Analysis of TLM-2.0 and it's Applicability to Non Memory Mapped Interfaces

Guillaume Delbergue
GreenSocs -
Bordeaux INP, CNRS IMS, UMR 5218
guillaume.delbergue@greensocs.com

Mark Burton
GreenSocs
mark.burton@greensocs.com

Bertrand Le Gal and Christophe Jego
Bordeaux INP, CNRS IMS, UMR 5218
bertrand.legal@ims-bordeaux.fr
christophe.jego@ims-bordeaux.fr

***Abstract*—Virtual prototypes have become a critical tool for software development, architecture design, and importantly to bring software and hardware developments together. The introduction of the SystemC and the TLM-2.0 standard [1] has enabled a degree of interoperability and model re-use. However while the ambition for TLM-2.0 was wider, in reality it has only enabled interoperability for memory mapped bus based protocols. Since a typical virtual platform contains a wide variety of interconnect, individual designers have to define their own, often non-interoperable solutions. Our aim in this paper is to propose improvements to the existing TLM-2.0 standard, and a methodology around the TLM standard with the objective to facilitate the definition of a wide range of interfaces (not just memory mapped buses) in a consistent manner. Therefore in this paper, the existing Accellera TLM-2.0 standard is analyzed, together with existing interface 'kits' (e.g. Accellera's OCP-IP TLM kit). Then, we examine a selection of serial protocols (SPI, I2C, CAN, UART, Interrupt) to ascertain what is required outside of memory mapped buses in terms of a SystemC interface, and what is lacking in TLM-2.0 which prevents its adoption for non memory mapped bus interfaces. Concrete proposals are made to improve the TLM standard for interoperability. A methodology is also suggested for building a new TLM interface standard kit. One important criteria is that there should be a singularity of outcome, a clear methodology that, given an interface protocol, yields a consistent modelling solution, such that the choice of one designer is likely to match that of another. Thus we hope to reducing the difficulty of providing standards for all interfaces.**

I. Introduction

With the increase of the complexity of programmable devices and systems of programmable devices, virtual prototypes are an essential tool used both during the conception of systems (both on chip (SoC) and more distributed systems). Virtual prototypes bridge software and hardware teams in the development and verification flows. They enable specifications and constraints to be checked during the design exploration.

Transaction Level Modelling (or TLM) is the term given to the levels of abstraction at which virtual platforms can be modelled. It's based on a technology, namely the use of "Transactions" for simulations. The TLM abstractions are designed to facilitate model creation, to provide early virtual platforms addressing software development, hardware and software integration, performance analysis and architectural constraint investigation. The models are also used within hardware, software and full system verifications to drive high level synthesis. The TLM abstraction levels are therefore relatively broad, and cover a wide range of use cases. This is potentially one of the problems with the TLM-2.0 standard as it tries to satisfy a wide range of requirements.

The Accellera TLM-2.0 standard is a part of IEEE 1666 and is built on top of SystemC. It standardizes the TLM abstraction levels from the perspective of model interfaces. It provides an interoperability layer and some utilities. TLM-2.0 is a layered standard built on interfaces between three kinds of elements : initiator, target and interconnect. While a single interface remains statically assigned to a specific role, the standard allows the three types of models to possess multiple interfaces and dynamically change their role between those interfaces. TLM-2.0 introduced the notion of time decoupling and "quantums". We will return to the question of the use of quantum's below in Section IV.

TLM-2.0 enables models to be built and operate at different levels of abstraction. Two principle levels are defined, so called "Loosely Timed" (LT) and "Approximately Timed" (AT). These names do not help to identify the abstraction level in terms of the use cases they cover, nor the technology that is used to implement them. Their names are confusing and unhelpful.

In reality, LT is the abstraction level aimed at supporting virtual platforms for software development, with only as much timing information as the programmer needs. It is implemented using a blocking function call across the interface which is expected to 'complete' a transaction in a single call. Conversely, AT is aimed at providing more accurate timing, that can be used driving architectural exportation, synthesis, etc. It's based on a non-blocking call mechanism and a sequence of phases which are protocol specific. It also defines the timing points of the protocol.

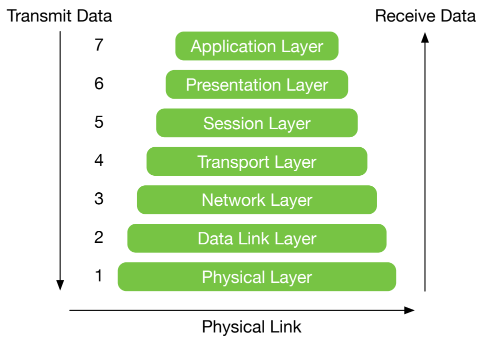


Fig. 1: OSI model layers

The additional modelling of time has a cost in terms of simulation performance. The transaction remains common between the two types of interface.

The rest of the paper is organized as follows. In order to guide the choice of how specific interfaces should be modeled within TLM, Section II proposes a more formal definition of the transaction. Some common non memory mapped protocols are examined and proposed TLM implementations suggested in Section III. Then, different approaches and issues to time and synchronization are discussed in Section IV and a solution is proposed in Section V. In section VI, improvements to the TLM-2.0 standard are discussed. Finally, Section VII proposes what should make up a ”first class” standard interface.

II. Towards a definition of a transaction

Care was taken in the original TLM-2.0 standard to try and ensure that models written at one abstraction level could be re-used at another one. It requires a transactor (hidden in the socket) to bridge between abstraction levels. The commonality between all abstraction levels is expected to be the transaction itself. It is therefore critical to understand what exactly is meant by a “transaction”, what should be held in a transaction, and what should be left out.

Within the context of a wider scope of interfaces, there is no clear agreement of what constitutes a transaction, and little literature on the subject. This leads designers to make radically different decisions about how to model interfaces (for instance see [2] and [3]). Our principle motivation is to try and ensure a common approach. To this end, we propose a more formal definition of what should constitute a “transaction”. But, we cannot loose sight of the fact that the TLM standard is broad in it’s applicability. Our aim is to guide the choice of “transaction” to support the wide range of use cases. Our proposal should re-confirm the choices taken for a memory mapped bus in the existing TLM-2.0 standard, while guiding decisions for non memory mapped bus interfaces.

