What's New in TLM-2.0.1?

John Aynsley, Doulos, July 2009

The definitive OSCI TLM-2.0 Language Reference Manual was released 27-July-2009 along with version TLM-2.0.1 of the release kit. In this article we take a look at the new features of TLM-2.0.1 as documented in the LRM.

Well, the source code has not changed much since TLM-2.0.0. There have been some bug fixes and minor enhancements, and some improvements to the unit tests and examples, but that's about it. The TLM-2.0.1 code is backward-compatible with TLM-2.0.0 in all respects that you are likely to notice or care about. The biggest changes are in the documentation. The User Manual from TLM-2.0.0 has been promoted to a definitive Language Reference Manual (LRM), and now contains many more rules, descriptions and explanations. The purpose of these rules is to ensure interoperability between TLM-2.0-compliant models. This article discusses the issues that have been clarified in the TLM-2.0 LRM, as well as presenting the minor changes in the TLM-2.0.1 release kit.

In case you were wondering, TLM-2.0 is not a "language" as such. The "language" being referred to is SystemC. TLM-2.0 builds directly on SystemC. TLM-2.0 is the name of the standard, whereas 2.0.0 and 2.0.1 are version numbers of the software distribution.

Most of the content of this article is rather technical and is aimed at TLM experts rather than beginners. For experts, this article adds further commentary and explanation going beyond that contained in the TLM-2.0 Language Reference Manual.

Topics discussed in this Document

Protocol Types and Extensions

The thinking of the TLM Working Group on the base protocol has evolved since the TLM-2.0 standard was first published in June 2008, particularly in the area of extending the base protocol to describe other specific protocols.

The Overall Approach

Each connection between TLM-2.0 components can either use the base protocol or not. The term *base protocol* refers to a specific traits class **tlm_base_protocol_types** and its associated rules, which effectively means that the components must communicate using the classes of the so-called interoperability layer of TLM-2.0, which include:

- The TLM-2.0 core interfaces, namely, the transport, DMI and debug transport interfaces
- The standard sockets **tlm_initiator_socket** and **tlm_target_socket** (or classes derived from these)
- The generic payload class **tlm_generic_payload**
- The phase class **tlm_phase**

The base protocol allows both the loosely-timed (LT) and the approximately-timed (AT) coding styles to be used, though these should be thought of as points along a spectrum of possible coding styles allowed by the base protocol rather than being definitive in themselves.

Use of the base protocol is signified by using the name of the base protocol traits class **tlm_base_protocol_types** to specialize one of the two standard socket classes **tlm_initiator_socket** and **tlm_target_socket**, for example:

```
tlm::tlm_initiator_socket<32,tlm::tlm_base_protocol_types> my_socket;
```

A *traits* class is a class that typically provides a set of typedefs used to specialize the behavior of another class, usually by having the traits class passed as a template argument to that other class (as shown here). A TLM-2.0 protocol traits class must provide two typedefs **tlm_payload_type** and **tlm_phase_type**.

Any component using the name of the traits class in this way is obliged to honor all of the rules associated with the base protocol. Most of these rules are just conventions and are not actually enforced by the software API. Nonetheless, it is only by complying with these rules that TLM-2.0 components are able to achieve interoperability. The base procotol rules have been considerably refined in the LRM, and some of the important clarifications are discussed below.

The alternative is *not* to use the base protocol, signified by using the name of some other protocol traits class when specializing the standard sockets, for example

```
tlm::tlm_initiator_socket<32,tlm::my_protocol> my_socket;
```

When creating your own protocol traits class, you make the rules associated with that protocol, and there are no *a priori* limitations on the form those rules might take. For example, you could create a minor variation on the base protocol, perhaps adding an extension to the generic payload and a couple of extra phases, or you could restrict the interface method calls through the socket to

b_transport, or you could replace the generic payload with an entirely new transaction type. You could even have a single protocol traits class that represents a whole family of related protocols, with the precise details of the protocol being settled by negotiation at run-time. The point is that when you define the protocol traits class, you make the rules. The only rule imposed by TLM-2.0 concerning user-defined traits classes is that you cannot directly bind together two sockets specialized with different protocol traits classes; you have to write an adapter or bridge between them.

