Requirements specification for TLM 2.0

Version 1.1
September 16, 2007

Bugs and Suggestions
Please report bugs and suggestions about this document to
http://www.systemc.org

History

James Aldis	Created	3/15/2007
Marcelo Montoreano	Updated from comments in Thorsten document, reflector, meeting minutes.	3/29/2007
Marcelo Montoreano	Updated from DATE F2F meeting notes, previous meeting minutes	4/22/2007
Tim Kogel	Added high-level requirements, started restructured chapter on loosely timed TLM API	4/25/2007
Tim Kogel	Restructured chapter on approximately timed TLM API	5/2/2007
Marcelo Montoreano	Updated from reflector input	5/25/2007
Marcelo Montoreano	Updated with WG input. Added copyright notice.	5/31/2007
Bill Bunton	Added HW performance verification use-case	6/1/2007
David C Black Mike Meredith	Added HW refinement and Implementation use case	6/1/2007
John Aynsley	Added clarifications to Interoperability Requirements	7/13/2007

Introduction

On 6/19/2003 the Transaction Level Modeling Working Group adopted the following charter:
“The TLM working group develops and recommends techniques and provides a foundation class library for SystemC which promotes interoperable modeling and communication at levels of abstraction higher than RTL for the purposes of specification, simulation, verification, implementation, and evaluation of SOC designs."

The TLM 2.0 standard is focused on model interoperability for SoC platforms, which are based on memory mapped buses. We also provide the basic primitives to model arbitrary components and systems, but for those we will not achieve the same level of interoperability.

This document covers the requirements for the definition of the OSCI TLM 2.x standard, which is being developed by the OSCI TLM working group.

The workgroup has identified several applications for Transaction Level Modeling but is concentrating in the following:

- Modeling of loosely-timed systems based on memory-mapped busses (sometimes called PV).
- Modeling of approximately-timed systems based on memory-mapped busses (sometimes called PVT).

Since they are many types of memory-mapped busses and we feel that we cannot anticipate all attributes that may need to be transferred, the workgroup has taken a multi-prong approach:

- A generic TLM API that is based on user-defined templates.
- Low-Level data type recommendations to be used in user-defined templates.
- Data structures for the TLM API that make it fully specialized and models a “Generic” memory-mapped bus.

The proposed Generic memory-mapped bus may be good enough to simulate some “real” protocols, at a loosely-timed functional level. Some other protocols may have functional details that cannot be mapped to the proposed data structures. In those cases, direct interoperability will be limited and bridges will be required.

Part I of this document defines the use cases for transaction level modeling, and highlights the areas that are is the scope of the 2.0 OSCI TLM release.
Part II of the document defines detailed implementation requirements for the OSCI TLM 2.0 standard. This part in split in several sections:
Section 3 deals with requirements that support software development and performance analysis.

Section 4 sets requirements that support architecture analysis, where timing information needs to be more accurate that in the previous chapter.
Section 5 sets requirements for efficient modeling, aimed at lowering the effort required to understand and use the standard.
Sections 6 and 7 set the requirements for the layered approach that is intended.
Sections 8, 9 and 10 set requirements for quality, portability and documentation.
Sections 11 to 17 list additional requirements that apply to all use cases, take in count legacy models and, in general, try to make the standard more robust.

Acronyms and Terms

The OSCI TLM Work Group has created a separate document (OSCI TLM 2 Glossary) that describes in detail the terms and acronyms used in this document. Please refer to that document.

Part I: TLM Use-Cases and High-level Requirements

This chapter provides a comprehensive (but not necessarily complete) list of use-cases for TLM models. For completeness we first list the use-cases we consider to be outside the scope of the OSCI TLM standard. The goal is to outline the purpose of transaction-level models. These use-cases infer high-level requirements in terms of speed, accuracy, flexibility, expressiveness, etc. The detailed implementation requirements for TLM2 have to satisfy the high-level use-case requirements.

1 Definition of non-TLM Use-Cases

We consider the following use-cases to be outside the scope of the OSCI TLM standard:

- *Requirements Modeling* is typically addressed with UML
- *Algorithm Development* and *Algorithm Performance Analysis* is typically addressed with a domain specific Model of Computation (MoC), like e.g. KPN, SDF, CSP, synchronous languages, etc. The SystemC based modeling using domain specific MoCs is to some extent discussed in other OSCI working groups.
- *Logic Synthesis* and *Behavioral Synthesis* is discussed in the OSCI synthesis working group

2 Definition of TLM Use-Cases

We consider the following use-cases to be generally in the scope of the OSCI TLM standard. Only a subset of these use-cases will be addressed in the TLM 2.x standard. All the use-cases refer to the specification and implementation of System-on-Chip platforms and/or their components.
(source: ESLX, NXP, CoWare, TI)

2.1 SW Application Development and HW/SW Integration

Use-case Description: SW implementation phase, use TLM model of the HW platform for SW application development, enables development of drivers/OSes/application frameworks/applications/etc., develop and debug real embedded SW on the target platform

Use-case requirements: run real, unmodified SW, attach SW debugger, very fast

Resulting platform model requirements: very fast (boot OS within seconds), register accurate, functionally complete to run real SW, loosely timed to run real SW (timer interrupts fire roughly at the expected time to successfully boot OS), correctly reflect the endianness impact of the HW (again to run the real SW)

2.2 SW Performance Analysis

Use-case Description: SW validation phase, use TLM model of the HW platform and real embedded SW to measure the expected performance of the SW application.