A. OSI model transaction

In making a proposal for formalizing what constitutes a transaction for TLM, we draw on an existing standard, the OSI (Open Systems Interconnection) communication model [4]. It divides communication protocols into multiple layers of abstraction as described in Fig. 1. This model is extensively applied to describe Internet communications. But it is designed to have wider applicability. For instance, the CAN (Controller Area Network) bus standard refers to the OSI model to describe layers of the CAN protocol.

From bottom to top, the first OSI layer, named the physical layer, treats communication at the digital bit level as raw data on a communication channel. This layer has to define physical characteristics like voltage levels and timing.

The second OSI layer, the data link layer, acts as a binder. This layer describes the transformation of the physical atomic information (bits) into frames (sequences of bits). This layer has to detect frame borders and manage errors during transmission. This is the layer at which data congestion and buffering is handled. The unit of information of this OSI layer is the frame. However, this layer doesn’t take data routing feature into account when multiple devices are interconnected.

The third OSI layer is the network layer. This layer manages routing to support networks and sub networks. It also manages resource conflicts and interconnect between networks. The unit of the information to this OSI layer is the packet. This layer is used for “One to Many” or “Many to Many” communications.

Higher layers like the transport layer will define flow control of data and systems for higher level error checking and recovering. We propose that the second and third layer, the data link layer and the network layer, map directly onto the transaction level modelling abstractions for AT and LT. We consider that the forth layer is frequently already part of the software domain that we expect to be written as part of one of the main use-cases for TLM models (software development especially of device drivers).

As shown previously, the TLM-2.0 standard introduces two levels of abstraction. Our proposal is that the OSI layer two maps onto the lower level of abstraction (AT), dealing with frames and buffer congestion (covering the common use case of bus sizing and architectural exploration of bus fabrics). The third OSI layer maps onto the higher level of abstraction (LT) dealing with routing and packets. In other words, we expect a TLM to expose the details of the network layer, and to use something akin to the OSI 'packet' as the transaction.

This mapping is not totally without issue, the OSI network layer includes resource conflicts along with routing. This makes perfect sense, but in the case of a TLM model that is more abstract in its timing, time-based resource conflicts may be ignored or un-modeled at the LT layer. Hence, we consider resource conflicts of this sort to more naturally belong at the lower level of abstraction in terms of TLM. Nonetheless, we consider that the OSI model is a good basis to determine what constitutes a transaction, namely an OSI layer three packet. The OSI model defines a packet as a bank of data containing control information and the payload. The control part contains information to deliver the payload like the address (if applicable) to destination, error and sequences. The packet provides enough information to route the data between multiple nodes for “One to Many” or “Many to Many” communications.

TLM-2.0, as we will see in Section VI, has focused heavily on memory mapped buses. A transaction for a memory mapped bus is universally assumed to be a read, a write, or in some cases a read-modify-write operation performed on that bus. The transaction carries all the data that are necessary for complete operation. For buses that are capable of performing bursts, the data covers the entire burst ([5]). In other words, the notion of transaction is the full operation. This fits perfectly with the OSI layer three definition. The packet contains the address and the data, just as does a TLM generic protocol transaction, and is route-able. Furthermore, the 'frames' and timing points associated with frames are exactly akin to the burst packets. They are modelled at the AT level in a complete bus modelling kit, such as the OCP modelling kit referenced above. Hence for a memory mapped bus, we consider that the OSI model to fit well. We therefore believe this somewhat validates the use of the OSI model to guide the choice of transactions for other specific interface protocols.

In order to guide designers choice about how to write interface models, we believe that it is critically important to have agreement on what constitutes a transaction. We consider that the adoption of the OSI model is not only technically robust, but also comes with the benefit of an existing, well documented, and well understood standard.

In the next Section, the implications of choosing the OSI model is discussed. Some protocols are analyzed. We will see how the OSI model can guide the choice of a transaction, and hence the appropriate phases, and how that can then be represented in a SystemC TLM context.

III. Non Memory Mapped Interfaces

This section examines a non exhaustive list of interfaces commonly used in and between SoCs. First, the family of “One to One” serial protocols is examined including signal (GPIO, IRQ, etc.) and RS232 [6]. Then, “One to Many” protocols are analyzed with RS485[7] and SPI (Serial Peripheral Interface). Finally, I2C and CAN, “Many to Many” protocols are examined. The analyses shows how the OSI model can help establish the transaction, and associated phases. The intention is to exploit these protocols to examine different aspects of how interface kits should be constructed, and to validate the use of the OSI model. A complete analysis or implementation is not provided for each protocol.

A. “One to One” protocols

The most common and low level communication protocol is surely the signal. A signal, often a single wire, can be used for an interrupt, for GPIO, reset etc. Although in general, all of these examples are considered as “simple signals”, often that masks a level of detail. For instance an interrupt can have different levels, depending of the interrupt vector IP and their physical wiring. Neither it is always the case that because a protocol is implemented on a single wire, the protocol is in any way 'trivial'. In many cases, single wire protocols can be some of the most complex.

By considering only signals that can be active (asserted), or not active, the 'packet' is the assertion of the signal. This can be modeled with no payload whatsoever, simply the fact that the signal has been activated is enough information. Many if not all of these “simple” signals often carry extra information such as vector address.

In real hardware this may be encoded by their physical placement, their wiring or indeed by multiple wires. Hence we consider its good practice to include the TLM-2.0 extension mechanism in the payload.

Our first conclusion is that the TLM-2.0 extension mechanism should be present in all payloads (even, and possibly especially, trivial payloads).

In the case of simple signals, there is only one OSI layer two frame in the layer three packet. Hence we consider such protocols to have only two phases: the start and end of the packet, e.g. when the signal becomes active, and when it stops to be active. Note that for something such as an interrupt, this can mean that the transaction is 'live' for a considerable amount of time.

A suitable TLM-2.0 implementation of a simple signal can be found in GreenSignalSocket, a part of GreenLib [8], a library based on TLM-2.0. In this case the payload only has the extension mechanism inside it.

UART (Universal Asynchronous Receiver Transmitter) is based on transmission (TX) and reception (RX). The communication between two UARTs is bidirectional. The protocol can be simplex, half or full duplex. The useful data length is typically between 5 and 9 bits. UARTs can be seen as a family of multiple serial protocols like RS232 or RS485. The RS-232 standard is a simple point-to-point serial communication protocol.

