Defining TLM+

Wolfgang Ecker
Infineon Technologies AG
85579 Neubiberg, Germany
Wolfgang.Ecker@infineon.com

Volkan Esen Robert Schwencker
Infineon Technologies AG
85579 Neubiberg, Germany
Firstname.Lastname@infineon.com

Michael Velten
Infineon Technologies AG
TU München
Michael.Velten@infineon.com

***Abstract*—Virtual Prototypes (VPs) based on Transaction Level Modeling (TLM) have become a de-facto standard in today's SoC design, enabling early SW development. However, due to the growing complexity of SoC architectures, full system simulations (HW+SW) become a bottleneck. Hence, it is necessary to develop modeling styles which allow for further abstraction beyond the currently applied TLM methodology.**

This paper introduces such a modeling style, referred to as TLM$^+$. First applications of TLM$^+$ in an industrial virtual prototyping project show a speedup of several orders of magnitude compared to state-of-the-art TL modeling.

I. Introduction

Transaction Level Modeling (TLM) is the de facto standard for developing so called Virtual Prototypes (VPs). As VPs are ready much earlier than the hardware (HW) and provide the same programming interface, the software (SW) development can start much earlier as well. This shortens the overall development time for a new product. However, with the steady increase of complexity in SoCs, the VP simulation performance becomes a bottleneck which reduces this time benefit.

This bottleneck starts already to effect the development of complex SoCs such as a mobile phone platform. The lack of simulation performance for running e.g., protocol stack SW on a VP increases the SW development time drastically, as modeling a complete SoC at TLM abstraction still leads to huge amounts of detail to be simulated.

In order to counteract this problem, further abstraction techniques are required which go beyond the current TLM approach. Therefore, a new modeling style called TLM$^+$ is introduced which allows for raising the level of abstraction but keeping the required level of detail for SW development and still preserving the overall architecture of the design for providing required HW details like clock, reset, and power control.

This TLM$^+$ style represents a combination of three new modeling concepts which are introduced and described in this paper:

- Transaction to transfer abstraction
- Bit true to content true abstraction
- Strong separation of functionality and timing

The key idea of the transaction to transfer abstraction is to increase the atomicity of actions, i.e., HW constraints such as bus widths and burst lengths are neglected with SW data packages being transferred instead. Therefore, a new interface concept is introduced which represents a merge of the device driver SW and the corresponding peripheral bus interfaces while being transparent to the higher-level SW.

For further improvement of the simulation performance we introduce the bit true to content true abstraction which helps reducing the amount of number-crunching operations to a minimum. Hence, this abstraction is especially applicable to communication oriented applications (e.g. UMTS, ciphering, etc.).

Since timing influences system functionality and since satisfaction of timing constraints is essential in embedded systems, a quite accurate timing representation is important for TLM$^+$, too. For this purpose we introduce a so called resource model (RM) which enables the strict separation of functional and timing computations in order to distinctively control and correct timing behavior of a TLM$^+$ model.

The paper is structured as follows. First, related work is discussed followed by a clear definition of the TLM$^+$ modeling style. In connection to that a generic CPU model for host code execution is introduced which represents a prerequisite to most applications of TLM$^+$. After that, a detailed description of the TLM$^+$ modeling concepts and techniques are described. Furthermore, the introduced concepts are analyzed in the context of an industrial application example and experimental results are presented followed by a brief summary and an overview on the next steps.

II. Related Work

Transaction Level Modeling is the de-facto standard for creating Virtual Prototypes. Open SystemC Initiative (OSCI) has released two standards for Transaction Level (TL) modeling up to now [1]. These standards define different communication concepts for modeling hardware interfaces. Especially, with the TLM2 standard timing abstraction techniques were introduced to boost simulation performance. These OSCI standards are complementary to the approach introduced here as the abstraction is obtained by block based transactions which can be modeled using the OSCI standards.

The approach presented in [2] connects the QEMU processor emulator to SystemC TL models to enable driver and SW development. Also several EDA companies provide high speed processor models for VP design, e.g. VaST or CoWare. These models and QEMU are instruction set simulators (ISS). Our approach is to merge the HW/SW interface to enable further communication abstraction. Due to this merge, an ISS

can no longer be used. However, we are providing mixed TLM/TLM$^+$ support. Hence, TLM$^+$ abstracted subsystems can be combined with these ISS based approaches.