Use-case requirements: run real, unmodified SW, derive reasonably accurate performance data from simulation to enable SW performance profiling and optimization,

speed requirements not as high as 2.1, but simulation still needs to be fast enough to execute significant amount of SW in interactive mode

Resulting platform model requirements: register accurate, functionally complete to run real SW, timing sufficiently accurate to derive reasonably accurate performance data (e.g.: the simulated execution delay of a piece of SW is +/- 10% of the real silicon), analysis instrumentation, do due speed difference with loosely timed mode an runtime switch from loosely timed to approximately timed is required.

2.3 SW Architecture Analysis

Use-case Description: Early SW specification phase, use TLM model of the HW platform and a workload model of the SW (non-functional or functional task graph) to explore and determine the partitioning and mapping of the SW onto the HW platform.

Use-case requirements: execute workload model of the SW on resource model of the

Use-case requirements: HW performance requires real software executed by a transaction timing accurate processor model (ISS), and the mixed TLM model which may be any or all of the following:

- Back-annotate PVT models
- RTL co-simulation
- RTL accelerator co-simulation
- RTL emulation “co-simulation”

Resulting platform model requirements: a highly configurable, transaction timing accurate hardware model which includes:

- Register accurate models for all hardware used by real software or used for verification.
- Bus performance visibility, visualization and data collection
- Subsystem performance visibility, visualization and data collection

The primary requirement is that the model has timing accuracy sufficient to generate performance data with an accuracy of +/- 1% and sufficient flexibility to allow performance optimization of the simulation speed. Simulation speed may be sacrificed, giving preference to timing accuracy and configuration flexibility.

2.6 HW Refinement & Implementation

Use-case Description: Use TLM system to verify a pin-level behavioral model that is input into synthesis (using TLM system to verify a RTL with a pin-level interface is covered in the section titled HW Functional Verification).

Need to be able to build a system in which:

- all communications is through one or more bus interfaces
- some communication is through one or more bus interfaces and additional communication occurs through other point-to-point connections.

To support modules with:

- one or more bus interfaces.
- one or more bus interfaces and additional point-to-point interfaces

Use-case requirements: Must be possible to write a model that conforms simultaneously to both the OSCI synthesis standard and the TLM standard.

Need to be able to write a model that can be used for both untimed simulation and for synthesis with minimum modification of source code. This allows the untimed model to be used as an executable specification, and reduces the implementation effort by allowing it to be the implementation.

Must be possible to have a convenient coding style that allows the user to switch between TLM ports and pin-level ports without excessive manual effort.

Resulting platform model requirements: TLM environment must be able to support communications to a bus and allow for side-band communications to other modules with either TLM or pin-level connections.

The implementation model can only communicate through pin-level bus interfaces. An untimed model based on the implementation model can only communicate through the TLM abstraction of the bus interface. For some other purposes, it may be desirable to make speed optimizations in an untimed model that cause some of it's communications to bypass the TLM abstraction of the bus interfaces. If the TLM standard supports such speed optimizations, it must be possible to swap between an untimed model that only communicates through the bus interface and a model that (as a speed optimization) sometimes bypasses the bus interface.

Must be possible to build an interoperable untimed model that contains processes. There must be available a clock for synthesis.

2.7 HW Functional Verification

Use-case Description: Use TLM model, representative software, and applicable portions of HW unit-test to verify that the Hardware implementation meets the system functional requirements. Goal is to verify that the Hardware implementation meets the system requirements: Software visible definition of hardware is accurate; Hardware and Software interaction works; and system interaction of the hardware unit under test and related software combination performs as required with other related hardware and software combinations.

Use-case requirements: Mix TLM model with any or all of the following:

- RTL co-simulation
- RTL accelerator co-simulation
- RTL emulation “co-simulation”
- Portions of the RTL testbench; potentially in a language other than SystemC

Will need to execute software models that are available. Software models may vary between “timing accurate” to “completely timing inaccurate”. Typical software models may include:

- Direct execution model of “proof of concept” device drivers
- Direct execution model of device drivers
- OS with device driver on a timing approximate processor model
- OS with device driver on a timing accurate processor model (cache accurate)
- OS and all device drivers on a processor emulation board

Resulting platform model requirements: The resulting platform model will need to have the following accuracy levels of accuracy including:

- Register accurate models for all hardware “touched” by the software model
- Interrupts that fire roughly at the expected time
- Bus performance visibility and visualization

- SW performance profiling (depending on the software model used)
- Correct implementation of endianess (need to verify that software and hardware have the same “sense of endianess”)

Model timing accuracy needs to have the ability to add randomness in order to assure that all timing correct scenarios are seen by the hardware. Randomness is needed at the TLM bus level and within the “transactor” used to communicate with the unit under test.

