-entry frameworks (multiple composite VIPs may re-enter capable frameworks).

System Scenarios	Description
	End Case: None of the frameworks are multi-entry. The integrator needs to instantiate all components that are part of the same framework into a single sub-tree for each framework.
Case 3: Non-Synchronized Pre- and Post-Run Phase	**Requirements**: Does not have any synchronized pre- and post-run phases beyond the requirement that the start and end of the pre- and post-run phase must be synchronized to coordinate initialization, run, and shutdown. Logical hierarchy facilities are not supported. Composite VIPs must be instantiated in their own frameworks. **Scenario**: Only TLM connectivity and non-hierarchical synchronization facilities are required. Multiple Framework sub-trees/tops may be instantiated in multi-top capable frameworks if disjoint instantiation is required.

The phasing solution supports all the scenarios (from the ideal fully synchronized, fully multi-entry system to the non-synchronized system) as described in the previous table.

Synchronization Facilities

Beyond configuration and resources, phase synchronization, TLM communication, and time synchronization, there are other synchronization methodologies present in UVM that need to be taken into account by the Backplane. These facilities are more light-weight and low-level as compared to other methods, such as TLM or phasing, and are often used in tighter forms of synchronization between objects in the system (intra-modeling synchronization internal to reference models, for example).

The light-weight synchronization facilities that are part of the UVM methodology include *events*, *objections*, and *barriers*. Examples of some of their properties are in the following table:

Sync Type	Actions	Data
Event	trigger	Optional payload
Objection	raise, drop	-
Barrier	initialize, wait	-

Each facility may be associated with one or more actions, and some may optionally transport data with their actions. The backplane needs to support propagating

synchronization requests from the originator to participating frameworks, in order to enable VIPs in different frameworks to react to the notifications.

This will be addressed in a future release of the UVM-ML Backplane along with providing the architectural details in this document.

Message Reporting and Control

This section will be included in a future version of this document.

Error Handling and Reporting

This section will be included in a future version of this document.

Time Management

This section will be included in a future version of this document.

Key System Use Cases and Scenarios

Multi-language (ML) verification environments are needed for various use cases. The UVM-ML solution is focused on three main use cases, while attempting to enable as many of the additional use cases as possible without over-complicating the solution.

The main use cases are:

- Integrating a Multi-Language Verification Environment
- Embedding a Reference Model
- Creating a Composite VIP

Other use cases include:

- Reuse of a Testbench for Simulation Acceleration
- Verification of SystemC Models

Integrating a Multi-Language Verification Environment

Testbench integration is a routine task for every verification project. Testbench components are replaced, or new components may be added, for every revision of the project. Often these components were developed in a different language or methodology than the rest of the testbench. Legacy building blocks are often critical components of the testbench and re-developing them in a different language is not an option.

The challenge here is to minimize the integration effort by standardizing the ML interfaces and control API between frameworks, by providing tools and a methodology to integrate frameworks into a coherent testbench.

Embedding a Reference Model

Reference models are often developed in languages different from the Verification IP, such as C, C++, or SystemC. Incorporating the reference model into the verification environment makes the task of verification checking more efficient and reliable than developing new checkers from scratch.

The main challenge here is synchronizing two independent simulation threads in such a way that the reference model provides the expected result at the right time—when the actual result is received from the DUT.

Creating a Composite VIP

A composite verification IP component comprises sub-components written in different languages. A typical example is a VIP component that was developed in a high-level verification language such as ***e*** or SystemVerilog, and is using a driver or collector developed in some low level language such as C or Verilog. A composite VIP component like this combines the advantages of high abstraction-level control with efficient handling of signals and clocks connected to the DUT.

The challenge in this case is to provide efficient interfaces between the sub-components of the VIP, both for passing data, and for controlling the DUT, while maintaining the independence of each sub-component.

Overview of Framework Modifications

The existing standard frameworks were not designed with collaboration in mind. This section describes the motivation for some extensions in UVM, SystemVerilog, and OSCI SystemC that will enable collaboration in a few areas. These changes can be used as a basis for further discussion in the corresponding standardization working groups.

Below is a brief description of each of the modified standard frameworks, included in the UVM-ML package.