A start bit and one or multiple stop bits delimit a typical UART frame. These are OSI layer two features, and should be modelled as timing events (phases) by marking the start and end of a frame. It can also be a parity bit to check the received data. Parity is a layer two feature, and should be modelled as content in the transaction. Parity errors can be generated by transmission collision on certain UART protocols. These would be handled by requesting re-transmission on a parity error. This may be handled in software or hardware depending on the UART protocol and/or system configuration.

A suitable TLM-2.0 implementation of an UART can be found in GreenSerialSocket, again a part of GreenLib [8]. GreenSerialSocket provides an initiator and a target bidirectional serial socket and also a payload for UART communication including RS232 protocol. It simplifies interoperability of nodes exporting a UART interface. As the UART protocol is not memory mapped, the payload does not follow the Generic Payload defined in TLM-2.0, but is designed specifically for the UART case. Indeed, fields like `address`, `byte_enable` or `dmi` don't make sense for a UART. The parity bit has been left in the payload. Though the parity bit is correctly part of the payload, the TLM-2.0 implementation should be revisited in future work as it may not be ideal.

B. "One to Many" protocols

The RS485 standard is similar to RS-232 except it uses a different physical layer, and contains a master/slave relationship adding routing. Hence, some aspects of RS485 are in layer three as well as OSI model layer 2, and a kit supporting RS485 would have to consider 'routing' between multiple masters/slaves on the same 'wire'. RS485 transactions will contain both address and data information, while an RS-232 implementation would only need to contain data. In this respect an RS485 implementation is relatively trivial to conceive, though it ideally requires a "broadcast" socket. A socket can be trivially constructed by using a `multi_passthrough_initiator_socket`. A router with single socket can also be used.

The SPI protocol, a "One to Many" protocol, is also used for communication between multiple nodes. A SPI node is either master or slave, ie an initiator or a target in terms of TLM. The communication is full duplex. There is no defined address to select the receiver node. Moreover, the master must enable a chip select on a slave to communicate. In this way, once set up, SPI is fundamentally a point-to-point protocol. The issue is the setup. Typically, a SPI controller or a GPIO interface can be used (which is like a DIY SPI controller), and all of this can be combined with SPI models being daisy-chained. In that case, communication for one slave node is passed onto another. However, the make up of the data packet itself (including any information used for daisy chain addressing) is not standardized. This issue has been pointed out by [9].

Viewing SPI through the prism of the OSI layer model, routing information should be handled in layer three. However, it is by no means clear that the routing mechanism actually forms part of the protocol. If we consider a simple SPI connection in which a master enables a single slave, then within a TLM context this can be adequately modelled as a point-to-point connection. The socket binding enforces the topology of the system. There is no "routing" as such. More complex systems that make use of SPI controllers, and/or chaining effectively layer routing on top of this. In the case of a controller, the controller itself has a bus interface, and "routing" happens within the controller. The controller itself sets up multiple SPI interfaces to the various slave nodes. In this case, the controller is acting as a sort of router, and SPI bridge. Finally, in the case of a chained SPI, there is a notion of "address" embedded within the data transmitted through the SPI interface. However, this is non-standard, and typically has

TABLE I: Serial protocols in regard of TLM

TLM / Protocol	Signal	RS232	RS485	SPI	I2C	CAN
Family	One to One		One to Many		Many to Many	
Payload	Empty	data_ptr, data_length	data_ptr, data_length, address	data_ptr, data_length, address	cmd, data_ptr, data_length, address	frame_type, identifier, data_ptr, data_length, address
TLM Phases	start, stop	start, stop	start, stop	start, stop	start, stop, ack, pause, restart	start, stop

to be set up under software control. Therefore, it seems appropriate that the SPI interface should transmit the data just as SPI does, and should leave the devices to interpret the data as best they see fit.

Another common implementation of the SPI interface is based on a GPIO controller as a sort of 'DIY' (Do It Yourself) SPI controller. In this case, the GPIO interface is acting as OSI layer 1. Typically, GPIO pins are used to drive chip select, and potentially the data and clock signals. In order to model the SPI device at OSI layer 3, we have to 'abstract' the layer 1 interface. To abstract the low level pin interface to the TLM interfaces requires a TLM transactor. Transactors are used for communication between models (including at the RTL level) or real implementation . We will examine later how all 'good' interface kits should include transactors down to OSI layer 1, the physical layer. This will be a key recommendation.

SPI is often used to communicate with memories. Some embedded devices use SPI memory to boot the software stack (often using Quad SPI). In that case, it may be helpful to abstract from the SPI protocol to the memory interface and support a DMI interface. Our expectation is that this should happen through a SPI controller, which has a memory mapped bus port, capable of supporting DMI.

C. "Many to Many" protocols

The I2C protocol is a "Many to Many" protocol. Each I2C node can be a master or a slave or a multi-master. A multiple master can talk at the same time on the bus to a number of slaves. I2C protocol is half duplex. Each node on the bus has an address or several addresses. The communication consists of multiple frames (OSI layer two) that contains address, command and data information in packets (OSI layer three). Hence at OSI layer 3, these three elements must be present in the transaction content. At OSI layer 2, there are frames for address, control and data, with start and stop phases. Collisions cause the completion times of some frames to be prolonged. It is important to model this at an "AT" level of abstraction in order to model potential bus throughput. To detect collisions, an I2C implementation has to find overlapping frames by using their start times and to check that their data isn't identical. According to the I2C standard, the first master that attempts to write a 0 while another master writes a 1 looses the bus arbitration. This can then be used to delay some master frames. I2C also has the concept of a frame acknowledge, pause and restart. These are all OSI layer 2 features, and should be modelled using phases in TLM. These phases can occur during the frame transmission, and may abort the frame or the packet.

Finally, the CAN bus is a broadcast protocol just like a standard memory mapped bus. Each node on the bus can be a master or a slave. There can be multiple masters. Messages are received by all nodes. Nodes use the message identifier to differentiate potential recipient(s). The message identifier is also used as a priority field. As collisions can happen, the standard guarantees that the highest message identifier will be distributed first. Each CAN frame contains a CRC (Cyclic Redundancy Check) to detect error during transmission. Again, the identifier is used by a routing mechanism which sits as OSI layer three. In terms of modelling, routing can of course happen as it does in hardware via a broadcast. However, it is computationally much more efficient to use a "router" to model the bus medium itself, and only pass transactions to the correct node. The OCP (Open Core Protocol) SLD Kit [5] shows how this can be effectively achieved using standard parameters (for instance those that will be standardized by the Accellera CCI (Configuration, Control & Inspection) WG) within the target sockets to hold the addressing information.