Model is to run at significantly faster speeds (100X+) than RTL-only simulation. The primary requirement is that the TLM model be flexible to allow multiple sources of software models and RTL models. Speed can suffer slightly to meet the flexibility requirement.

2.8 TLM use-models addressed by the TLM 2.x standard

For the TLM 2.x standard we intent to focus on the following use-models:
2.1, 2.2, 2.4, 2.7
On the other hand we defer the following use-cases for later TLM releases:

- 2.3: we do not standardize an API for the application modeling (in the past, this has been referred to a "SystemC 3.0")
- 2.5: we do not standardize an API for cycle accurate bus models

Part II: Implementation Concepts and Requirements

This part of the requirements specification defines the detailed implementation requirements for the OSCI TLM 2.x standard. The implementation requirements are supposed to full-fill the high-level requirements inferred by the TLM use-cases.

3 TLM Implementation Requirements for the support of SW Development and Performance Analysis

This section summarizes the requirements for the creation of TLM platforms, which can be used for SW Development (and Performance Analysis?). We also list the considered and discarded implementation options for the requirements.

3.1 Interoperability

Obviously Interoperability is the foremost requirement of every standardization effort. The goal is that users can develop interoperable models for SW development platforms independently, using only OSCI documentation.

3.1.1 Interoperability Requirements

3.1.1.1 Interoperable Memory Mapped Bus API(2.0)

The goal is to define a highly interoperable, loosely timed TLM API for functional modeling of SoC platforms, which are based on memory-mapped busses. The idea behind this requirement is that there be a single API defined which can model most MMB protocols at a pure-functional level. Interoperability can be accomplished by using the proposed API and data structures, using the API and a data structure derived from the proposed, or using adaptors from proprietary TLM to the proposed API and structure.

3.1.1.2 Generic data structure for the loosely-timed Generic Memory-Mapped Bus (2.0)

We need a generic transaction data structure that works well for the majority of memory-mapped busses. The transaction data structure must be sufficiently expressive to create TLM platform models for SW development.
We have to support the key features common to all MMB protocols: read, write, bus locking, master-to-slave connection.
Starting point for discussion of typical transaction attributes is data structure in TLM2 draft kit.

3.1.1.2.1 Support transactions of run-time-selectable size

3.1.1.2.2 The semantics of the attributes in the data structure must be fully defined

Obligations of both Master and Slave (which attributes must be interpreted, how master/slave interact with the attribute) should be stated.

3.1.1.2.3 Keep structure as simple as possible

Simple masters and simple slaves should not be taxed interpreting a full set of attributes that doesn't apply to them.

3.1.1.3 Bridges between modules should be avoided as much as possible (2.0)

It is desirable to avoid the need for bridges (adaptors) as far as possible between models for different protocols (e.g. AXI and OCP) or which use different features of the same protocol (e.g. slaves that do not support burst mode). In some cases, bridges are unavoidable, i.e. between detailed models of fundamentally incompatible protocols.

Where the attributes and semantics of the Generic MMB are inadequate, the recommended approach is to derive a new class from the Generic MMB data structure (payload) containing the necessary extensions. There is no guarantee of interoperability between extended protocols. The creator of such extensions must assume responsibility for ensuring interoperability or for providing bridges (adaptors) where appropriate.

The semantics of the Generic MMB protocol, the choice of attributes and the default values of the attributes should be carefully chosen to minimize the need for explicit bridges (adaptors).

3.1.1.4 Interface should prohibit binding of incompatible models (2.0)

We want to connect only compatible components which can successfully communicate.
Incompatible models should be detected at compile-time or run-time.
The information in the transaction data structures must be sufficiently complete and well-defined that a component is always able to determine whether or not it will be able to interoperate without the need for a bridge (adaptor).

3.1.1.5 Compatibility with Approx-timed (2.0)

- use of same data structures as in approx-timed (normally loosely-timed will use a subset of those used by approx-timed)
- But: possible to create an loosely-timed -only module (no requirement that all modules support loosely-timed and approx timed APIs in parallel)
- On-the-fly switching of an interface between loosely-timed and approx-timed must be possible (see 2.2)

It must be possible to switch the simulation from a loosely-timed mode to a more accurate timing mode.
The idea is to execute large amounts of simulated code and then study certain aspect of the simulation with greater timing detail.

3.1.1.6 Endianness (2.0)

It should be possible to model big-endian systems on a little endian host.
It should be possible to model mixed-endianness platforms.