But having said "you make the rules", TLM-2.0 sets a clear expectation. When defining a new protocol traits class, you are strongly recommended to stick as closely as possible to the rules of the base protocol in order to reduce the cost of achieving interoperability between different protocols. If you need to model the specific details of a given protocol then of course your model will depart from the base protocol. But remembering that the objective is fast simulation speed through transaction-level modeling, it will often be possible to abstract away from the details of a specific protocol and end up with something not too far removed from the base protocol.

Think of the TLM-2.0 base protocol as being two things: it is both an abstract memory-mapped bus protocol in its own right and also an exemplar for the transaction-level modeling of specific protocols.

In summary, if you use the base protocol, you have to keep to the rules of the base protocol, which are quite strictly defined. If you create your own protocol you can make your own rules, but you lose the ability to bind your sockets to base protocol sockets, thereby limiting interoperability. So a natural question to ask is whether you can maximize interoperability by mapping your particular protocol onto the base protocol, or whether you need to define a new protocol traits class. In other words, how far can you stretch the base protocol without breaking it?

Ignorable Extensions

The base protocol can be stretched in two ways without any need to define a new traits class: by adding ignorable extensions to the generic payload, and by adding ignorable phases to **tlm_phase**. An extension or extended phase is said to be *ignorable* if any and every component other than its creator is permitted to behave as if the extension were absent, regardless of whether or not that or any other component recognizes or understands the extension.

Again, there are no *a priori* limitations on the semantics associated with ignorable extensions. For example, an ignorable extension might modify the behavior of a READ or WRITE command to fail if the initiator does not have the appropriate privilege level, thereby altering the behavior of the standard generic payload command attribute. The point is that a component on the receiving end of the extended transaction must be allowed to behave as if the extension did not exist, in other words, to ignore the extension entirely and perform a plain READ or WRITE. If the target were actually obliged to interpret and honor the extension, a new protocol traits class would be necessary.

One point that emerges from this discussion is that *ignorable* is not quite the opposite of *mandatory* when it comes to extensions. Rather, the two concepts are somewhat independent. Nonetheless, a base protocol component is forbidden from generating an error response due solely to the absence

of an extension. In this sense, an ignorable extension cannot be mandatory. The base protocol does not require the existence of any extensions.

The definition of "ignorable" applies equally to extensions and to phases. An ignorable phase can be totally ignored by a receiving component (whether it be upstream or downstream of the sender), and there is no way (within the base protocol) for the sender to know that the phase has been ignored other than by inferring the fact from the absence of any explicit response. On the other hand, suppose that the ignorable phase is not ignored, but instead the receiver sends back another ignorable phase in the opposite direction. Using extensions, a set of base protocol components could negotiate between themselves to use an extended set of phases. Such a protocol negotiation would be outside the scope of the base protocol, but at the same time would be permitted by the base protocol just so long as any connected component is allowed to *ignore all extensions and extended phases entirely* and communicate according to the strict rules of the base protocol alone. In other words, any extended base protocol component must have a *fall back* mode where it reverts to the unextended behavior of the base protocol.

To maintain interoperability, there are some rules on how ignorable phases must be handled within the base protocol. A base protocol component that does not recognize a particular extended phase must not propagate that phase (i.e. pass it further along the call chain between initiator and target) except where that component is a so-called *transparent component*, meaning that it is a dumb checker or monitor component with one target socket and one initiator socket that simply passes on all transactions. Transparent components are expected to propagate every phase immediately, ignorable phases included. There is a further base protocol restriction that ignorable phases must not be sent before BEGIN_REQ or after END_RESP, that is, outside the normal lifetime of an unextended base protocol transaction.

As usual, any deviation from these interoperability rules would require you to define a new protocol traits class and use it to specialize your initiator and target sockets.

Base Protocol Rules

We discussed above how the base protocol rules only strictly apply where the traits class **tlm_base_protocol_types** is used, and otherwise serve as recommended guidelines for creating transaction-level models of other specific protocols. Again, the thinking of the TLM-WG has moved on a little when it comes to modeling specific protocols at the approximately-timed (AT) level. Now we will look at some of those rules in detail.