Data and error information (as with UARTs) similarly sit in the transaction contents as they are an OSI layer three feature. The CAN protocol defines four kind of frames: data frame, remote frame, error frame and overload frame. The data frame is the general frame used to send data to another node. It seems perfectly reasonable to have a frame identifier in the transaction and a union of the components that each frame carries. As the size of a CAN frame is not fixed, the data can be from 0 to 8 bytes, the TLM-2.0 data mechanism can be used to handle the data itself. Only one frame is considered per transaction, so at OSI level 2 there is nothing more than the single frame whcih can be modelled at AT using two phases.

The Table I sums up several protocols. In this section, a number of protocols are examined with reference to OSI layer model, the transaction and associated phases have been identified. As the OSI model is an international standard applicable to communication protocols, it decreases ambiguity between developers about how to define the transaction. The OSI model has been established for many years and the content of each layer is clearly defined. It fits with the existing Accellera TLM-2.0 generic protocol designed for memory mapped buses, and it seems to work for a wide cross section of other interfaces. Therefore, it seems a good approach for determining what should be identified as a transaction.

However important a transaction is, there are unfortunately other issues with the TLM-2.0 standard which also allows for ambiguity when building a new interface. The next section examines synchronization issues between models, which could be nodes of a serial protocol.

IV. Ordering and timing of the simulation

One of the most important features of the SystemC library is way time is modelled. SystemC defines it's own notion of time that is normally called the simulation time. The SystemC kernel manages this time and provides a global time stamp. The SystemC kernel advances time to the next outstanding SystemC event. Hence SystemC time is non-linear and bares no relationship to real time. Furthermore, activity can occur in zero simulation time.

TLM 2.0 introduced temporal decoupling and the notion of a quantum. Time decoupling is the term given to models which run within their own time reference "decoupled" from the SystemC kernel simulation time. Typically they run ahead of the kernel simulation time. The quantum is a measure of the amount of time that a model is permitted to be decoupled. SystemC is fundamentally constructed to provide a "co-operative serialization" of parallel threads. Each (parallel) thread in SystemC (SystemC threads) is expected to periodically yield to other threads in the system, allowing all threads to advance. In terms of TLM-2.0, it is expected that initiators are SystemC threads. A quantum simply indicates how often they should guarantee to relinquish control to other threads.

As initiators in a TLM context are typically complex IP blocks, it is very often the case that to simulate them efficiently requires considering a relatively large block of functionality. In reality, it may take considerable time to execute, but as an entire block can be more easily modelled. Therefore, it is advantageous from a simulation point of view to execute these blocks atomically, effectively decoupled from time. The Quantum limits the size of these blocks. Hence in general, the larger the quantum is the larger the block of functionality that can be considered, and the more optimal the simulation performance may be. Clearly, this is very dependent upon the technology used for the model concerned, and there will be a limit to the size of block that is optimal, an examination of this effect can be found in [10].

TLM doesn't enable a hierarchy of quantums, with local quantums and a global quantum. At the end of a quantum a model is expected to be re-synchronized with the simulation time. For instance SystemC's wait function can be called so other events can occur. A convenience function is provided in the TLM Quantum Keeper. The TLM Quantum Keeper is present in the TLM-2.0 kit to help manage local quantum time. TLM-2.0 recommends the usage of the TLM global quantum, but it is possible to use different local quantums in different section of systems.

Additionally, TLM-2.0 introduced the notion of annotated time on transactions. This is a means of sharing the current temporal decoupling time from initiators to targets. Temporal decoupling and the use of a quantum is only supported at the higher (LT) level of abstraction defined by TLM-2.0. Overall, different notions of time exists: wall clock time experienced by the person running the simulation, the SystemC simulation time, the quantum time, and the annotated time.

Generally, local quantum time of initiators increases through local compute operations or through TLM transactions by using annotated time. Typically, an initiator can send a transaction with a delay α. The target reads this delay, prepares an answer, possibly increases the delay by including the time it takes to compute the answer and then sends it back to the initiator. The initiator sets local quantum time by using the value received from the target. Hence, the expectation in the TLM-2.0 library for typical target models is that they may not need to interact directly with a quantum keeper (and the initiator local / quantum time), rather they simply use annotated time.

In previous works [10], we have reported on the efficacy of using quantums, and how much of a speed increase we are able to get by correctly adjusting the quantum length. However it is apparent that there is no simple formula to calculate the quantum length. This is a significant disadvantage of this approach. Indeed when the effect on simulation speed against quantum length is plotted, it shows that there can be several optimal values. Meanwhile the commonly accepted practice within industry seems to be to set the quantum to the minimum interrupt period expected in the system. Indeed our earlier work shows a minimum simulation time at exactly the point where the quantum is set to the period of the clock used to drive the Linux kernel. In our case, 10ms as the kernel clock is set to 100Hz. This is a reasonable approach and to be expected, as what ever is driving those interrupts will need to be triggered from the SystemC kernel. However this approach has it's own limitations. System clocks are typically responsible of regular and frequent interrupts. They are not only programmable but different operating systems have more or less tolerance of different clock 'skew' and require different frequencies. Hence, the quantum need adjusting by the very end user. As systems become more complex, of course finding a common denominator for all the clocks in a system becomes more complex, especially if some of those clocks are not known, and the result could be a very short quantum.

A different approach has been proposed in which ordering, rather than time is considered. In this approach, rather than periodically yielding each quantum, specific points are found in the design where yielding is imperative. These are typically where information is passed between initiators. In this case, models are "synchronized" with each other as often as required. It's not necessarily with the SystemC simulation time. Effectively, an optimal serialization of the different simulation threads is achieved [11]. This has a number of advantages: it assists by finding problems in a design architecture caused by the expectation of synchronicity that is not enforced in the hardware architecture. It forces designers to consider how synchronization will happen in their architecture. It does not require any adjustment of the quantum by the end user. However, it has a number of disadvantages, not least, as noted above, models of this sort may not work with sub-systems that expect quantums to be used.