3.1.1.7 Compatibility with future accurate-timed MMB API (researched by 2.0)

Cost in simulation speed of bridging from a loosely-timed Generic bus to some accurate-timed Generic bus must be as small as possible (see 2.7). Note that this requirement is rather difficult to enforce given the lack of such an API to test against. But some consideration of how it might look is warranted.

3.1.2 Discarded Interoperability Requirements

3.1.2.1 "No Bridges" Interoperability

We agreed to have an API, which prohibits the binding of non-compatible models as much as possible. A weak API, which allows the binding of arbitrary compliant (but non-compatible) models is considered to be unsafe.

3.1.3 Considered Implementation Options

3.1.3.1 Typical transaction attributes

The version in TLM-2.x would be a "Generic" MMB, which would contain a typical subset of the features (transaction attributes) to be found in real MMBs such as OCP, PCI, AXI, etc. Typically such 'real' MMB protocols are simply different implementations of the same basic functionality. The common denominator of the functionality should be captured as far as possible in a set of typical attributes.
The MMB is assumed to cover only the view from the module: the module has a socket, typically with address bus, data bus and control lines, and sees the rest of the system as a single external entity on the other end of these busses. Sockets are either masters or slaves. Interoperability guaranteed between users developing correct models independently, using only TLM 2 documentation, without any bridge between them.

Example attributes for typical protocol are e.g. byte-enable and endianness.

3.1.3.2 Non-template data types

To avoid interoperability issues due to different data-types for the same data-size. Instead we use uint64 for the address type and unsigned char array for the data type.

3.1.3.3 bi-directional transport API

Chosen for high-speed functional modeling,
Transport enables use pointers to data structures with well-defined ownership policy

3.1.3.4 Attribute to indicate the data-size of a module

Obviously this limits the out-of-the-box interoperability, but this is required to fulfill requirement 3.1.1.4.

3.1.4 Discarded Implementation Options

- Minimum set of attributes + extension mechanism: speed overhead for access to extensions, unclear how to handle of un-safe transactions
- No extension mechanism provided, only example on how to extend data structure using C++ inheritance

- Weak API, with no means to detect non-compatibility between models.

3.2 Timing Accuracy

3.2.1 Timing Requirements

3.2.1.1 TLM platform for SW development must be loosely timed (2.0)

It must be possible to loosely keep track of the timing of a module with respect to other modules in the system, to be able to make the system advance in a predictable manner. There must be the capability to provide sufficient timing to boot an OS in a system composed of loosely-timed MMB components (see 2.1).

3.2.1.2 Model Synchronization (2.0 stretch; research @ minimum)

A synchronization scheme should allow complex system modeling in an interoperable way with high speed. In the context of MMB based TLM model, synchronization refers mostly to the coordination of the initiators in the system. The model synchronization scheme also defines the handling of interrupts.

3.2.2 Considered Implementation Options

- Add timing parameter to transport signature
- Timing information shall only be modeled with sc_time objects

3.2.3 Discarded Implementation Options

- Add timing parameter to transport data structure
- Modeling of timing using unsigned int

3.3 Speed

Speed is one of the principal objectives of a loosely-timed simulation. The following mechanisms have been identified as enablers of high-speed simulations:

3.3.1 Speed requirements

The speed requirements heavily depend on the use-case. Of course we always want to be as fast as possible, but the laws of physics on the tradeoff between speed and accuracy of SystemC models must be acknowledged.

3.3.1.1 Speed requirements for the use-cases addressed by the loosely timed TLM API (2.0)

We consider the following numbers to be lower boundaries to render a TLM model useful for the respective use-case.

- SW Application Development and HW/SW Integration: 50 MT/s
- SW Performance Analysis: 10 MT/s

Obviously these numbers are highly dependent on the complexity of the modeling system. The goal is that if the TLM API does not hinder us to reach at least this minimum required simulation speed for a system of medium complexity.

3.3.1.2 Mixed-mode simulation speed (2.0)

Cost in simulation speed of bridging from a loosely-timed Generic bus to the approx-timed Generic bus must be as small as possible

3.3.2 Considered Implementation Options

The implementation options for high simulation speed are very controversial, since these features impact other important things like safety (pass-by-pointer, DMI), general interoperability (temporal decoupling), and modeling efficiency. Nonetheless these options are widely applied in the industry to achieve the required simulation speed, especially for use-cases 2.1 and 2.2. As long as there is no standard API for these features, it will not be possible to create "speed-interoperable" models, i.e. model that run fast in every standard compliant environment.

3.3.2.1 Pass by Pointer

3.3.2.2 Direct Memory Interface (TLM2.0 stretch; research @ minimum)

There must be a way for components to acquire a host-pointer to the memories of interest. The DMI mechanism infers the following requirements:

3.3.2.2.1 Invalidation of Direct Memory Pointers

Under certain conditions (change of memory map, for example) it is necessary to invalidate the host pointers to memory acquired with the DM interface.