Requests, Responses and Causality

The lifetime of a base protocol transport transaction, sent using either the blocking or the non-blocking transport interface, is structured into a request that propagates from initiator to target and a response that propagates from target back to initiator, passing through zero or more interconnect components along the way. A transaction is represented by one transaction object, which is passed forward and backward between initiator and target as a reference argument to a series of interface method calls. This is true of both the blocking and non-blocking transport interfaces. There are a number of events that must occur in a tightly prescribed order during the lifetime of the transaction:

1. The initiator must initialize the attributes of the transaction object before sending the request using an interface method call, and must not subsequently modify the attributes (extensions excluded)
2. An interconnect component can only forward the request after having received the request from the initiator or from an upstream interconnect
3. An interconnect component is permitted to modify the address attribute, but only before sending the request forward using an interface method call
4. The target is permitted to update the response status and DMI allowed attributes (and data array for a READ command) at any point between first receiving the request and sending the response, but must not modify any attribute thereafter (extensions excluded)
5. An interconnect component can only send the response back toward the initiator after having received the response from the target or from a downstream interconnect
6. The initiator must not inspect the response status (or data array for a READ command) until it has received the response.
7. The transaction attributes remain valid only until the end-of-life of the transaction. A component must not attempt to access the tranaction object after having called the **release** method.

In summary:

```
Initiator sets attributes -> interconnect modifies address        -> target copies data
Initiator checks response <- interconnect sends response upstream <- target sets response
```

It says "extensions excluded" above because the meaning of any extensions is by definition outside the scope of the base protocol.

These rules for modifying generic payload attributes also apply to the debug transport and direct memory interfaces where appropriate.

The construction and destruction of generic payload transaction objects can be prohibitively expensive (due to the implementation of the extension mechanism). Hence, you are strongly advised to pool or re-use transaction objects. In general the best way to do this is with a memory manager, but ad-hoc transaction object re-use is also possible when calling **b_transport**.

The roles of initiator, interconnect, and target can be assigned dynamically. In particular, a component can receive a transaction and inspect its attributes before deciding whether to play the role of an interconnect and propagate the transaction downstream, or to play the role of a target and return a response. Specifically, a component that is usually an interconnect plays the role of target when setting an error response.

The Response Status

A response status of TLM_OK_RESPONSE means that the transaction has been received and executed successfully at the target. Any other response status is regarded as an error response, including the default value of TLM_INCOMPLETE_RESPONSE, which in effect means that the transaction has not been received by a target capable of executing the command (and by inference, capable of failing to execute the command and thus setting an explicit error response status).

When setting an error response, a target has several values to choose from, and it may be that more than one error response value would be equally applicable. For example, a target that is unable to write to a given address could choose between TLM_COMMAND_ERROR_RESPONSE, TLM_ADDRESS_ERROR_RESPONSE, and TLM_GENERIC_ERROR_RESPONSE. Thus an initiator should not rely too heavily on the specific choice of error response. The intent is to use the error response to construct an informative message for the user.

A command attribute value of TLM_IGNORE_COMMAND means that the transaction is neither a read nor a write, and so a base protocol target is not obliged to take any action on receiving the transaction. The purpose of TLM_IGNORE_COMMAND is to provide a vehicle for transporting extensions, but, as we discussed above, the meaning of those extensions would be outside the scope of the base protocol. A target receiving a TLM_IGNORE_COMMAND is allowed to set the response status to TLM_OK_RESPONSE or to select an error response. For example, if the address were out-of-range the target might choose a TLM_ADDRESS_ERROR_RESPONSE. TLM_IGNORE_COMMAND is permitted by the transport and debug transport interfaces, but not for DMI.

END_REQ and END_RESP for AT Transactions

In the case of the non-blocking transport interface (as used for the AT coding style) and a WRITE command, the BEGIN_REQ phase nominally marks the time of the first beat of the data transfer, and END_REQ would typically mark the time of the last beat of the data transfer. Typically, but not necessarily; it is just a model. In the case of a READ command, it is BEGIN_RESP that marks the first beat of the data transfer, and END_RESP the last beat. If we focus on WRITE for a moment, BEGIN_REQ and END_REQ nominally mark the beginning and end of the time period during which the write channel is busy tranferring the data. Assuming there is a single write channel, it would not be possible to start the subsequent WRITE transfer until the current WRITE transfer has completed. This fact is represented in the base protocol by the request exclusion rule, which states that over a given hop, BEGIN_REQ cannot be sent until the previous END_REQ has been received. There is a similar response exclusion rule for BEGIN_RESP and END_RESP.