In an attempt to mitigate these issues, we propose some changes to the way in which the quantum keeper is implemented and used. Models that are capable of generating interrupts from external sources (e.g. clocks) are a critical driver for quantum lengths. We have focused on them, though our solution should work for a much wider range of events in the system. It is detailed in the section V.

V. Improved TLM Quantum Keeper

We have seen how the critical driver for quantum lengths seems to be timer interrupt generators and other sources of interrupts. These are typically driven from events based on the SystemC simulation time. However, really their time relates to the local quantum time of the device that they will be interrupting. It would make more sense if they were based, not on the simulation time, but on the quantum time.

Doing so would also remove the need to base quantums on events that were essentially local to the local quantum domain (typically a SystemC thread). This would allow system designers to concentrate on the inter-thread issues, either adopting an ordering, or a quantum based approach. Enforcing an ordering approach across these interfaces does not preclude the use of quantums in addition, this may be a way forward in terms of standardization. In other words, one possibility is that the standard stipulates that any communication between a quantum domain is accompanied by a synchronization between the domain and the SystemC kernel. There is currently no technical means of ensuring this which does not also effect internal-quantum domain communication.

A. *Notification system*

Currently, a timer can use SystemC events (`sc_event`) to get a periodic callback based on simulation time. If the period is smaller than a quantum, the timer will be called less frequently than it aught to (with respect to the initiator that it will interrupt). Conversely, if the quantum is too short, there will be a detrimental effect on simulation performance. Our proposal is to modify the TLM Quantum Keeper class, and provide an event queue based on the local quantum time. Using the same mechanism for interface registration as the transport interfaces use, the improved quantum keeper class adds methods to register a callback at a certain time. As quantum time progresses under the control of the initiator for instance the improved util class checks and executes any pending callbacks. Typically, a timer can use this notification system to run a clock decoupled from SystemC time, without synchronization between the local quantum time and the simulation time. As the timer is now based on the initiators local quantum time, this solution also improves the timing accuracy of the model. Note that, while we provide an example Quantum Keeper utility class, the key is the API.

B. *Experimental results*

Two major changes are required to implement a notification mechanism based on the local quantum time. First the quantum keeper itself needs modification - the API to the quantum keeper needs extending. Second the quantum keeper must be 'findable' by all models within the quantum domain (which is likely to be the SystemC thread, and is likely encapsulated by an element of the SystemC model hierarchy). We propose that the quantum keeper should

```
template<typename MODULE> class QuantumKeeperPlus : public tlm_quantumkeeper {
public:
    typedef void (MODULE::*NotifyPtr)(sc_core::sc_time&);
    QuantumKeeperPlus(): tlm_quantumkeeper(), m_notify_ptr(0), m_mod(0), m_notify_time(0, sc_core::SC_NS) { }
    virtual void sync() {
        if(m_notify_time >= get_global_quantum())
            m_notify_time -= m_local_time;
        tlm_quantumkeeper::sync();
        if(m_local_time >= m_notify_time)
            notify();
    }
    virtual void inc(const sc_core::sc_time& t) {
        tlm_quantumkeeper::inc(t);
        if (m_notify_ptr)
            if(m_local_time >= m_notify_time)
                notify();
    }
    virtual void set(const sc_core::sc_time& t) {
        tlm_quantumkeeper::set(t);
        if (m_notify_ptr)
            if(m_local_time >= m_notify_time)
                notify();
    }
    void notify() {
        assert(m_mod);
        NotifyPtr notify_ptr_copy = m_notify_ptr;
        m_notify_ptr = 0;
        (m_mod->*notify_ptr_copy)(m_local_time);
    }
    void register_notify(MODULE* mod, void (MODULE::*cb)(sc_core::sc_time&), const sc_core::sc_time& time) {
        m_notify_time = time;
        assert(!m_mod || m_mod == mod);
        m_mod = mod;
        m_notify_ptr = cb;
    }
    MODULE* m_mod;
    NotifyPtr m_notify_ptr;
    sc_core::sc_time m_notify_time;
};
```

become a first class object (`sc_object`) supportable through the upcoming Accellera CCI standard, such that it can be found through the CCI parameter searching system. Below is our proposed Quantum Keeper itself.

The improved version of the TLM Quantum Keeper has been tested with a timer. In our implementation, the target accesses the TLM Quantum Keeper as a constructor parameter on instantiation, in the future we would rather see this use the CCI mechanism. Before the end of the elaboration, the timer registers a callback using the notification system for a time equal to it's period. Then, as quantum time moves in the initiator (e.g. as a result of annotated time from transactions) the timer callback is triggered. The timer will generates its IRQ and re-register the callback for the next timer cycle. The improved quantum keeper should be a native CCI object. This will allow models to find the correct quantum keeper in the hierarchy. It also avoid the need to pass a quantum keeper to all models whether they need it or not which would be an invasive change to large quantities of models. This approach will allow new models to be written, taking advantage of the new quantum keeper, and removing dependencies on the quantum duration, while allowing co-existence with existing models.

VI. Improvements to the existing TLM-2.0 standard

Each TLM transport class is based on two interfaces : one for forward and another for backward communication. The forward interface inherit from four interfaces all containing exclusively pure virtual methods. These classes contains methods for blocking/non blocking transport but also for DMI (Direct Memory Interface) and debug. This means that the DMI and debug methods must be implemented even if they are not applicable. As we showed for signal, RS232 and RS485 protocols, the DMI mechanism makes no sense. To solve the issue, we purpose to re-factor the current TLM standard and we believe this can be done while maintaining backward compatibility.