3.3.2.2.2 Must be optional

3.3.2.3 Correct level of abstraction

Do not include information such as bus widths and detailed bus burst structures which affects only the timing and not the functionality of the transactions

3.3.2.4 Temporal decoupling (should be possible by 2.0)

In order to obtain maximum simulation performance the use of "*temporal decoupling of masters*" is used. This refers to not executing bus masters in temporal lock-step but, instead, letting one master run ahead in time. The amount of time a master is allowed to run ahead ("*slack*") needs to be configurable. In practical cases this is required in order to find the proper balance between performance and temporal accuracy. Often it is also required to reduce the slack to some small value for testing and debugging purposes.
It must support and demonstrate the ability to apply uniform temporal slack to PV masters.
Requirements for temporal decoupling mechanism:

3.3.2.4.1 Temporal decoupling must be aligned with PV synchronization mechanism

3.3.2.4.2 It must be possible to control the slack between the master and the SystemC kernel.

3.3.2.4.3 It must be possible for the target to trigger the synchronization with the SystemC kernel *before* the execution of the behavior (synchronization on demand)

3.3.2.4.4 It must be possible for the target to trigger the synchronization with the SystemC kernel *after* the execution of the behavior (synchronization on demand)

3.3.2.4.5 It must be possible to mix masters using the standard temporal decoupling mechanism with regular masters, which are unaware of temporal decoupling.

3.3.3 Efficient extensibility (2.0)

When the Universal MMB needs to be extended, this shall be achievable with high simulation speed.

3.4 Completeness

3.4.1 Requirements for completeness

3.4.1.1 Non-intrusive transactions (2.0)

Provide a mechanism by which a debugger connected to a master can initiate a non-intrusive debug transaction requests through the system.
Transactions are limited to those already supported by the existing data structures, only that they are non-invasive in this case. Transfer of more advanced debug information is not covered. (see 2.1)

3.4.1.2 Visibility (2.0)

The API must include support for analysis ports, which give the user visibility into the system. (see 2.2, 2.3, 2.4, 2.5)

3.4.1.2.1 The analysis mechanism must be non intrusive from a user point of view

3.4.1.2.2 The analysis mechanism must be useful for tasks like functional coverage and scoreboarding.

3.4.1.2.3 The analysis mechanism must be useful for tasks like transaction recording, performance analysis and transaction level assertions

For functional coverage we only need the transaction itself, but for the others we need start and end times.

3.4.1.2.4 The analysis mechanism must work when there is no channel.

Practically speaking, this means that the analysis mechanism must be attached to ports and exports.

3.4.1.2.5 The analysis mechanism must be non-blocking.

3.4.1.2.6 Protection : the analysis mechanism is a snapshot of the transaction at that particular time. There must be some mechanism or other to ensure that subsequent changes to the transaction do not affect the analysis process.

3.4.1.2.7 The analysis mechanism must be robust in the face of both pipelining and multiports.

3.4.1.2.8 It must be possible to turn various "analyzers" on and off at any time during the simulation.

3.4.1.2.9 Out of module binding

It must be possible to attach an analyzer which exists anywhere in the hierarchy to an analyzed object anywhere else in the hierarchy.
The primary use case here is where we receive a design but are not allowed to tamper with it in any way, but we still want to add analysis probes of various kinds to it.

3.4.1.2.10 Hierarchy aware

If we attach an analyzer to a port or export high up in the hierarchy, this port or export must understand the transactions taking place between the lower level ports and exports that are bound to it

3.4.1.3 Ability to transfer Memory Map information (not in 2.0. Investigated for 2.1)

Certain simulation artifacts require memory map information to be transferred from one component to another.
Transfer of such information from master to slave and slave to master must be possible at runtime.

3.4.2 Considered Implementation Options

- Debug mode of MMB transaction
- Analysis ports

4 TLM Implementation Requirements for the support of Architecture Analysis

This section summarizes the requirements for the creation of TLM platforms, which can be used for Architectural Analysis. Again the considered and discarded implementation options for the requirements are listed.

4.1 Interoperability (2.0)

The goal is that users can develop interoperable models for architecture exploration of SoC platforms independently, using only OSCI documentation.

4.1.1 Interoperability Requirements

The idea behind this list of requirements is that there be a single API defined for a set of typical attributes of common MMB protocols. This would permit interoperability of models at both the loosely-timed and approximately-timed level, but only where such a functional abstraction was appropriate. In its un-extended form, this API would typically be inadequate for high-fidelity models of specific MMB protocols such as might be provided by IP vendors.

4.1.1.1 Interoperable Memory Mapped Bus API (2.0)

The goal is to define a highly interoperable, approximately timed TLM API for the creation of high-level performance models of SoC platforms, which are based on memory-mapped busses. The idea behind this requirement is that there be a single API defined which can model most MMB protocols at a timing-approximate level. Interoperability can be accomplished by

- using the proposed API and data structures, or
- using adaptors from proprietary TLM to the proposed API and structure.

