TLM 2.0 Draft2 Standard

STMicroelectronics feedback

AUTHORS

Author	Company
Jérôme CORNET/Laurent MAILLET-CONTOZ	STMicroelectronics

1 - Introduction

1.1 Purpose

The goal of this document is to formulate specific feedback from STMicroelectronics on TLM2.0 Draft2 Standard proposal, besides the feedback that ST provides as part of the SPRINT consortium (together with ARM, ST, Infineon and CoWare).

1.2 Scope

This document reports on the feedback expressed by ST during the public review period of TLM2.0 Draft2 proposal.

1.3 Structure

The document structure follows the structure of the TLM2 Draft2 user guide. Each section addresses one specific topic covered by TLM2D2. It covers general feedback, problem statement if any, and when appropriate, feedback on documentation

2 - Standard Introduction

2.1.1 General feedback

Documentation and examples:

- A significant effort has been put on the documentation, to try to explain the concepts integrated within the kit. However, some concepts remain unclear, probably due to intrinsic complexity.
- The set of examples released with the kit is not covering all the interfaces of the kit. In particular there is no example of the blocking transport interface, while this example is available internally to the working group from early December. It is difficult to run a fair review of the proposal, whereas examples are not or partially available.

Background:
This introductory part contains statements which are at best misleading:

- "*TLM1 models would typically implement delays by calling wait, which slows down simulation*": Independently of the technique used to implement timings, models have to ultimately call wait. This is true also for the technique using a Quantum Time keeper. Simulation performance depends on the number of calls to wait, which can be minimized using a technique or another.
- "*The TLM1 interfaces require all transaction objects and data to be passed by value or const-reference which slows down simulation*": The TLM1 interfaces permit any kind of data type to be transported, which includes pointers and references. These latter are similar to the way transactions objects are passed in TLM2, and do not slow down simulation.

As a remedial, we propose to rephrase these assertions.

Transaction-Level modelling, use case and abstraction:

- "*Process scheduling can be modeled in two ways either by having the user introduce explicit communication and synchronization points into their application, or by having a coding style or underlying modeling framework that forces each process to yield control at certain points. The former style is typified by the CSP and KPN formalisms, and is easily implemented in SystemC using FIFOs, semaphores, or other synchronization primitives*": we do not see the relation between CSP/KPN formalisms and the notion of synchronization points in the context of Transaction-Level models. CSP formalizes a notion of asynchronous processes synchronizing solely by rendez vous whereas KPN is about communication by unbounded fifos only. This has little to do with TLM, where the communication is done by shared-memory buses (in the current assumptions taken by the working group) and interrupts. These buses do not necessarily contain fifos, depending on the abstraction considered (for instance transactions can be solely *routed* by the bus model); as well, interrupts do not necessarily correspond to *rendez vous* like synchronizations.
  Regarding this particular section, these unrelated citations of formal works could be removed.
- Sequel of the text: see general remarks on "Coding style" below.

Coding styles:

The feedback from SPRINT already expresses the lack of balance between the various APIs proposed (blocking and non-blocking transport) and not well justified arguments for using one or another.

Synchronization:

This section of the standard also covers the problem of synchronization.
Paradoxically, it fails to guarantee good synchronization of components in an interoperable way while at the same time mixing this topic with the need to optimize simulations.

- Section 3.3.1: "*The application is expected to implement synchronization between processes by calling explicit synchronization statements. For example, a blocking transport call could […] return a flag to instruct the caller to wait, giving other processes a chance to execute*". This is the most detailed explanation of how components using the blocking transport interface should synchronize. A flag is evoked, that would materialize a way for a component to inform others that a synchronization point is reached, but none such flags is actually part of the standard.

- Section 3.3.3/3.3.5: "*an untimed model does not rely on the notion of simulation time for process synchronization and scheduling*". This is a quite naive view of untimed modelling. While models at the "untimed" level of details do not include actually timings (hence their name), the concrete simulation still needs some notion of simulation time. Many situations need the simulator time to advance: polling, intrinsically timed components such as timers, UART, I/O, etc. These situations correspond to places where the SystemC model must yield back control to the scheduler, but cannot do so by waiting for a specific event. It is then forced to wait for some time. Waiting for zero time (a delta-cycle) is notoriously subject to scheduling dependencies and does not compose well: a model using such statements may work well as is, but can then fail when adding an extra component because of spurious effects on the scheduling. The only way to solve this problem is to wait for some time strictly greater than zero. In order to still represent an untimed model and not a timed one, the delay waited for should be allowed to vary. This can be done by specifying a timing interval, and randomizing the values within this interval at simulation time. The standard is even self-contradictory here, because it recognizes that "*[the blocking transport interface] could in principle call wait to model a latency or to introduce a random delay in the process scheduling*" whereas the interface is explicitly linked to untimed modeling (and untimed modeling only) in the rest of the document.
  Actually, a clear separation between "untimed" and "loosely time" is irrelevant, because modeling with a total lack of timings information is difficult for many platforms, notably the ones that contain intrinsically timed components (timers, etc.). Models may contain more or less precise timing information: "untimed" like models will include very large timings intervals whereas "loosely-timed like" will have more precise intervals. We also note that there is no standard way in TLM2 to transport imprecise timings (timings intervals) with the transactions. This feature is needed here, because the interface function call can cross multiple imprecise timings annotations before reaching a randomized wait, and thus the sum of these annotations needs to be transported.