The SystemQ approach presented in [3] provides high-speed simulation models based on queuing networks. These models are mainly used for system performance estimation and cannot be used for SW development.

Several approaches presented in [4], [5], [6] target the development of fast and timed Real-Time Operating System (RTOS) simulation models to increase the simulation speed. These approaches are increasing the simulation speed due to native software execution of the RTOS models in combination with e.g., SW timing annotations or task scheduler models.

The authors of [7] are presenting the generation of timed OS simulation models using delay annotation for the SW execution. These OS models are communicating through bus functional models with the HW model. The OS models presented in [8] are dealing with synchronization problems of SW and HW. A timed HW/SW co-simulation at an early design stage which allows simulation performance up to 3 orders of magnitude faster than using an ISS is presented in [9]. Other approaches target automatic timing annotations of the native SW execution. A compiler based approach is presented in [10]. In [11] the SW execution time is derived from a static analysis and combined with dynamic runtime information in order to achieve Cycle Approximate (CA) simulations of the native SW execution. A combination of instruction set simulation and an abstract RTOS is presented in [12].

None of these approaches deal with an abstraction of the HW models. In contrast to that, our TLM$^+$ abstraction considers both, the HW and the SW models due to the merge of the low-level device driver SW and the HW model. However, since these other approaches deal with areas of optimization not or only marginally covered by our approach, it might be possible to combine them to increase the timing accuracy and system performance of the native SW execution.

III. TLM$^+$ Definition

Prior to describing the various new techniques in detail it is necessary to explain the exact notion of the TLM$^+$ abstraction and how it relates to other abstraction levels. Figure 1 outlines the major modeling related abstraction levels.

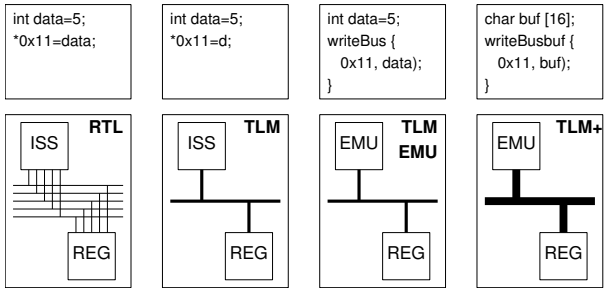

Fig. 1. Abstraction Chain

Our view of the abstraction chain starts with the RTL level which is a cycle, bit, and architecture accurate model representation of the hardware. At RTL, communication between different blocks and components is described in terms of signal protocols. The protocol in turn is described and simulated at bit level. Due to the high level of detail a SoC at RTL abstraction cannot be used as development model for e.g. software. The simulation effort required for this task is just too high. In this case it makes more sense to wait for the first engineering samples. However, this creates a high sequential dependency between SW and HW development during the design cycle and impacts the time-to-market window.

These disadvantages have led to the establishment of TLM. Here, communication between blocks is captured with function calls, clocked synchronization is replaced by event based synchronization, and abstract integer value and complex data types are used instead of bit types. All this techniques applied yield a more abstract model of the overall system, with much less detail and hence, better simulation performance.

In TLM, transactions are modeled by virtual function calls and payload objects, instead of signal based protocols between hardware peripherals. A serial interface (SIF) is a good example to explain the TLM abstraction: At RTL the SIF is controlled by a core via a bit accurate bus protocol like AMBA. The state machines within the SIF control the buffering of transmission (TX) and reception (RX) data, and furthermore, control the physical layer path which applies/interprets transmission/reception protocol information to/from the data stream. For instance, each data byte is extended by delimiters and the whole character is transmitted bit-by-bit, each consuming one clock cycle. At TLM the CPU core makes a transaction (a simple function call) over the bus to the SIF. The SIF also applies the physical layer information to the data and uses a transaction containing the complete data and the protocol information as payload in order to communicate to another serial device. Hence, the overall character is transmitted at once in contrast to bit-by-bit. This however, requires also that timing has to be modeled in an abstract way, to still being able to capture the relevant timing information. Since in TLM a high simulation performance is mostly achieved by avoiding a toggling clock, and hence, numerous simulation cycles, timing is modeled in different ways, as for instance, by timing annotations and simulation time based event notifications. As an example, a transaction payload can contain a field for tracking the virtual time a transaction consumes. Each hardware module and bus which is involved by such a transaction adds its timing increment to this information. By continuing the accumulation of these values further, the actual system time needs only be executed when processes from different timing domains need to be synchronized, at so called synchronization points.