Typically, "One to Many" protocol will inherit from `tlm_base_fw_transport_if` and `tlm_base_bw_transport_if` without inheriting from DMI interfaces. The DMI interface should only be used when it is suitable for the interface concerned. Our conclusion is the debug is always valuable and makes (some) sense. Hence, we recommend all interfaces to inherit debug. However, DMI makes no sense for signals, UART based protocols and CAN. On the other hand, the DMI interfaces does make sense for memory like nodes with an I2C or SPI interface. These protocols support a sort of burst read/write. The authors of the document

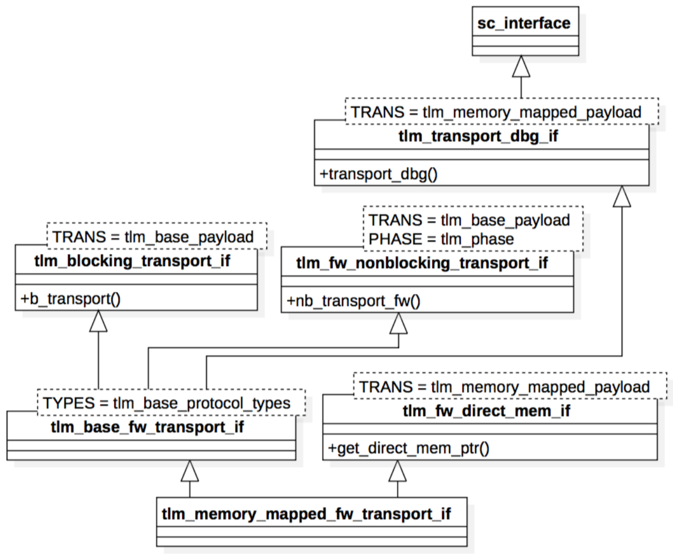
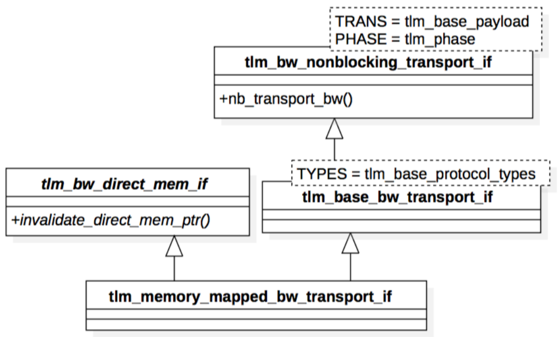


Fig. 2: Transport Forward interface fitted for memory and non memory mapped interfaces



TRANS = tlm_base_payload
PHASE = tlm_phase
tlm_bw_nonblocking_transport_if
+nb_transport_bw()
tlm_bw_direct_mem_if
+invalidate_direct_mem_ptr()
TYPES = tlm_base_protocol_types
tlm_base_bw_transport_if
tlm_memory_mapped_bw_transport_if

Fig. 3: Transport Backward interface fitted for memory and non memory mapped interfaces

[3] preferred to implement new interfaces inheriting only from the blocking transport interfaces. However, we consider the blocking and non blocking transports as one of the key cornerstones of TLM-2.0. Hence we build on `tlm_base_fw_transport_if` and `tlm_base_bw_transport_if`.

The base socket classes (initiator and target) add a notion of bus width as a template parameter (`BUSWIDTH`) with a default value equal to 32. These classes also implements a method to retrieve bus width. This is a significant issue. Our proposal is to refactor this providing a more generic initiator socket as illustrated on Fig. 4 for the initiator socket. A more generic base class `tlm_base_generic_initiator_b` has been implemented to replace `tlm_base_initiator_b` removing related bus width fields and methods. Convenience sockets for non memory mapped protocols can be implemented inheriting from `tlm_base_generic_initiator_socket` without redefining port related methods. The TLM-2.0 class has been left for backward compatibility using the new proposed base classes.

A. *Socket and binding*

The TLM-2.0 core interfaces pass transactions from initiators to targets using blocking or non-blocking transport. As found in Section III, many serial protocols requires bidirectional communication, for instance those based on full duplex UARTs. GreenSocs has an implementation of a TLM bidirectional socket in the GreenSocket library (a part of GreenLib). The GreenSocket implementation instances a pair of initiator and target socket in a single top level socket. As GreenSocket is based on the TLM standard, this implementation is considered as a good candidate for supporting backward compatibility. From a users perspective, it simply looks like a bidirectional TLM socket.

For all the “One to Many”, “Many to Many” protocols a solution to the binding and routing problem, just like with a normal memory mapped bus is to use a router as has been mentioned in Section III-C. This is computationally efficient (as it eliminates the need for each target to check the relevance of each transaction). “Addresses” can be associated with targets as they should be, and the router itself can be relatively generic and lightweight. It can also be used at both levels of abstraction: at the AT level it is used to calculate collisions and delays on the transport medium.

However, for completeness it is useful to see what solutions exist for “routerless” simulation of broadcast protocols. In that case a bidirectional multiple socket is required. The current TLM standard defines single initiator and target sockets but also multiple initiator and target sockets. It doesn’t define bidirectional sockets and TLM multiple sockets only permit a single socket to be bound to multiple sockets. We believe these should be added (potentially drawing on the GreenSocket library).

GreenSocket sockets would be suitable for SPI for instance. For “Many to Many” protocols we require both multiple-socket binding for broadcast and bidirectional data flow. In this case, the TLM standard doesn’t offer a ready to use socket. However, GreenSocket also defines a multiple bidirectional socket class enabling multiple bidirectional binding. This solution would be suitable to bind all nodes of a broadcast based “Many to Many” protocol like the CAN bus.

B. Payload

On top of the current communication mechanism, TLM-2.0 defines a “generic payload” (`tlm_generic_payload` class). Again the naming is somewhat misleading as it is actually an example memory mapped bus. It is however the case that some considerable care was taken to allow many simple memory mapped buses to be modelled at least at the LT level using this “generic payload”. It does not explicitly cover many bus features, but rather allows an 'extension' mechanism to allow those to be modelled. We have noted above the usefulness of this extension mechanism, and recommend it's use in all protocol classes. For a “One to One” serial protocol implementation, address, byte enable, byte enable length and other fields of the generic payload are useless and add confusion in the transaction. However we have also noted the usefulness of the Data structure for some protocols. Overall there are certain aspects of the Generic Payload that are useful for different protocols. Fortunately, although TLM-2.0 defines the TLM generic payload, it's clearly stated that it's possible to implement a different protocol which will clearly be non-interoperable with the generic payload. However, implementing a new payload, and re-using elements of the Generic Payload is current hard with the current standard. The extension mechanism and other payload fields are in the same unique generic payload class. It is not possible to define a new payload without duplicating some of the features of the original generic payload. This is a pity as the generic payload gives us the start of a pallet of fields that can be re-used in other protocols, again increasing interoperability. To solve this issue we propose to re-factor the generic payload. The idea is to separate payload fields and the extensions mechanism. We believe the extension mechanism should form the basis of all payloads, hence the extension mechanism is exclusively defined in a new class called `tlm_base_payload` as in Fig. 5. It allows specific payloads to inherit from this class. Then, for a memory mapped payload, a `tlm_memory_mapped_payload` class is defined inheriting from `tlm_base_payload` which only reuses the fields specifics to the (generic) protocol. We leave to future work the mechanism by which fields themselves can be standardized and re-used, probably using a macro approach like the extension mechanism.