- Section 3.3.8: "*The blocking transport interface is particularly appropriate for models that include explicit, strong synchronization between processes such that the sequence of synchronization points is deterministic and independent of implementation details. […] The non-blocking transport interface is particularly appropriate […] where the exact sequence of synchronization points between concurrent processes is indeterminate*". We do not see in which way these assertions can be true. In an untimed/loosely timed world, the lack of precise timings involves representing multiple possible situations of the hardware in the models. For instance, if two IPs raise each an interrupt, either one IP will raise the interrupt first or the other; this is because the exact

date of occurrence for the interrupts is unknown. One cannot say in this case that the sequence of synchronizations is determinate. Contrarily, adding more precise timings in the models actually determines the order of the various events; it restricts the set of possible different sequences represented in the model. This can hardly be characterized as "indeterminate". And this has nothing to do with a choice of interface.

- For models that use the non-blocking transport interface, the standard proposes a way to optimize simulation speed by using a very specific technique called "temporal decoupling". This technique consists in yielding to the scheduler only when the sum of the timings annotation transported reaches a given threshold, called the "time quantum". There is a risk that this technique is understood by the user as a general synchronization scheme for the platform whereas it is not. As stated in the standard itself (section 3.3.2 and 3.3.3), this technique breaks the functionality of the platform if wrong values are chosen for the time quantum ("*Too large a quantum introduces large timing inaccuracies, possibly to the point where important events are missed and the model ceases to function*"). Therefore, the standard should insist on the fact that Time Quantum is a particular technique to speed up simulations, not a synchronization principle. Also, the notion of explicit synchronization point should be defined and explained, and rendered mandatory independently from the interface chosen, except of course when Time Quantum is in use. Only this can guarantee proper interoperability between models from the synchronization viewpoint.

Minor points:

Additionally, several minor points need to be worked out in the section:

- Section 3.3.7 "Cycle-accurate modeling": it seems confusing to talk about "cycle-accurate" in the context of Transaction-Level Modeling because for most people this term points to an abstraction level that is *not* TLM, and where components synchronizes via one or multiple clock signals. We do not see the need of a specific section, when approximate modeling is can already reach "cycle-count" accuracy if the user chooses to do so.
- Section 3.3.8: "*The blocking transport interface […] is also appropriate where an untimed transaction-level model is to be transformed directly to an RTL implementation without any intermediate timed SystemC model*". It is illusory to think and let think that transaction-level models are the new entry point to be "refined" into models with more details and finally reach RTL. There is still a huge gap between even a very precise timed transaction level model and RTL. Experience with high-level synthesis tools shows that the code that is favourable to these tools is not the same that is relevant to TLM

and simulation speed. This is also without talking of some models of I/O components, such as LCD controllers, whose TLM representation opens a graphical window when their RTL counterpart generates driving signals: refinement from TLM to RTL is hardly possible here. Globally, the scheme that is proven currently is to start to develop RTL and TLM at the same time, with TLM model creation finishing before RTL; this allows some activities such embedded software development to be started early. Again, the assertion made does not seem to be relevant for an interoperability standard.

3 - Core TLM2 interfaces

3.1 Blocking transport/ Non blocking transport interfaces

This topic is already fully covered in the SPRINT feedback.

3.2 Direct memory interface

3.2.1 General feedback

This interface does not fully address the problem of Direct Memory Accesses because some information are missing, and some assumptions are incorrect:

- The DMI pointer transported in tlm_dmi is of type `unsigned char *`. This assumes that the component accessed will represent the memory using this type, which is not always true.
- Because of this assumption, the standard does not make any provision to give information about the actual type used for the memory data storage. To be efficient and also for endianness reasons (see later), a memory can choose to represent its internal data storage using arrays of words. In this case, getting a pointer on a memory region without knowing the type of data in the array is useless.
- The standard gives the possibility to transport latency timing information. This is a useful possibility, but it should be noted that even when respecting these latencies, the resulting timings will not be very accurate, because they ignore any contention that would occur on the interconnect path to the target.
- While the standard allows to get information about latencies, there are no ways for an initiator accessing a target through DMI-only to know if it accesses addresses that are explicit synchronization points and whether a wait instruction should be inserted.
- The “endianness” attribute should not contain the simulated endianness of the component, but the actual endianness of the data: an IP may choose to store data in one endianness or another, depending on its needs. This is independent of the actual data returned when accessing these data through a conventional transaction.