Still, a pure TLM based VP of a complete SoC is too expensive with regard to a complete simulation. A first step to improve the simulation performance of such a VP for faster system simulations is to replace the instruction set simulated core model by a host-simulation wrapper model, which allows a direct execution of the SW on the simulation host. This core abstraction leads to better simulation performance however, the granularity of timing information is reduced as the SW is

now no longer interruptible per instruction, but rather per bus access. This abstraction is also depicted in Figure 1 and forms the basis for the continuation of abstraction towards TLM$^+$. Details with regard to this approach are explained further in Section IV.

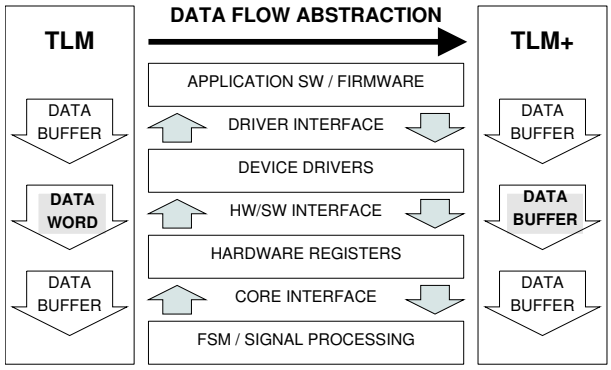

Fig. 2. Transaction to Transfer Abstraction

TLM$^+$ represents a transaction to transfer abstraction on top of TLM in combination with host-executed SW and hence, continues the abstraction chain as shown in Figure 1 to a higher level. Figure 2 depicts the levels of communication within a system from the application SW down to particular HW blocks. Data going in between has to pass several layers.

Starting from the application SW which allocates a buffer of data to be processed by the HW the corresponding request and the buffer are passed down to the driver of the required device. The driver in turn translates the request into control values for the HW and performs sequences of accesses on the HW/SW interface in order to initiate the HW. The data to be processed is also tailored into smaller pieces, depending on the data width and burst capabilities of the underlying bus system. Finally, the data is also sent to the HW through several bus accesses. Within the HW however, the data to be processed is again reassembled to bigger frames or blocks before starting the actual operation.

Simulating all these activities within a VP, requires a non-feasible amount of time as the complexity of the targeted platforms continues to grow, even though it is modeled at the TLM abstraction even with host-executed SW. Hence, the key idea of the TLM$^+$ abstraction is to reduce the number of communication activities in order to reduce the overall simulation time. This is achieved through data abstraction, basically by avoiding most of the activity at the HW/SW interface. As the data blocks at the level of application SW are reconstructed down in the HW, the separation into smaller units of data within the device drivers and the HW/SW interface and the reconstruction into buffers can be avoided. Moving to TLM$^+$ can be achieved through a combination of the following steps, which are explained in detail in Sections IV and V:

- Insertion of an interruptible and timing aware host-simulation wrapper (EMUCPU)
- Introduction of a new HW/SW interface at driver as well as peripheral level

The first item represents a precondition for the TLM$^+$ data abstraction, as introducing a more abstract HW/SW interface requires a merge of the low-level driver SW with the associated peripheral interface which cannot be accomplished if the SW is cross compiled to instructions. The latter item addresses the actual TLM$^+$ modeling techniques. It incorporates techniques on how to cut at driver level and incorporate a new HW API based on block transfers, while preserving the original architecture and providing means for a migration path from TLM.