Some issues with the terminology associated with the TLM-2.0 standard have been noted. TLM-2.0 proposes the bus protocol as the most interoperable layer of the standard. Calling this protocol “generic” is at best misleading and at worst counter productive as it leads people to believe that TLM-2.0 only addresses such buses. Hence we propose the new memory map payload should not use the name “generic” but instead `tlm_memory_mapped_payload`. For backward compatibility, a `typedef` between the new definition and the old payload name could be provided.

Most of the serial protocols could be covered by a single definition for serial payloads, with the exception of any protocol specific “commands”: for example, for the I2C protocol, an implementation will be a class `tlm_i2c_payload` inheriting from `tlm_base_payload` and adding `cmd`, `data_ptr`, `data_length` and `address` with associated setters and getters as illustrated in Fig. 5. However the phases of many of the protocols are different, depending on how the protocol is constructed.

A payload also defines the TLM responses status and commands. The current TLM implementation defines a default enumeration for both. However the enumeration does not cover the needs of each protocol as noted above it is likely to be protocol specific. The TLM reposes status on the other hand is perfectly adequately defined in the TLM-2.0 standard as it is essentially used to manage the mechanics of the TLM interface itself.

C. Phases

TLM default class for phases is named `tlm_phase`. Methods of this class are based on the default enumeration `tlm_phase_enum` which defines four phases. It's currently possible to add new phases on top of the default one through a macro. However, as described in serial protocols analysis, the default TLM phases are not appropriate for every protocol. Again this means we can't re-use the `tlm_phase` class, and it is necessary to define a new enumeration specific to each protocol. However, in order to improve class inheriting and to avoid code duplication, we propose to add a template parameter to the `tlm_phase`, the TLM phase enumeration. We will then be able to alias `tlm_phase` for backward compatibility. Again we suggest that, as far as possible, Phase names are kept within a pallet.

D. Generic Serial Protocol

A TLM-2.0 protocol is a structure containing the payload type and the phase type. The default TLM protocol uses the “generic” payload and the (generic) `tlm_phase` class. The Generic Protocol is an abstract “bus like” protocols. For many it has been helpful to get going with TLM. However it does not model the specifics of a bus

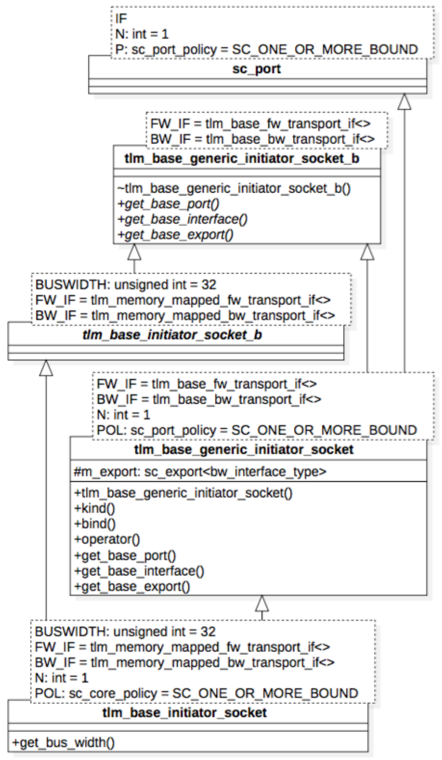

IF
N: int = 1
P: sc_port_policy = SC_ONE_OR_MORE_BOUND
sc_port
FW_IF = tlm_base_fw_transport_if<>
BW_IF = tlm_base_bw_transport_if<>
tlm_base_generic_initiator_socket_b
~tlm_base_generic_initiator_socket_b()
+get_base_port()
+get_base_interface()
+get_base_export()
BUSWIDTH: unsigned int = 32
FW_IF = tlm_memory_mapped_fw_transport_if<>
BW_IF = tlm_memory_mapped_bw_transport_if<>
tlm_base_initiator_socket_b
FW_IF = tlm_base_fw_transport_if<>
BW_IF = tlm_base_bw_transport_if<>
N: int = 1
POL: sc_port_policy = SC_ONE_OR_MORE_BOUND
tlm_base_generic_initiator_socket
#m_export: sc_export<bw_interface_type>
+tlm_base_generic_initiator_socket()
+kind()
+bind()
+operator()
+get_base_port()
+get_base_interface()
+get_base_export()
BUSWIDTH: unsigned int = 32
FW_IF = tlm_memory_mapped_fw_transport_if<>
BW_IF = tlm_memory_mapped_bw_transport_if<>
N: int = 1
POL: sc_core_policy = SC_ONE_OR_MORE_BOUND
tlm_base_initiator_socket
+get_bus_width()

Fig. 4: Generic TLM initiator socket with backward compatibility

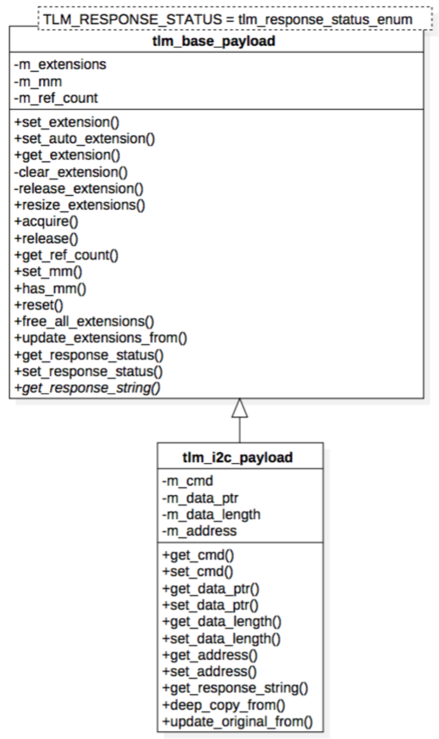


Fig. 5: TLM I2C payload inheriting from TLM Base Payload

protocol such as OCP, for which a dedicated kit is needed. These features are present in the OSI Layer three (LT) as much as they are in Layer two (AT).

