Transaction Level Modeling using OSCI TLM 2.0

By Marcelo Montoreano, Synopsys, Inc.

May 31, 2007

1. Introduction

Currently, Transaction Level Modeling is being used in the industry to solve a variety of practical problems during the design, development and deployment of electronic systems. These problems include:

- Providing an early platform for software development.
- Aiding software/hardware integration.
- Enabling software performance analysis.
- System Level Design architecture analysis.
- Functional hardware verification.

As the level of abstraction becomes higher, the current OSCI TLM 1.0 standard becomes less applicable and not fast enough for the task.

Another issue ailing the current approach is the lack of model interoperability, where different vendors create models to be used by a common customer; it is up to the latter to interface the components so they talk to each other.

The main objective for the TLM 2.0 standard is to solve these problems while defining a solid API and suggested data structures that, when used as proposed, enable model interoperability. The intention is that two models written by different people, without any knowledge about each other, when written according to the standard, will be interoperable.

The API is based on templates, providing a generic framework to be used in case the proposed data structures prove unworkable for the protocol being modeled. In that case, interoperability will suffer and adaptors will be required to interface with standard TLM 2.0-based components.

The standard is being worked out in phases. The current roadmap is as follows:

TLM 2.0:

- Generic TLM APIs and data structures for transaction execution
- Interoperable Memory Mapped Bus (API + data + protocol semantics) for loosely-timed and approx-timed coding styles
- Support for non-intrusive/debug transactions
- Support for unobtrusively monitoring or probing transaction activity (Analysis ports)
- Recommendations on common data-types

TLM 2.0 stretch:

- Direct-memory interface
- Model synchronization

After TLM 2.0:

- PV Synchronization and Interrupts
- Temporal Decoupling
- Model-model Memory Map transfer
- Language Reference Manual
- Cycle accurate coding style
- More General (System/Component) Debug APIs
- More General Configuration APIs to Include Memory Map
- Profiling APIs
- Hardware Watchpoints
- Registers/memories

This document provides a high-level description of the TLM 2.0. For a more detailed description, please consult the TLM 2.0 Requirements document. If this document diverges or contradicts the Requirements document, the latter takes precedence.

2. OSCI TLM 2.0 proposal

Different use models of Transaction Level simulations require different levels of timing accuracy and resource contention modeling. Software development, for example, can ignore most if not all resource contention issues, assuming that the hardware will sort it out. That is, the code path the software takes doesn't generally depend on any resource contention that the CPU transactions may be subjected to.

On the other extreme, hardware performance analysis requires that contention be fully modeled for the component of interest, and maybe even some surrounding components.

TLM 2.0 defines two coding styles for Transaction-Level models, depending on the timing-to-data model dependency.

2.1. API Architecture

The TLM 2.0 API is constructed as follows

- 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 model a "Generic" memory-mapped bus.

The 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.

2.2. OSCI TLM 2.0 Model coding styles

Models can be separated into two coding styles depending on the timing-to-data dependency that they must obey. Sometimes these coding styles are confused with "levels of abstraction", but the presented coding styles can both be used in very detailed models or models that include little detail of the modeled component.

2.2.1. Loosely-timed models

Sometimes called PV models, these models have a loose dependency between timing and data, and are able to provide timing information and the requested data at the point when a transaction is being initiated.

These models do not depend on the advancement of time to be able to produce a response. Normally, resource contention and arbitration are not modeled using this style.

Due to the limited dependencies and minimal context switches, these models can be made to run the fastest and are particularly useful for doing software development on a Virtual Platform. Reaching simulation speeds of 50 M Transactions per second allows software developers to boot an OS and run test code in seconds.

2.2.2. Approximately-timed models

These models have a much stronger dependency between timing and data. They are not able to provide timing information and/or the requested data when a transaction is being initiated.

These models can depend on internal/external events firing and/or time advancing before they can provide a response. Resource contention and arbitration can be modeled easily with this style.

Since these models must synchronize/order the transactions before processing them, they are forced to trigger multiple context switches in the simulation, resulting in performance penalties.

2.2.3. Mixes of models

In some cases, it is desirable to create models that can behave in either style, deferring the decision of which to use as a run-time decision. Models of busses and arbitration are particularly useful when coded in this hybrid style.

This enables execution of the simulation at high speed with limited accuracy, while allowing the user to switch to a more accurate mode for further inspection.

To enable maximal reuse of models and enable functional HW verification, it is necessary to be able to mix models of both styles.

The TLM 2.0 allows this by sharing API and data structures across the coding styles. Models written in one style can be directly connected to models written in the other, without the need of adaptors.

Models can be easily and gradually refined from loosely-timed to approx-timed.

Conversely, we are also able to take advantage of available approx-timed models for software development.

2.3. Compatibility with TLM 1.0

We will not deprecate any parts of TLM 1.0. It may continue to be used in its original form after the availability of a TLM 2.0 standard.

3. TLM 2.0 Recommended Methodology

3.1. Developing new components

The following methodology is recommended when developing components for the TLM 2.0 standard:

- Start with creating a loosely-timed model of the component, even if the final model will be approximately-timed.
  It should include:
  - Component functionality
  - Debug functionality
  - Memory map functionality

  This initial model can be used to kick-start target software development, functional verification, and system-model assembly.
- Add timing details to the loosely-timed model, modeling contention if applicable.
  Not all models need to be refined to Approximately-timed style, only those in the path of interest.

3.2. Adapting existing components

The complexity of adapting existing components greatly depends on the coding style of the original component. The common case is probably cycle-driven models. For those, a bridge needs to be written that takes an incoming transaction and passes it through to the model. Part of the bridge should clock the model as SystemC time advances and notify an event once a response is received and translated to the TLM 2.0 data structures.

4. Details and Areas under discussion

4.1. Pass by pointer vs. pass by value

Passing transaction data by pointer minimizes data copying and maximizes speed. Transaction data should be passed by pointer and respecting the ownership rules that define the lifetime of the pointers.

4.2. Direct Memory Interface

Providing models direct access to another model’s memory storage bypasses function calls, allowing faster simulation execution.

4.3. Temporal decoupling of modules

Allowing models to run ahead and then wait for the system to catch up exploits host machine cache and minimizes context switches

4.4. On the fly switching to more accurate mode

To enable SW performance analysis at an acceptable speed, TLM 2.0 recommends that most high-traffic components that require an approximate-timed style be coded in the “combination” style. This creates a simulation that can execute very fast during uninteresting parts (e.g. target SW decompression), and then switch to a more accurate mode (e.g. driver feeding HW accelerator).

4.5. Model synchronization

When dealing with loosely-timed systems, it is necessary to enforce model synchronization to make the model advance time in a predictable manner. TLM 2.0 provides means for model synchronization.

4.6. Blocking in the master vs. Blocking in the slave

Approx-timed models cannot complete transactions at initiation time, as they depend of other parts system to do so. Whether the transaction call into the model should be blocking or non-blocking, or both styles supported, has not been decided.

4.7. Extension mechanisms

Although there is consensus that extensions to the data structure would add flexibility to the standard, it seems that there is no practical extension mechanism that can handle the mix of optional and must-have extended attributes that are likely to be required.

4.8. Moments in time

Six “moments in time” seem to be enough to represent the phases of most protocols. However, the way these “moments” will be communicated has not been decided.

5. Summary

TLM 2.0 provides a standardized approach for creating models and transaction-level simulations. The standard enables exchange of models and a common ground for interfacing.

A simple but solid architecture allows TLM newcomers to quickly get up to speed and produce interoperable models. For seasoned TLMers, the standard provides a solution for the increasingly hard problem of model interoperability.