In order to provide further improvements to the simulation performance the TLM$^+$ style offers the bit true to content true abstraction. Here, the data is not encoded at one side of the system and decoded at the other but transferred directly including its configuration parameters. The combination of this abstraction with the data flow abstraction is obvious, since data packages are rather related to logical entities such as OS buffer sizes or data content sizes such as pictures. The bit true to content true abstraction is orthogonal to the data flow abstraction as it is shown in Figure 3. The figure shows

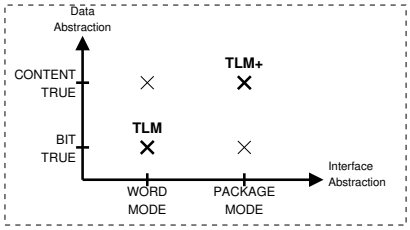

Fig. 3. Data versus Interface Abstraction

combination possibilities of the word and package mode with bit and content true data. As it is shown, word mode combined with bit true abstraction represents today's TLM. The combination of the two newly introduced abstraction concepts represents TLM$^+$. Other combinations are also possible e.g., for providing a migration path from TLM to TLM$^+$.

TLM$^+$ block transfers decrease the timing accuracy of the simulation model even more than for instance TLM2 burst transactions because the transported data blocks of TLM$^+$ transfers are not limited by the constraints of the underlying HW communication architecture. Hence, bigger chunks of data are transferred by one function call, and the transfers are atomic. With the third TLM$^+$ concept, namely the separation of functionality and timing we introduce a new auxiliary unit called resource model (RM). The RM is responsible for ensuring the best possible timing accuracy with regard to regular TLM, while preserving the speed obtained by moving to block transfers. The RM keeps track of resource requests and handles resource conflicts by applying timing corrections according to a prioritization scheme.

IV. Native Software Execution

Native software execution is the basic requirement for our TLM$^+$ methodology to enable interface abstractions at the HW/SW interface. The merge of software and hardware at the HW/SW interface is not possible with an ISS CPU model

because an ISS is restricted to its maximum data width. Therefore we developed an EMUCPU model in SystemC which supports synchronization mechanisms like interrupt handling and software timing annotations. The EMUCPU does also support multiple instances in order to form a multi core system even in combination with instruction set simulator (ISS) CPU cores. The EMUCPU model is highly generic and can be configured to replace any type of ISS CPU model. The following sections illustrate the main concepts implemented in our EMUCPU.

A. HW/SW Interface

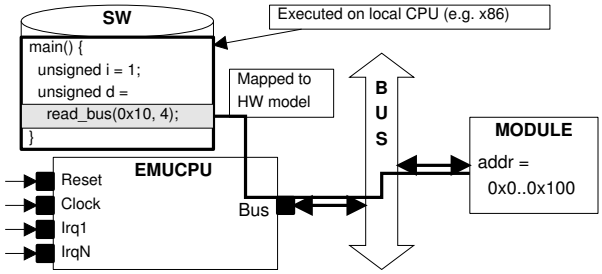

Fig. 4. EMUCPU Model and HW/SW Interface

Figure 4 gives an overview of the EMUCPU model and how the HW/SW interface is modeled. As shown, the software is executed on the simulation host CPU. Software accesses to memory mapped IO are modeled via the functions `read_bus` and `write_bus`. These functions are wrapped to access the bus initiator port of the SystemC EMUCPU model. Hence, the C-software can directly access registers and memories of HW modules. In general, to avoid consistency issues it is recommended to generate register, bit field and memory access functions based on a meta model specification like IP-XACT. If the application and driver software uses these functions or macros for accessing the hardware, switching from an ISS to the EMUCPU can be achieved in negligible time, because only the implementation of these access functions has to be changed to call the bus access functions `read_bus` and `write_bus` from the EMUCPU. Here, again if generators are applied to create the various access functions, rerouting these to the EMUCPU bus access functions can be accomplished quickly.

B. Software Timing

Native execution of the SW on the simulation host inevitably leads to differences in timing behavior in comparison to an ISS based SW execution. Apart from timing added by the system infrastructure and peripherals an ISS introduces additional timing into SW due to its instruction execution cycles. This information is lost when moving to the host execution approach. This drawback is tackled by two timing mechanisms supported within our EMUCPU model. On one hand, each bus access initiated from the SW is blocking and hence suspends the SW execution for the duration of the access. On the other hand, the programmer has the possibility to annotate the SW with a parameterized wait function to force additional SW suspension. Automated solutions for software timing annotations may be applied as well.