The OCP [5] kit has been constructed in such a way as to allow models to 'degrade' gracefully to the generic protocol. This way, an IP block written using the OCP kit can be used in a different context, and will behave without the extra features that OCP brings. There are a complex mixture of OCP features some of which are mandatory for the communication to work, others are not. The OCP kit allows model writes the flexibility to assert which features they rely on, then during bind time, the models will automatically "configure" themselves, potentially reporting errors. The technology for this exists within TLM-2.0 (based on the extension mechanism). It can be deployed for any family of interfaces. To this end, we could conceive of a 'Generic Serial Protcol', and then specific protocols could build on that, just as OCP builds on the Generic (memory mapped bus) Protocol.

VII. TLM Interface Kit pattern

The Accellera OCP SLD kit is a 'full' TLM kit available openly. The kit is based on TLM-2.0's Generic Protocol, providing a set of extended phases and payload extensions. OCP is a (very) large family of protocols with many options, the kit supports the automatic negotiation of which options will be used in communication, reverting to the generic protocol if appropriate. In addition the kit provides transactors to go between the physical layer (OSI layer 1, OCP TLM 1, TLM "CC"), AT and LT levels of the protocol. In doing so, the kit can be directly used in mixed simulations. The kit also contains protocol checkers and analysers, examples and documentation.

Taking the OCP kit as the basis of what should be expected in a TLM interface kit, we would suggest the following:

- OSI analysis : An analysis of the protocol in terms of it's OSI layer 2/3 characteristics. This should directly guide the choice of payload and phases, and should form part of the documentation of the kit. This is absent from the OCP kit.
- Protocol : This is the only mandatory part of the interface definition as it defines the API. The protocol phases and payload must be defined. Specifically this means for a payload defining all the fields of the transactions (ideally selecting from the pallet of already defined fields) and associated setters/getters inheriting from `tlm_base_payload`, implementing `get_response_string()` and defining `deep_copy_from` and `update_original_from()` methods. An enumeration of TLM response status has to be defined and

passed as a template parameter to `tlm_base_payload`. A phase class specific to the protocol also need to be defined (except if the default phases are ok), specifying an enumeration of phases as a template parameter of `tlm_phase` class. Again, where possible the phases should be defined referencing the phases defined by the TLM WG.

- Convenience sockets : Protocol kits should provide convenience sockets to cover common use cases.
- Transactors : In order to support software protocol emulation, RTL or even real hardware, a transactor down to OSI layer one (physical) should be provided. This becomes especially important for interfaces such as SPI that are commonly implemented using GPIO where there will be a need to abstract up from the lower level.
- Host Bridges : In addition to connecting to RTL or real hardware, for many interfaces it also makes sense to provide connectors to host interfaces. In other words an Ethernet interface kit might provide a way to connect to the hosts Ethernet.
- UVM (Universal Verification Methodology) : UVM-SystemC [12] is currently under public review; interfaces kits should provide transactors to work with this standard (this is currently missing from the OCP kit).
- Routers : For routed and broadcast protocols a router provides a convenient way to model the transport medium, and should be provided in the kit as it's a "generic" part for the protocol.
- Documentation and Examples. Note that the documentation will 'inherit' mostly from the TLM documentation itself. To the degree that phases and payload fields are used from the pallet of existing items, their documentation can also simply be referenced.
- Legal : Though listed last, this is perhaps the most important feature, the interface kit should make it clear under what license the kit is made available (and the writes that have been given to construct the kit in the first place). IP that uses this interface kit will become a derivative work, hence the license is exceedingly important.

VIII. Conclusion

This paper has proposed a formal approach to defining the transaction in transaction level modelling, working towards a common understanding of how protocols should be modelled. We have introduced the use of the OSI layered communication model as the basis for defining transaction. We have analysed a number of protocols and shown how the OSI model can be used to identify the transaction, and guide the construction of an interface kit.

We have gone on to look at how time is modelled and synchronization between time domains is achieved, finding that there are currently ambiguities in the standard and different approaches. We have proposed a new Quantum Keeper that provides an event queue that we believe will assist in simplifying the differences between these approaches, and allow more efficient models.

We have examined in some detail the existing TLM-2.0 library and made concrete proposals about re-factoring the code to better allow more interoperable interface kits to be written, covering a wider range of protocols, including non memory mapped and serial protocols.

Finally we have provided a blue print for the contents of a full TLM interface kit, with reference to existing standard kits already available from Accellera (namely the OCP SLD kit).

Future work will focus on concrete implementations and proposals of serial protocol standards for TLM. Our contributions are planned to be open source.

References

[1] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.

[2] S. Swan, V. Motel, J. Cornet, and L. Maillet-Contoz, "Beyond TLM 2.0: New Virtual Platform Standards Proposals," in *Proceedings of DAC 2012*, Jun 2012.

[3] R. Swaminathan and V. Goel, "TLM Signal: A non-memory mapped bus model," in *Proceedings of ISCUG 2013*, Apr 2013.

[4] "Open System interconnection," International Organization for Standardization, Standard, Nov 1994.

[5] Accellera. OCP Modelling Kit. [Online]. Available: http://accellera.org/downloads/standards/ocp

[6] "ANSI/TIA/EIA-232-F," Telecommunications Industry Association/Electronic Industries Alliance, Standard, Mar 1998.

[7] "ANSI/TIA/EIA-485-A," Telecommunications Industry Association/Electronic Industries Alliance, Standard, Mar 1998.

[8] GreenSocs. GreenLib. [Online]. Available: https://git.greensocs.com/greenlib/greenlib

[9] J. Cornet and M. Schnieringer, "What is needed on top of TLM-2 for bigger Systems?" in *Proceedings of DVCon EU 2015*, Nov 2015.

[10] G. Delbergue, M. Burton, B. Le Gal, and C. Jego, "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0," in *Proceedings of ERTS 2016*, Jan 2016.

[11] D. Becker, M. Moy, and J. Cornet, "Challenges for the Parallelization of Loosely Timed SystemC Programs," in *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2015.

[12] S. Schulz, T. Vrtler, and M. Barnasconi, "UVM goes Universal - Introducing UVM in SystemC," in *Proceedings of DVCon EU 2015*, Nov 2015.