Additionally, this interface in itself addresses only a minor part of the needs to get information about components. There is no standard way to get/set (when applicable) the following information from a component:

- List of addresses that are explicit synchronization points. As an example, some memory synchronization addresses are often set during the simulation in function of the embedded software executed,
- Simulated endianness,
- Vendor and information about the IP model actually simulated,
- Etc.

Indeed, a more general interface allowing to ***get and set*** any kind of information on a component (IP model or bus model) is required. A subset of mandatory information should be identified by the standard, with some vendor dependent information still possible. These vendor-specific additions could evolve with time and may be registered as "mandatory" or standardized in subsequent versions of the standard if the community agrees to do so.

3.3 Debug transaction interface

3.3.1 General feedback

This interface is aimed at only one problem: accessing internal data of a component from a debugger without issuing transactions. There are many other uses where indeed issuing a "fake" transaction is useful: filling a memory from another standard component without interfering with transaction synchronizations and other traffic, debugging some particular accesses with the rest of the platform running normally, etc. To do so, one really needs of the same transactions than in the normal case, except than these transactions needs to be tagged "debug". Instead, the TLM standard reinvents a particular payload. This payload has no accessor methods, no consistency with the generic payload for the naming and representation of data fields and furthermore does only include a limited subset of the data fields available in the generic payload. Therefore, it is impossible for instance to debug streaming accesses or transactions with byte-enable. In addition, the standard reinvents here a transport interface (specific for the debug payload): it is thus also impossible to generate debug traffic *a la* non-blocking transport (debug requests and responses). All this means that a user implementing an IP component will have to write specific code for the debug interface in addition to the handling of standard transactions.
To us, this choice of separating both the payload and the interface for debug purposes as none advantage and many drawbacks. It would be far simpler and would cover many more applications to add a debug mode to transactions. The only disadvantage of this latter solution is the constraint to use a two-path access with a non-blocking transport model to access a register's value from a debugger. We believe such accesses are sufficiently rare so that the associated performance penalty is negligible.

4 - Sockets and combined interfaces

4.1 Problem statement

- Why is the BUSWIDTH parameter expressed in bits? This choice imposes to do rounding everywhere, because the natural use for this parameter does seem to get its value in *bytes*. Even the standard needs the rounding expression for some of its assertions (BUSWIDTH/8, (BUSWIDTH+7)/8). We believe this parameter should be expressed in bytes instead.

5 - Generic payload

5.1 Generic payload attributes

5.1.1 Problem statement

- The standard does not make any provision for transporting a "socket id" with the transaction.

  Such an attribute is necessary when an IP model has got several sockets with the same protocol (for instance, here several target sockets receiving transactions): it then needs to known, in the target's implementation of the interface function call (inside `b_transport()` or `nb_transport()`), from which socket the transaction is coming. Because of the way SystemC interface function calls work (using C++ `operator->`), this information needs to be injected in the transported data *before* the function call is done. The examples provided with the standard (see in particular the `simple_initiator_socket` class in directory "`common`") shows what happens when trying to escape this constraint. A class `simple_socket_utils` is used, that implements the equivalent of a global variable containing the socket id of the "current" transaction being received. This variable is supposed to be assigned with the socket id value of the "current" transaction being received; it can thereafter be read by the target. In order to allow this assignment, the user must "break" the way SystemC interface function calls work: the interface function call does not end directly in the implementation of the function inside the target, but is received instead by the user-implemented socket (which then assigns the socket id value). But then, the user-implemented socket has to call the target's implementation of the interface: it needs a pointer to the component, and a pointer to the function. This leads to a complex registration process that has nothing to do with the standard SystemC binding of port/export.
  Additionally, the global variable solution has a big disadvantage in the long term: it forbids the use of any SystemC simulator that would support parallel architecture. Indeed, it would not be possible to simulate in real parallelism two independent part of a system, because all transactions would use this same global variable to set their socket id.
  To us, it would be better to include a standardized socket id in the generic payload. This solution has one drawback: it mandates the initiator to assign the socket id of the target when initiating the transaction and it requires to get this

information using the platform's bindings. However, without this, model implementers will resort to custom solutions that will possibly be equivalent to the global variable scheme demonstrated in the standard's examples, with the problems aforementioned.