C. Interrupt Handling

The EMUCPU model provides a generic configurable number of interrupts and an interface for configuring the priority level of each interrupt. The main software and the interrupt service routines are executed in the context of one *SC_THREAD*. The main software thread needs to be suspended in case of the occurrence of an interrupt. In case of an ISS CPU model the software execution can be interrupted at the granularity of instructions or even more fine grained. In contrast to this, SW running on the EMUCPU model can only be interrupted when the software execution is suspended by the aforementioned means. When serving an interrupt the EMUCPU model automatically corrects the wait times of the main SW as the interrupt service routine consumes time as well. A detailed description of the interrupt handling is given in [13] and hence, is skipped here.

D. Multi Core Support

Today's systems mostly contain more than one CPU core. For instance, CPU cores of the same type or different CPU cores in heterogeneous systems. Therefore, the EMUCPU model provides multi core support and can be instantiated multiple times within a VP. As linking the Software to the SystemC module has to happen through regular C functions, a mechanism is necessary which detects which EMUCPU instance is executing the C function which requests a bus access. This is solved by obtaining the currently active *sc_module* through accessing the simulation context API of SystemC.

V. TLM$^+$ Modeling Concepts

This chapter describes the aforementioned steps towards implementing the TLM$^+$ abstraction within the following sections. Figure 5 illustrates an example system which is used to describe the TLM$^+$ abstraction concepts in the following sections. The example system includes two different interfaces - one for the bus communication and one for the external interface of the SIF which is connected to the IO device. In Section V-A the TLM$^+$ interface abstraction concepts are

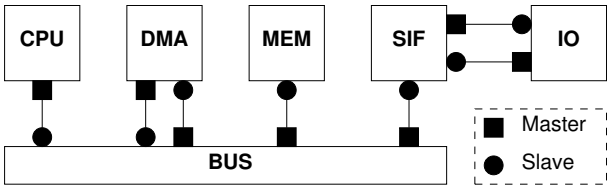

Fig. 5. Explanation Example

described in detail. Following that Section V-B explains the bit true to content true TLM$^+$ abstraction concept. At the end in Section V-C the concept of the separation of timing and functionality is described by introducing the resource model (RM).

A. TLM$^+$ Interface Abstraction

In this section transaction to transfer abstraction is described. Here, data blocks are transferred instead of single words. In contrast to state-of-the-art TLM modeling style, the data blocks are not related to infrastructure details as e.g., bus transaction burst size. The data blocks are rather related to logical entities such as OS buffer sizes or data content sizes such as pictures. This abstraction technique leads to a huge speedup since the HW/SW and HW/HW interactions are reduced to block accesses at OS level. Nevertheless, the programming technique is quite close to the concepts applied in TLM for modeling burst transactions. Hence, OSCI TLM1 or TLM2 libraries can be used for the implementation of the TLM$^+$ interface abstraction concepts which is explained in the following.

The TLM$^+$ interface abstraction is applied at the HW/SW interface for enabling direct block transfers from the SW to the HW models of the VP. The abstraction of the HW/SW interface is described in Section V-A1. Following that, the abstraction of the TLM interfaces is described to allow block communication between all HW modules and SW. The TLM interface abstraction is described in detail in Section V-A2

1) HW/SW Interface Abstraction: The requirement to allow for further abstractions of the HW/SW interface was described in Section IV by introducing an EMUCPU SystemC model which provides a function interface (`read_bus` and `write_bus`) between the SW and the HW model. In TLM$^+$ the EMUCPU model is extended by additional interface functions which provide block transfer capabilities. Listing 1 shows the abstract functions which enable the transfer of complete SW buffers to the HW model. The `count` parameter of these functions yields the number of bytes which are to be transferred.

```
void write_bus_pkg(uint32_t addr, char* data,
    uint32_t count);
char* read_bus_pkg(uint32_t addr, char* data,
    uint32_t count);
```

Listing 1. Abstract HW/SW Interface Access

The reasons for not replacing the existing access functions with the abstracted functions is to be able to switch dynamically between word and block accesses. Hence, the control flow can be modeled using word accesses to setup the necessary HW details while data blocks are transferred using the abstract interface functions. Another reason is that TLM and TLM$^+$ modules can be mixed within one embedded HW/SW system because modules which are not TLM$^+$ abstracted can be accessed by the SW using the word based functions.