UVM SystemVerilog Support of Synchronized Phasing

In order to enable synchronized phasing, we added the following methods in *uvm_root*:

- *add_top_level()* supports procedural instantiation of a *root node* (see terminology described in section Tree Terminology).
  Creation of the ML root nodes, potentially in multiple frameworks, can be considered the initial step of the phasing process. *Uvm_root::add_top_level()* is invoked by the adapter, if the integrator specifies the UVM SV root node either procedurally or declaratively. This function accepts the corresponding type and instance name arguments and creates the UVM component using `uvm_factory`.
- *do_nonblocking_phase()* and *do_blocking_phase()* execute one phase per one ML sub-tree, beginning either with an ML root node or a child junction node. The

latter may happen if a SystemVerilog component is a hierarchically instantiated child of a foreign framework component. In both cases, the backplane iterates over the list of root nodes and lets the phasing service provider to propagate the phases by way of the backplane API and the ML adapter. The adapter invokes *do_nonblocking/blocking_phase()* for each phase of each ML sub-tree separately (rather than in a hard-coded native framework iterator).

Some related internal functions were also added to the *uvm_phase* class.
Procedural activation of the ML test requires a function other than the native UVM *run_test()*. This function (*uvm_ml_run_test*) is implemented in the ML adapter. Consequently, an additional check was needed in order to prevent a conflict between *uvm_ml_run_test()* and *run_test()*.

The extra modification in *uvm_root::run_test()* issues a warning if it is called after beginning of the test.

UVM SystemVerilog TLM2 Modification

UVM-ML passes serialized transactions. In order to implement a generic predefined packer for *uvm_tlm_generic_payload*, it needs to access to the payload extensions. Only in this way can it iterate over the actual extensions of the corresponding transaction object. Hence, we removed the 'protected' access qualifier on *m_extensions* to enable this.

UVM SystemVerilog Packer Modification

As described in Serialization and De-serialization of Transactions in UVM-ML, most of the packed fields are aligned to the word boundary. An exception was made for the arrays of two-value elements (array of bits), which are tightly packed, so that each element occupies only one bit in the serialized stream. In order to support this for an ML packer by extending the *uvm_packer* class, we added the methods *pack_field_element()* and *unpack_field_element()*. Now these methods are used for the array elements instead of the original methods *pack/unpack_field()*. In the native packer, both pairs are fully equivalent but the separation of the functions enables them to be overridden independently in the UVM-ML packer sub-class.

Another local modification is the addition of a *uvm_packer* function named *unpack_object_ext* that receives an *inout* directed argument, which is in contrast to the native function *unpack_object* that receives an input argument only.

OSCI SystemC Support of Synchronized Phasing

Similar to what was described above for UVM SystemVerilog, the native OSCI SystemC does not allow synchronizing its phases with other frameworks. In order to enable this, the ML adapter needs to be able to execute each phase separately for each SystemC quasi-static sub-tree. Additionally, all the phasing activities also need to be applied to the

TLM ports, threads, and any phase sensitive object created during the quasi-static phases. For these purposes, the UVM-ML package contains a modified version of files `sc_simcontext.h` and `.cpp` taken from the standard OSCI version 2.2.

We hope that in the future, these corresponding enablers will be included in the standard SystemC version. Refer to the UVM-ML package for an example of how to apply these changes.

OSCI SystemC TLM2 Generic Payload Extensions Accessibility

Similar to what was described above for UVM SystemVerilog, OSCI does not provide a public access for iterating over all extensions in the payload. In order to enable this capability in the transaction packer, we included in the UVM-ML package a modified version of file `tlm_gp.h` which adds a friend class to *tlm_generic_payload*.
The UVM-ML packages illustrates how this change can be applied to the TLM library used here (in particular TLM 2009-07-15). The UVM-ML package will be updated in the future with the corresponding changes for SystemC 2.3, where TLM has been folded into the SystemC packages directly.

It should be noted that the UVM-ML package includes the *make* file that compiles the adapter automatically, as part of the centralized setup.

Current Limitations

System Initialization and Shutdown Limitations

System Initialization and Shutdown currently has the following limitations:

- An API that allows a framework to notify the backplane whether the framework supports each UVM-ML feature (the capabilities API) is not implemented yet. As a current workaround, a framework ML adapter can assign NULL pointers to the corresponding required API tray function pointers. The backplane will not attempt to call those functions for that framework.
- The following services are not implemented yet:
    - Shutdown coordination by way of a shutdown service framework is not implemented
    - Time Quantum feature is not implemented
- The aliasing of framework names is not implemented. As a result, for example, the UVM SV framework is identified always simply as `SV` (not `UVM-SV`).
- The `uvm_ml_run_test()` function is not implemented in SystemC.
- Simulation *restart* and *reset* are not supported.