A hop is a single connection from an initiator socket to a target socket. A request may pass over one or more hops on its path from initiator to target, and the response must always return by exactly the same path. Whereas the lifetime of a transaction extends across multiple hops, the phase of the **nb_transport** call is local to a single hop such that different hops can be in different phases.

Whereas BEGIN_REQ and BEGIN_RESP must propagate end-to-end from initiator to target and back again, there is no such restriction on END_REQ and END_RESP. These phases are local to each hop, and serve only only to govern the flow control over each hop. A component can send END_REQ just as soon as it has received BEGIN_REQ, regardless of whether or not it has propagated the request to the downstream component.

The request and response exclusion rules are based on call order alone, irrespective of any timing annotations. For example, an **nb_transport_fw** call to send BEGIN_REQ with a large timing annotation would forbid any other BEGIN_REQ being sent over that hop until END_REQ had been received, regardless of whether or not the timing annotation had yet elapsed. For example:

```
// Pseudo-code
```

```
socket->nb_transport_fw(trans1, BEGIN_REQ, 1000ns);
socket->nb_transport_fw(trans2, BEGIN_REQ, 0ns);
  // 2nd call forbidden until END_REQ received
```

Only **nb_transport** has phases. **b_transport** does not have a phase argument, and so **b_transport** calls are not subject to request and response exclusion rules. Blocking transport calls do not interact with non-blocking transport calls when it comes to exclusion rules.

End-of-life of an AT Transaction

An AT transaction may be passed back-and-forth several times across several hops during its lifetime. In the case of a transaction with a memory manager, control over the the lifetime of the transaction object is distributed across the hops and is co-ordinated by the memory manager through calls to the **acquire** and **release** methods of the generic payload. The only method call provided by TLM-2.0 that co-incides with the end-of-life of a transaction is the call to the **free** method of class **tlm_mm_interface**, which occurs automatically when the transaction reference count reaches zero. In other words, if you wanted to execute some action at the end-of-life of the transaction, you would have to do so from the **free** method of the associated memory manager.

In the case of **nb_transport**, for each individual hop, the final activity associated with a transaction may be either the final phase of the transaction (passed as an argument to nb_transport) or TLM_COMPLETED (returned from nb_transport). For the base protocol, the final phase is END_RESP. Having **nb_transport** return TLM_COMPLETED is not mandatory. However, if TLM_COMPLETED is returned then no further activity associated with that transaction is allowed over that particular hop.

For the base protocol and **nb_transport**, certain phase transitions may be implicit:

- The BEGIN_RESP phase infers the existence of the END_REQ phase (if it has not already occurred)
- TLM_COMPLETED returned from the target infers both the END_REQ and BEGIN_RESP phases for that transaction.

Because TLM_COMPLETED infers BEGIN_RESP, returning TLM_COMPLETED would be disallowed by the response exclusion rule if there were already a response in progress over that particular hop.

Timing Annotation and Coding Styles

The rules for timing annotation have not changed, but the intent has been clarified a little. This is a tricky conceptual area to understand due to what it is trying to achieve; the reconciliation of two coding styles that represent differing timing accuracies into a single, interoperable API.

Each **b_transport** and **nb_transport** call takes a timing annotation argument, for example:

```
b_tranport(trans, delay);
```

The delay is added to the current simulation time to give the time at which the recipient should execute the transaction, that is,

```
effective local time = sc_time_stamp() + delay
```

The idea is that the recipient (whether the caller or the callee) should behave *as if* it had received the transaction at the effective local time, but it is up to each recipient to decide how to handle that intent, depending on its coding style. As you probably already know, TLM-2.0 describes two named coding styles, *loosely-timed* (LT) and *approximately-timed* (AT), which represent two points on a spectrum of possible coding styles.

Let's clarify some rules. Firstly, for a given transaction sent through a given socket, the effective local times must be strictly non-decreasing in all circumstances. This includes the timing annotation across the call to and the return from **b_transport**, as well as the timing annotations across multiple calls to **nb_transport**, but it does not include calls for distinct transactions or calls through different sockets.