At the SW side normally, device drivers offer read and write buffer access to the HW models. The SW application can call the interface function of the device drivers by passing a data buffer and its length as arguments. Normally, the device driver cuts the buffer into data words and transfers each of them separately to the HW device. In TLM$^+$ the device driver functions directly pass the buffers to the HW model using the introduced abstract interface functions.

As an example if the SW wants to write a buffer containing 4000 bytes to the SIF the complete buffer is transferred by the device driver using the TLM$^+$ interfaces instead of transferring 1000 single words. Also the control flow and interrupt details are simulated but only once for the complete buffer.

In the following section the abstraction of the TLM interfaces to provide block based communication within the VP is explained in detail.

2) TLM Interface Abstraction: This section describes the abstraction of the TLM interfaces for the bus communication and other HW interfaces like the external interface of the SIF module. The interface concepts of the OSCI TLM1 or TLM2 libraries can also be used for TLM$^+$ because only the payload classes need to be extended to implement the TLM$^+$ interface abstraction. Hence, TLM$^+$ does not require its own modeling library but can be realized using the existing library and interface concepts. So to say, the *plus* of TLM$^+$ is a definition of the payload structure, and a recommended practice on how to use the defined payload items.

In case of TLM1 special payload classes need to be defined for bus communication or other HW interfaces. If for instance the blocking transport interface is used for bus communication the request and response classes need to be extended by the `count` member which corresponds to the number of bytes of a TLM$^+$ data transfer. This interface member is also used to differentiate between TLM and TLM$^+$ transactions. If the value of `count` is zero then it is a TLM transaction otherwise it is a TLM$^+$ transfer.

In case of TLM2 a generic payload (GP) is predefined which already supports most of the common bus protocols. This GP can be used for TLM$^+$ but with an extension containing the `count` member as well. The extension definition is shown in Listing 2 and can be added with `set_extension` to the GP.

Both in TLM1 and in TLM2 the *data* interface member shall be a pointer to provide word and block transfers. Hence, it is not required to extend the payload by an additional pointer member for TLM$^+$ data blocks. To enable TLM$^+$ data transfers for other HW interfaces like the external interface of the SIF again only the payload definition of this interface needs to be extended.

```
class tlmp_payload_extension :
  public tlm::tlm_extension<tlmp_payload_extension>{
public:
  uint32_t count;
  ...
};
```

Listing 2. TLM$^+$ Generic Payload Extension

As it is shown, the TLM$^+$ interface concepts and extensions are fully compatible to state-of-the-art TLM interfaces. Hence, the TLM$^+$ interface concept provides a migration path of today's TLM to TLM$^+$. It is not required to abstract the whole VP to TLM$^+$ at once. Because of the dynamic detection of TLM and TLM$^+$ transfers it is possible to migrate one module after the other of a VP to TLM$^+$. If the bottleneck of the simulation performance is located in only one subsystem of the complete VP then only this subsystem can be abstracted.

B. TLM$^+$ Data Abstraction

Especially in communication applications data is usually not transferred as raw data. Extra information might be added to support error recognition resp. correction (e.g. parity bit, CRC or channel coding), data might be extended or split into smaller chunks in order to match a protocol or other transformations are being done to support access protection, data separation, etc. Transformations on one end of the communication channel are usually reverted on the other end. Thus, in case the communication protocol is known the data transfer can be modeled on a higher abstraction level by providing:

- the raw data to be transferred and
- control information describing algorithm parameters or results.

By skipping the data encoding and decoding processes and blockwise data transfer, the simulation can be accelerated significantly but transmission error injection as well as checking for correct configuration of hardware accelerators is still possible. Of course the encoding and decoding processes cannot be verified within such an accelerated model as they are not modeled any more.

1) Serial interface Data Abstraction Example: The idea of data abstraction can be illustrated quite well on a simple serial interface characterized by:

- Size of a character (e.g. 1..32 bit)
- Number of stop bits (1 or 2)
- Parity mode (even or odd)
- Good parity or parity error
- The raw data itself

As shown in Listing 3, this serial interface can be modeled using e.g. the OSCI TLM put interface definition in conjunction with a custom payload definition.

```
struct SerialData {
  unsigned data_size_in_bit;
  unsigned number_of_stop_bit;
  bool parity_is_even;
  bool parity_error;
  uint32_t data;
};
void put( SerialData &serial_data );
```

Listing 3. Abstract data transfer across a simple serial interface

Note that data transfer could also be modeled using an `sc_signal<SerialData>` but a value changed event will not be emitted when the same datum is transferred twice and thus it might be missed. Using the suggested modeling style the most important effects on simulation time are:

- Less activity as simulation is only triggered once per character and not once per transferred bit.
- Less calculation as bit stuffing as well as parity calculation can be skipped.