The timing approximate TLM API is assumed to cover only the view from the module: the module has a socket, typically with address bus, data bus and control lines, and sees the rest of the system as a single external entity on the other end of these busses. Sockets are either masters or slaves.

4.1.1.2 Bridges between modules should be avoided as much as possible (2.0)

It is desirable to avoid the need for bridges (adaptors) as far as possible between models for different protocols (e.g. AXI and OCP) or which use different features of the same protocol (e.g. slaves that do not support burst mode). In some cases, bridges are unavoidable, i.e. between detailed models of fundamentally incompatible protocols.

4.1.1.3 Generic data structure for the Approx-timed Generic Memory-Mapped Bus (2.0)

We need a generic transaction data structure that works well for the majority of memory-mapped busses and is able to represent transaction timing.

4.1.1.4 Compatibility with Loosely-timed (2.0)

Cost in simulation performance of bridging from a loosely-timed Generic bus to an approx-timed Generic bus must be as small as possible

- By implication: no need to deep-copy any structures passed by pointer
- By implication: use of same data structures as in loosely-timed (normally loosely-timed will use a subset of those used by approx-timed)
- On-the-fly switching of an interface between loosely-timed and approx-timed must be possible.
- But: possible to create an approx-timed-only module (no requirement that all modules support loosely-timed and approx timed APIs in parallel)

4.1.1.5 Compatibility with future accurate-timed MMB API (research)

The penalty in simulation performance of bridging from an approx-timed Generic bus to some accurate-timed Generic bus must be as small as possible. Note that this requirement is rather difficult to enforce given the lack of such an API to test against. But some consideration of how it might look is warranted.

4.1.2 Discarded Interoperability Requirements

4.1.2.1 Timing Accuracy

The creation of 100% cycle accurate models is not the goal of the approximately timed TLM API. Cycle accuracy is addressed by a to-be-defined cycle accurate TLM API, which is not in the scope of TLM 2.x.
Of course it is possible (and in fact desirable), that the approximately timed TLM API achieves 100% accuracy for a certain bus protocol or a certain set of stimuli.

4.1.3 Considered Implementation Options

- Approximately timed TLM API is based on non-blocking transport API
- The data structure uses a native 64 bit type for the address and an unsigned char* for the data

4.1.4 Discarded Implementation Options

- Approximately timed TLM API is based on the uni-directional put/get API
- The data structure is templated with address and data types

4.2 Timing Accuracy

The definition of an interoperable approximately timed TLM API poses an even bigger challenge than the loosely timed TLM API. The goal is to define the common denominator, which captures the performance aspects of the common bus protocols. The resulting performance measurements from the approximately timed TLM platform model must be sufficiently accurate to take design decisions (see 2.3 and 2.4).

4.2.1 Timing Accuracy requirements

4.2.1.1 Relevant performance metrics reach x% accuracy for typical stimuli and typical memory-mapped buses

- Typical performance metrics are latency, throughput, utilization, contention, etc.
- Comparing a approximately-timed model and a corresponding cycle accurate model the error in the considered measurement interval should be smaller than (100-x)%.
  FIXME: define x
- Considered typical memory mapped buses are APB, AHB, AXI, OCP, CoreConnect
- Typical stimuli is a rather vague term. It basically means that one can easily create scenarios where the accuracy requirements cannot be met using the timing approximate TLM API.

In the end, this is the primary requirement to serve the purpose of the addressed use-cases (see 2.3 and 2.3). The following requirements can be seen as secondary requirements to achieve this goal.

4.2.1.2 Ability to model burst structures

Supports a wide range of burst types natively

- Incr, wrap, private, block, …
- SRMD and MRMD
- Precise (fully known at the start) and imprecise (created on-the-fly)

4.2.1.3 Ability to model multiple outstanding transactions

This corresponds to the pipelining of requests and responds, i.e. an initiator can start the next transaction before the previous transaction has finished.

4.2.1.4 Ability to model out-of-order transactions

4.2.2 Discarded Timing Accuracy requirements

4.2.2.1 Ability to model individual phases of a transaction

For the purpose of high-level performance analysis it is not required to model, e.g. the pipelining of address and data-phases in AHB or the data-handshake phases in OCP.
FIXME: this is probably a debatable requirement

4.2.3 Considered Implementation Options

4.2.3.1 Moments in time

In order to meet the timing accuracy requirements in 4.2.1, the timing approximate TLM API needs to be able to model the moments in the lifetime of a transaction like the following:

1. Time Initialization started on the bus.
2. Time Initialization completed on the bus.
3. Time first request asserted on the bus.
4. Time last request is accepted on the bus.
5. Time first response asserted on the bus.

6. Time last response accepted on the bus.

4.2.4 Discarded Implementation Options

4.3 Speed

4.3.1.1 Speed requirements for the use-cases addressed by the approximately timed TLM API (2.0)

We consider the following numbers to be lower boundaries to render a TLM model useful for the respective use-case.