Secondly, when multiple transactions are sent by an initiator through the same socket, their effective local times are expected to be non-decreasing, but this is an expectation and not a strict rule. The same expectation would apply when a buffered interconnect component is itself determining the order of the transations it sends on downstream (rather than just forwarding the transactions immediately it receives them). On the other hand, when an interconnect component merges two or more transaction streams and sends them on through the same socket, there is no expectation that their effective local times should be non-decreasing. Indeed, transactions from two different temporally decoupled initiators may well have unrelated effective local times. There are no constraints on the mutual order of transactions when transaction streams are merged or split.

Putting the above two rules together, whenever a downstream component receives two transactions with decreasing effective local times, it can infer that those transactions most probably originated from separate temporally decoupled initiators. It is fundamental to temporal decoupling that a downstream component can receive transactions out-of-order. Furthermore, the base protocol does not enable a downstream component to distinguish between transactions coming directly from an initiator and transactions arriving through one or more interconnect components.

So, we can now consider the main rule that allows coding styles to be mixed. Whenever a component receives incoming transactions where the interface method call order differs from the effective local time order (that is, the timing annotations are out-of-order with respect to the method calls), that component is free to chose the order in which it executes those transactions. In the case of a target, *execute* would mean execute the command as determined by the command attribute of the transaction. In the case of an interconnect, *execute* could mean perform arbitration, or store the transaction in a buffer, or route the transaction on downstream, for example. Out-of-order transactions would typically have originated from distinct temporally decoupled initiators, and so there should not be any immediate functional dependencies between them. It is because of this freedom of a downstream component to choose the order in which it executes out-or-order transactions that each initiator should generally send transactions in-order.

This brings us to coding styles. The LT coding style is focussed on simulation speed, so an LT component would typically execute incoming transactions immediately it receives them, in interface method call order, and would typically increase and pass on the timing annotations. In other words, an LT component would defer responsibility for implementing the timing annotation to other components, and would account for time merely by increasing the timing annotation. On the other hand, the AT coding style is focussed on timing accuracy (within a TLM framework), so an AT component would typically schedule incoming transactions for execution at the proper simulation

time, that is, to synchronize simulation time with the effective local time before executing the transaction. TLM-2.0 provides the payload event queue (PEQ) utilities for this purpose.

In summary, when effective local times are non-decreasing, transactions must be executed in-order. But when effective local times are decreasing, transactions may be executed in any order, the choice depending only on the tradeoff between simulation speed and timing accuracy as chosen by the recipient.

Transaction Ordering Rules

In general, it is only to be expected that an interconnect component may split, merge, or re-order transaction streams. For example, a memory controller may buffer incoming requests from multiple initiators and then send on requests to a fast memory in advance of requests to a slower memory. With the base protocol, incoming transaction streams arriving through different sockets (or even through the same socket in the case of **b_transport**) may be merged in any order, transaction streams sent out through different sockets may be mutually delayed, and responses may arrive back through different sockets out-of-order. There are, however, some constraints on the ordering of requests within a single transaction stream sent along a particular path.

The objective of the base protocol transaction ordering rules is to ensure that a sequence of requests issued by the same initiator and sent to the same target will be executed in-order at the target. For example, a sequence of reads and writes sent by a CPU to a given memory location should be executed in-order to avoid unexpected results. Because the base protocol provides this guarantee, a base protocol initiator can assume that transactions will be executed in-order without the need for extensions or special arrangements.

The first transaction ordering rule forbids degenerate routing. If an interconnect component receives several concurrent requests with overlapping address ranges, they must be sent forward through the same socket. In other words, all concurrent requests to the same address must be routed along exactly the same path. But note that reads and writes to the same address could be routed along different paths just so long as they are not active concurrently, and that the entire memory map could be changed from time-to-time, just so long as there were no active transactions at the time of the change.

The second transaction ordering rule ensures that requests do not overtake one another. If an interconnect component receives several concurrent requests through the same socket using **nb_transport** and those requests have overlapping address ranges, then the requests must be passed on in the same order they were received. Note that this assumes the effective local times are non-decreasing; otherwise, the interconnect component would be free to execute the transactions in any order anyway.

The third transaction ordering rule permits responses to be re-ordered on their way back from target to initiator. An initiator cannot assume that base protocol responses will be received back in the same order that the requests were sent, even in the case where the base protocol guarantees the execution order of the requests at the target.

b_transport is a separate case. The second rule above does not apply to **b_transport**, but then again, a given initiator thread is incapable of making a second call to **b_transport** until the first call has returned a response, so the issue of pipelined requests sent from the same initiator thread does not arise. The third rule above does permit **b_transport** responses to be re-ordered, but again the issue cannot arise from a single initiator thread. However, it is a rule that each thread process is treated as a separate initiator, so concurrent calls to **b_transport** through the same socket could arise if made by separate thread processes within a given initiator component.