- The generic payload includes an accessor called "`get_response_string()`", that allows to obtain a human-readable information on the reason for a response error. However, this information is just a standard translation of the `tlm_response_status` enumerated type already returned in the payload. It would be more informative to allow the target IP to return a customized plain-text error message in addition to the response status field.
- There are several instances of statements about error reporting that mandates the use of the SystemC error reporting mechanism (notably in section 9.18.1). We do no think it is the role of the standard to impose a particular error reporting mechanism, users may as well use their own in-house mechanism (with possibly more advanced features such as error logging to file, more specialized error classification, etc.). In addition, we do not understand why the possibility b) and c) in section 9.18.1 are strictly exclusive: a component should be permitted to both display an error message (if the user chooses to do so) and return an error status to the initiator.
- The generic payload provides a mechanism to duplicate a transaction and all its related data, known as "deep copy" mechanism. This mechanism violates a well-know and accepted software engineering rule about memory management: "an object allocating memory is at the same time responsible for releasing it". Here, a call to the function `deep_copy()`results in a memory allocation for the data and byte-enable arrays, but there is no code inside the generic payload to release anything. Actually, the owner of the data and byte-enable arrays is not the payload but the initiator model itself. We think that providing this mechanism "as is" will result in code where people forgets to release the memory allocated for these arrays, producing numerous memory leaks. It should be the component/software part requesting the transaction's copy that should bear the responsibility to allocate the necessary memory before calling the copy mechanism. This has one additional advantage: it lets the software part requesting for the copy optimize memory allocation for this use.

5.2 Endianness

5.2.1 Problem statement

The issues of endianness, data pointer, unaligned addresses, byte enable and unaligned size are all linked together. The philosophy of the TLM Standard is to consider data as an array of char (`unsigned char *`) while the data actually

represented are words of size BUSWIDTH bits. However, given the type of the data pointer used, it is explicitely possible to access data byte per byte. Accessing a byte inside a word (or a subword inside a word) is notoriously dependent on the endianness of the host machine. To solve this endianness problem, the standard mandates the representation of data in the generic payload to be host endian (Section 9.15.c). Separately, the standard permits transactions to addresses that are not aligned with the BUSWIDTH, as well as transaction sizes that are not a multiple of the BUSWIDTH.

When dealing with endianness issues, the goal to achieve performance is to minimize the number of endianness conversions required, whatever the simulated endianness and the host endianness are. The solution proposed in the standard is efficient only in the case where the simulated endianness is the same than the host endianness. As an example, simulating a big-endian initiator sending data to a big endian target on a little-endian host will incurs two endianness conversions: one for putting data in the payload, the other for extracting them. These conversions are unnecessary and unacceptable in this case from a performance viewpoint.

Because of the way data is represented in the generic payload and the fact that it is host endian, it becomes possible to handle accesses to unaligned addresses or whose size is not a multiple of the word size. These transactions are hardly necessary. While they could be handled without complexity in memories (by using array of chars as internal representation), they require specific management in all other IPs, which generally do not support unaligned accesses and have to systematically return an error. Also, for an IP that works at word-level, it is impossible to cast the data pointer of an unaligned transaction to a word pointer; an expensive data copy is required instead.

To cope with these problems, we propose the following solution:

- All transactions must be address-aligned and size-aligned with the word size given by BUSWIDTH.
- The type of the data pointer should be `void *`, to clearly means that accesses to individual bytes are not systematically allowed.
- Transactions without byte-enable (byte enable pointer set to 0) are considered as word-level accesses. In this case no endianness conversion is needed if both IPs do have the same endianness, independently from the host endianness. Casting of the data pointer to a word pointer which matches the BUSWIDTH is allowed.
- Subword or unaligned accesses are performed using byte-enable (with re-aligned addresses). In this case, the endianness of the IP needs to be taken into account, and a conversion function used. Casting of the data pointer to a subword/byte pointer is allowed provided that the target ensures it will only access subwords/bytes that are enabled in the byte enable array.

This solution minimizes the cost of endianness simulation:

- Endianness conversion is required for sub-word accesses, which are assumed to be in far less number than word accesses (if the majority of accesses are bytes, then the BUSWIDTH should surely be changed to 8

bits). The endianness conversion is still free if the host and IP endiannesses match, otherwise data swapping is required.

- Aligned transactions means less error management for the implementation of IP registers and allows the data pointer to be casted to a word pointer in situations that are clearly identified, enabling efficient data interchange with the IP internals.

5.2.2 Documentation feedback

This section of the standard seems to be a placeholder for explanations belonging to other sections. While the examples given are clearly welcomed, this section should not be the place for explaining byte-enable, which should have been discussed extensively before.

5.3 Extension mechanism

5.3.1 Problem statement

- We do not understand why the actual implementation of the extension mechanism as currently proposed by the standard is so complex. In particular, the `tlm_array` class seems to reinvent the wheel and would better be replaced by a standard STL associative structure such as `std::map`. We also do not understand the purpose of discussions in points 9.19.4.v and 9.19.4.w about the internal size of the extension array: this should be left to STL structures.