- SW Architecture Analysis: 1 MT/s
- HW Architecture Analysis: 1 MT/s

4.4 Completeness

Supports all functional features of the loosely-timed Generic MMB

4.4.1.1 Analysis of performance metrics

It must be possible to measure the performance metrics mentioned above. This boils down to observing the moments in the lifetime of transactions as defined by the timing approximate TLM API.

4.4.2 Ability to switch to loosely-timed mode (not 2.0)

It must be possible to switch the simulation from approx timing to loosely-timed mode.

5 Modeling Efficiency (2.0)

These requirements are very subjective, and not easy to measure. These requirements refer to the effort required to understand the standard and code following it.

5.1 Conceptual simplicity

The working group needs to convince themselves that the underlying concepts are clean, as orthogonal as possible and easy to explain. Not being able to create collateral material that concisely explains the fundamental concepts in an intelligible way signals that we did not fulfill this requirement. Inability to explain the concepts renders a standard unusable.

5.2 Communication consistency

The number of bindings between communicating modules should be kept to a minimum. The number and use of ports in the module should resemble the hardware representation of the modeled component.

5.3 Ease of use

This requirement boils down to "*one should not have to bend over backwards to write models taking advantage of the full range of OSCI TLM features*". That is, while it is obvious that supporting individual requirements listed in this specification needs to be

feasible in a reasonably easy-to-use fashion, we also need to consider use models where multiple requirements play a role.

The APIs and use models must be simple enough to understand and employ properly that diverse practitioners of normal skill can reasonably be expected to produce models that interoperate as expected when combined in systems.

5.4 Compact code, expressive power

We can take the number of lines of code required to code models of masters and slaves as a measure of ease of use and expressive power.

5.5 Safety and robustness

Coding styles that are more amenable to pitfalls result in higher development and maintenance cost. It is hard to quantify this requirement, though. One possible metrics is to look at the amount of "*additional*" code required to make a model safe in all possible use cases.
It must be possible to build convenience functions that are safe to use.

6 Generic TLM API (2.0)

Generic TLM APIs that are based on user-defined templates, to be used when the suggested data structures fail to properly represent a MMB.

7 Low-Level data type recommendations (2.0)

A list of low-level data types to be used when user-defined templates are required; in the hopes of minimizing work if/when transactors are required.

8 Quality of Proof-of-Concept Implementation (2.0)

8.1 All source code files need to have proper copyright notice

8.2 At least all header files and examples need to have meaningful comments

- Description of APIs (parameters, return values, restrictions, assumptions)
- Description of relevant portions of examples

8.3 Use of doxygen-style comments would be nice but isn't a must-have

8.4 Each feature needs to be covered by a regression test

8.5 Examples and regressions are Purify clean

8.6 No compiler errors and warnings (unless explicitly waived)

8.7 We need examples demonstrating that we fulfill the requirements

This doesn't imply that the examples should be structured along the list of requirements. There should be better ways to organize our collateral material.

8.8 The benchmark example(s) should be shipped (as part of regressions/examples)

8.9 Consistent use of Unix file format (no DOS CR/LF, please)

8.10 Don't open namespaces in header files

8.11 No junk files in the distribution

8.12 Use of #include guards in all header files

9 Documentation (2.0)

9.1 LRM

LRM strength documentation is required for normal releases

9.2 Whitepaper

A technology whitepaper covering all major API features is required for community review packages.

9.3 Design (w/UML) & Rationale

9.4 Examples walkthroughs

Walkthroughs are required for community review packages

10 Porting (2.0)

10.1 Community review packages need to work at least on Windows and Linux

10.2 Normal releases need to be ported to the standard range of platforms.

At the time of release, all the code included in the kit should compile in the same set of OS/Compiler configurations as the OSCI simulation kernel.

10.3 SystemC 2.2 simulation library

Code included in the kit should compile against OSCI SystemC 2.2 library.

11 Backward Compatible to existing OSCI TLM 1.0 IP (2.0)

We should preserve (i.e. not deprecate any parts of) TLM 1.0. It may continue to be used in its original form even after the availability of a TLM 2.0 standard (e.g. for verification). The transport layer (a.k.a. TLM 1.0) may change; for example, if we add nb_transport(). We should document how to migrate TLM 1.0 models to TLM 2.0.

12 Must not create obstacles for modeling off-chip communication (2.0)

or non-MMB based links in general.

13 Multi-port target support (2.0)

Need to support target models with several target ports, and discriminate incoming path (i.e. port)

14 Hierarchical models support (2.0)

Port-to-port connection on the initiator side, port-to-export at the top, export-to-export on the target side.

15 Separation of concerns (2.0)

Models written in a loosely time style should not be contaminated by approx-timed considerations

*16 Systematic support of both PV and PVT features in a model is *not* a requirement (2.0)*

17 Models in object code (2.0)

It must be possible to distribute models without having to include the source code for the model. The model header file and compiled object file must be sufficient.