DMI and Debug

The direct memory interface (DMI) has not changed, but a few points have been clarified in response to questions from users.

The first point is a rather obvious one; DMI by-passes the transport interface. Once a DMI pointer has been granted and is being used, any interconnect components are by-passed, and thus any side-effects of those interconnect components are by-passed too. As a consequence, DMI *cannot be used* in situations where interconnect components have significant side-effects, such as buffering or caching that affects the behavior of other parts of the system. It is your responsibility to decide whether it is acceptable to by-pass the behaviour of interconnect components before choosing to use DMI.

The second point is that the initiator is responsible for checking that the DMI region is still valid every time it uses a DMI pointer. The only way a DMI region could become invalid is due to a call to **invalidate_direct_mem_ptr** from the target. The initiator must clear some kind of *dmi_valid* flag from the body of **invalidate_direct_mem_ptr**, then check that flag before using the DMI pointer. However, bear in mind that SystemC is non-preemptive, so that once a temporally decoupled initiator thread starts executing, no other process will get to run until that initiator has yielded control. Hence, a temporally decoupled initiator only needs to check that DMI is valid when it resumes execution, and can then safely assume that DMI will remain valid until it yields control.

Thirdly, it is not permitted for a target to deny DMI access by setting the value of the DMI access type attribute to DMI_ACCESS_NONE (as opposed to read, write, or read/write). In order to deny DMI access, **get_direct_mem_ptr** must return **false**, in which case the attributes of the DMI descriptor are used to define the DMI region that is being denied. In other words, the DMI descriptor is used both when DMI is being granted and when it is being denied.

Finally, a minor clarification concerning the debug transport interface. It is possible to have a dummy debug transport command that does not actually read or write any data, either by having the command attribute set to TLM_IGNORE_COMMAND or by having the data length attribute set to 0 (or both). If the data length is 0, then the data pointer is allowed to be null. The point of such a dummy debug transaction is as a vehicle for extensions; the extended command may or may not make use of the data array.

TLM-2.0.1 and Backward Compatibility

Compatibility with SystemC versions

TLM-2.0.1 has been tested with SystemC 2.1.v1 and with SystemC 2.2.0. It has also been tested informally with various beta versions of SystemC 2.3.x, and works fine.

Removal of the Dependency on Boost

Certain parts of TLM-2.0.0, specifically the convenience sockets, made explicit use of the Boost library. This required Boost to be installed on each host computer in addition to the partial (and very old) Boost installation that comes as part of SystemC-2.2.0. This was a nuisance and often caused installation problems for beginners. All dependencies on Boost have been removed from TLM-2.0.1, which only requires that SystemC be installed.

Directory Structure

Compared to TLM-2.0.0, the directory structure of the TLM-2.0.1 release has been collapsed below **tlm_h** to remove the intermediate **tlm_trans** directory. Also, the directories **tlm_req_rsp** and **tlm_analysis** have been moved to a new top-level directory **tlm_1**, which now contains the TLM-1.0 include files and the analysis ports. These changes are backward-compatible with TLM-2.0.0. The user simply needs

```
#include "tlm.h"
```

to include the standard TLM-1.0 and TLM-2.0 headers, with the exception of the TLM-2.0 utilities, which must be included separately and explicitly. TLM-1.0 is still a current OSCI standard, although TLM-2.0 is in a sense independent of TLM-1.0.

Analysis ports were not part of the original TLM-1.0 release, but in some peoples' minds analysis ports are more closely associated with TLM-1.0 than they are with TLM-2.0. That said, analysis ports can be useful in TLM-2.0 for implementing transacton-level broadcast operations, such as might be needed to model interrupts.