TLM Communication Limitations

TLM communication currently has the following limitations:

- Currently, the adapters support only TLM transactions and their class fields of types derived from `uvm_object` or `tlm_generic_payload`. Support of other types (including other user-defined types and primitive types) is yet to be added.
- Default maximum size of a TLM transaction is currently 4k bits. The user needs to explicitly set the environment variable: `UVM_PACK_MAX_SIZE` to increase the supported transaction size.
- Disabling (or *killing*) of blocking ML threads is not supported.
- ML broadcasting of analysis ports is not supported on the SystemC side.
  This means that an SV analysis port can broadcast to ML exports including SystemC, but a SystemC analysis port cannot broadcast to ML exports.
- ML hierarchical binding is not supported on the SystemC side.
  This means that you cannot bind a SystemC export to another SystemC export and to a SystemVerilog port.
- 4-value logic fields of transactions are not supported.

Phase Limitations and Observations

The phase solution currently has the following limitations:

- Only the common and UVM phases have been implemented in this release.
- User phases are currently not visible, or are notified between frameworks:
  - Currently, a local framework can issue user run phases within its own framework.
  - Future implementation will allow the registration of user schedules and phases.
- Child Hierarchical Proxies only propagate pre- and post-run phases to target frameworks. Run-time phases are notified independent of hierarchy.
- The Breadth First Search (BFS) algorithm is not enforced between frameworks. This implies that a fully synchronous ML system may not follow the same BFS traversal algorithm that a single framework system may follow. However, depth first is enforced for participating frameworks in logical hierarchy. This implies that no assumptions should be made about peer ordering in either a single or multi-framework system.

Resource and Configuration Implications

The distributed resource and configuration solution has the following implications:

- Resource or configuration data which cannot be copied and serialized cannot be broadcast since such data cannot be referenced across language boundaries or in other frameworks.
- ML Configuration shall not rely on sibling or parallel branch ordering for configuration specification due to the limitation of non-predictable breadth-first phasing in an ML system. Only parent-child build ordering is guaranteed in a synchronized build system. As such, non-lineage references are deemed to be unsafe and cannot be guaranteed in different ML implementations.
- ML Safe resources require that all updates to a resource are performed through the resource item. Any update performed to an object referenced in a resource will not be visible to the backplane and may compromise synchronization of resource stored values. It should be noted that only those resource items which are required for ML need to adhere to ML Safety guidelines. Other *local* resources may follow normal local framework guidelines.
- For the initial ML implementation, the priority concept will not be supported. This is because it is difficult to distinguish between similarly named resources whenever an attribute is updated. If the distributed ML resource database approach is deployed, then it could be incumbent upon the adapters to track the ID of any duplicated resource that is presented to the system. The added adapter complexity would make it more difficult to implement. In addition, the rules for priority would need to be consistent between each distributed copy of the database in any participating framework as well, which would be hard to enforce and achieve. As such, for the purposes of the initial ML specification, it is assumed that only one copy of each resource will be provided to the resource database. If the implementation moves towards a central database approach, then supporting priority and duplicated resources may be considered (as the handling and rules would be centralized).

Future Enhancements

Regular release distributions will be provided to address and add planned future enhancements for UVM-ML. The following areas will be addressed in subsequent releases of UVM-ML:

- Messaging service

- Time quantum service
- Error reporting and handling
- Configuration and Resources
- Synchronization facilities (events, objections, barriers)
- Framework Capabilities
- Coordinated system shutdown
- Additional frameworks as demonstration vehicles
- Enhanced demonstrations and installation tests
- Comprehensive user and reference guides

Acknowledgements

The following people provided material for this white paper and participated in reviewing it:

- Alex Chudnovsky, Cadence Design Systems, Inc.
- Ngoc Luu, Advanced Micro Devices, Inc.
- Guy Mosenson, Cadence Design Systems, Inc.

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

References

[1] The UVM library and User Guide from Accellera Systems Initiative can be downloaded from here: http://www.accellera.org/downloads/standards/uvm

[2] UVM-ML previous contributions can be found here:
http://www.uvmworld.org/contributions-details.php?id=98&keywords=UVM_ML

This includes the document: *uvm_ml_opensource.pdf* (located in the *docs* directory of this UVM-ML tar file). The document includes sections on UVM-***e*** and UVM-SC (as they were in the previous contribution, July 2012)

[3] Horace Chan; Brian Vandegriend; "*Hardware/Software Co-Verification Using Specman and SystemC with TLM Ports*"; DVCon 2012

[4] Marcio F. S. Oliveira; Christoph Kuznik; Wolfgang Mueller; Wolfgang Ecker; Volkan Esen; "*A SystemC Library for Advanced TLM Verification*", DVCon 2012