18 Clarifications regarding the Interoperability Requirements

This section makes a number of clarifications concerning the intended interpretation of the requirements for interoperability given in this document above.

18.1 Standard Interfaces

OSCI should define standard interface classes, transport mechanisms, and timing mechanisms for untimed, loosely timed and approximately timed modeling, independent of the details of any specific payload, and usable with both the Generic Payload (see below) and user-defined payloads.

18.2 Generic Payload

OSCI should define a typical MMB payload, known as the "Generic Payload", with precise semantics to support out-of-the-box interoperability. It should be possible to create interoperable models using the Generic Payload using only the classes and documentation supplied by OSCI, and without the need for adaptors. "Generic Payload" means typical attributes common to most MMBs and applicable at the functional level, abstracted away from the details of any specific bus protocol.

18.2.1 Limitations

The Generic Payload is *not* sufficient by itself to model any and every specific MMB protocol at a detailed level, and hence is *not* necessarily sufficient to provide interoperability between models of specific MMB protcols.

18.2.2 Draft 1 kit

In order to define a suitable set of attributes for the Generic Payload, the attributes in the TLM2.0 Draft 1 kit should be used as a starting point.

18.3 Extension Mechanism

OSCI should define an extension mechanism to the Generic Payload to allow the modeling of other specific protocols.

18.3.1 Inheritance and Composition

Regardless of whether or not OSCI provide an explicit extension mechanism, the Generic Payload could still be extended using inheritance or composition, and models connected using adaptors. OSCI should give definitive guidance on this matter (e.g. “Yes, that’s okay, we recommend you do it like this ...” or “No, that’s not recommended.”)

18.3.2 Ignorable Extensions

There may or may not be a distinction to be drawn between the mechanism for extending the Generic Payload to describe other specific protocols and the mechanism for adding ignorable attributes to the Generic Payload (to avoid the need for adaptors). This is regarded as an implementation detail to-be-decided.

18.4 Specific Protocols

The modeling of specific protocols (AXI, PLB etc) is the responsibility of the owners of those protocols, not of OSCI (but see sub-clause 7 below).

18.5 Guidelines

The TLM 2.x standard could include guidelines on how to use the mechanisms provided in sub-clause 1 above to model common bus attributes (whether or not those attributes are part of the Generic Payload). This is **not** to say that the TLM 2.x standard should include a catalog of attributes, merely that it could give guidelines on how best to approach the modeling of certain kinds of attribute using the mechanisms provided, e.g. how to deal with mutable fields such as addresses passing through routers, how to code byte enables, how to distinguish between transactions arriving on multiple exports (the tlm_export_id feature from the Draft 1 kit) etc.

18.6 Generic Payload and Adaptors

As described in sub-clause 2 above, one purpose of the generic payload is to support connection without adaptors.

18.6.1 Generic Payload – to – Generic Payload

Where a port and an export both use a TLM2 core interface with an identical instantiation of the generic payload class template, they can be connected without an adaptor. The extent to which the generic payload class template may be parameterized is detail to-be-decided.

18.6.2 Generic Payload – to – Specific Protocol

Connection without adaptors between the Generic Payload and any other specific protocol is not a requirement. It would only be possible where the specific protocol was modeled using only the Generic Payload and ignorable extensions.

18.7 Interoperability for Specific Protocols

It is not the responsibility of OSCI to ease the task of achieving interoperability between models of specific protocols (e.g. AXI and PCI) over and above defining some machinery (sub-clause 1) and some coding guidelines (sub-clause 5).

18.7.1 Generic Protocol as a Base

It is the intention that models of specific protocols should be based on the Generic Payload (sub-clause 2) wherever possible, although this is not a strict requirement. The intent is to improve productivity and consistency, to simplify adaptors between protocols, and to maximize the simulation speed of adaptors.

18.7.2 Repository of Extensions

An external repository of extensions is not an OSCI requirement, and would be outside the scope of the TLM-WG.

18.8 Approximately-Timed Models

For approximately timed modeling the Generic Payload will be identical to that for untimed and loosely timed modeling, but approximately timed models will make use of a wider set of attributes.

18.9 Cycle-Accurate Modeling

At some point in the future beyond TLM 2.0, OSCI should define interfaces and mechanisms for cycle accurate modeling.

18.10 TLM 1.0

If the TLM 2.x implementation diverges from that of TLM1.0 and TLM2.0 draft 1, then issues of compatibility and interoperability between TLM1 and TLM2 need to be considered and resolved.

18.10.1 Backward compatibility

There is an assumption that the TLM1.0 standard will continue to be maintained in its own right, but there is no assumption of automatic compatibility between TLM1.0 and TLM2. In other words, there is no assumption that TLM2 transactions can pass through TLM1.0 interfaces without the code developer needing to take some kind of corrective action.

Requirements Summary