Minor Changes and Bug Fixes in TLM-2.0.1

Generic Payload

The generic payload has a new method **update_original_from**, which is intended to complement the existing **deep_copy_from**.

```
class tlm_generic_payload {
  ...
  void deep_copy_from(const tlm_generic_payload & other);

  void update_original_from(const tlm_generic_payload & other,
                            bool use_byte_enable_on_read = true);
```

```
  ...
};
```

These two methods are intended for use in a component that bridges two protocols, where the transactions in each protocol are represented by separate TLM-2.0 transaction objects. In this case, the bridge component, which acts as a target for incoming transactions and initiates new transactions, may need to take a copy of an incoming transaction before decorating it with further extensions, and may need to perform the reverse operation on return, that is, copy the response status and any existing extensions back to the original transaction.

deep_copy_from copies all the attributes and extensions, but only copies the data and byte enable arrays if the pointers in both the *from* and the *to* transactions are non-null. **update_original_from** copies back the response status, the DMI hint, and extensions. It only copies the data array in the case of a READ command, and then only if both pointers are different and non-null. See the TLM-2.0 LRM for further details.

PEQs

The two PEQs **peq_with_get** and **peq_with_cb_and_phase** each have a new method with the following signature:

```
void cancel_all();
```

As the name suggests, this method cancels all outstanding events in the queue, leaving the PEQ empty. This can be useful for resetting the state of a model.

const

There have been minor changes to the signatures of some methods, in particular adding the **const** modifier in places where it was missing. In each case, this change allows the methods to be called in contexts where it was previously not permitted to call them. The new signatures are:

```
class tlm_generic_payload {
  ...
  int get_ref_count() const;
  bool has_mm() const;
  ...
};

class peq_with_get {
  ...
  void notify(transaction_type& trans, const sc_core::sc_time& t);
  ...
};

class peq_with_cb_and_phase {
  ...
  void notify (tlm_payload_type& t, const tlm_phase_type& p);
  void notify (tlm_payload_type& t, const tlm_phase_type& p, const sc_time&
when);
  ...
};
```

Standard TLM sockets

The standard TLM-2.0 sockets **tlm_initiator_socket** and **tlm_target_socket** now include a **kind** method with the following signature:

```
virtual const char* kind() const;
```

As with the standard SystemC classes derived from **sc_object**, this method returns a text string containing the class name.

Also, the **sc_object** names generated by the default constructors for these socket classes have changed slightly (although this is very unlikely to affect user-level code).

Convenience sockets

A bug has been fixed in the **simple_target_socket** nb-to-b adapter, which did not correctly implement timing annotations on incoming **nb_transport_fw** calls.

The constructors of the convenience sockets have been extended to provide two constructors, a default constructor and a second constructor that takes a **char*** argument, just like the standard sockets.

Multi-sockets

The multi-sockets now provide default implementation for the DMI and debug transport methods in the same way as the simple sockets, so there is no need to register every method. In TLM-2.0.0, it was necessary to register every method explicitly with the multi-sockets.

```
struct my_module ...
{
  tlm_utils::multi_passthrough_target_socket<my_module> socket;
  ...

  my_module(sc_module_name _n)
  : sc_module(_n)
  {
    socket.register_b_transport(this, &my_module::b_transport);
    // Defaults provided for the other three methods
  }

  void b_transport(int id, tlm_generic_payload& trans, sc_time& delay);
  ...
};
```

Endianness Conversion Functions

A bug has been fixed in the endianness conversion functions that prevented them from working on 64-bit hosts, and another bug that caused compilation errors with Microsoft compilers.

Quantum Keeper

A new method **set_and_sync** has been added to class **tlm_quantum keeper** with the following definition:

```
void set_and_sync(const sc_core::sc_time& t)
{
  set(t);
  if (need_sync())
    sync();
}
```

As you can see, the method updates the local time, checks whether synchronization is necessary, and if so, synchronizes. The method is provided just for convenience when executing this common sequence of operations.

Unit Tests and Examples

The source code, makefiles and project files of both the unit tests and the examples have been cleaned up such that they compile, run, and pass the regression checks on Linux, Microsoft and Sun host platforms, including little- and big-endian, 32- and 64-bit machines. The examples have been updated to eliminate issues identified by the Doulos and Jeda protocol checkers.

A new example **lt_mixed_endian** has been added, which illustrates the use of the endianness conversion functions when modeling a memory that is shared between little- and big-endian initiators.

Further unit tests have been added for the new **cancel_all** and **update_original_from** methods, for the endianness conversion functions, and for the nb-to-b adapter in the **simple_target_socket**. These unit tests serve as examples of how to use the respective features.