The example of the simple serial interface also shows that the concrete data abstraction is individual to the transmission protocol and also depends on the abstraction level being addressed (e.g. packet transfer instead of character transfer).

C. TLM$^+$ Timing Handling

Since timing influences system functionality and since satisfaction of timing constraints is essential in embedded systems, a quite accurate timing representation is important for TLM$^+$, too. In order to gain high simulation speed, computation of functionality and timing is strictly separated. Timing is computed in a so called resource model (RM), which is considering the resources and infrastructure (e.g. bus, CPU) of the complete SoC architecture and the resulting resource conflicts for the computation of time. This RM performs timing corrections and actively reschedules pending transfers to achieve a good timing accuracy at TLM$^+$ level.

An overview about the resource model and its structure is described in Section V-C1. After that the handling of resource conflicts is explained in Section V-C2.

1) Resource Model Overview: The last TLM$^+$ modeling concept is that the handling of the system timing is separated from the functionality of the system. This is achieved by introducing a resource model (RM) which handles on one hand the timing of the native software execution and handles on the other hand the timing of the hardware VP. Figure 6 gives an overview about the RM for the explanation example. As it is shown in the figure, each module in a VP corresponds to a resource (R1-R6). Each initiating module also corresponds to an initiator (I1-I4). The resource model provides interface

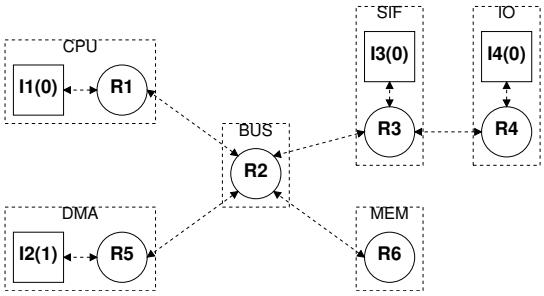

Fig. 6. Resource Model Overview

functions for the registration of resources and initiators during the SystemC construction time. These functions are:

- `int registerResource(string name)`
  This function registers the resource at the RM using its hierarchical name and returns a unique integer identifier for the resource referred to as resource ID (RSID).
- `int registerInitiator(string name, int prio, sc_event* resume_ev)`
  This function registers an initiator at the RM using its hierarchical name, its priority and a sc_event pointer. A unique initiator identifier (IID) is returned.

The resource model itself has no structure but it inherits the structure from the underlying VP. This structural inheritance is achieved during the simulation runtime when an initiator starts a transaction addressing a specific module (resource). Here, each resource which gets passed by this transaction requests itself for a specific amount of time which refers for instance to the block size of the transferred data. The resource model provides the following function:

- `requestResource(int IID, int RSID, sc_time time)`
  This function uses the IID to request the resource which is specified in the RSID argument for the specified amount of time. Usually this function gets called from the corresponding resource itself, e.g., from the transaction interface or state machines.

In the explanation example following requests are executed for a data transfer from the CPU to the memory module. First the CPU resource gets requested using the initiator ID (IID) I1 and resource ID (RSID) R1, after that the transaction passes the bus resource which is requested also using IID I1 but RSID R2. When the transaction reaches its destination the memory resource is requested using IID I1 and RSID R6. Before the actual read or write access is performed the transaction is suspended for the requested amount of time. The RM offers the following function to suspend and resume transactions:

- `waitResumeEvent(int IID)`
  This function suspends the transaction of the specified initiator for the requested amount of time using the sc_event which corresponds to the IID.

Each time a resource is requested using a specific IID the corresponding resume event is scheduled to the new time. The payloads of the TLM interfaces need to be extended further on by the IID which needs to be passed along the transaction.

Each transaction has a forward path (to its destination) and a backward path (returning to the initiator). This results in two synchronization points where the `waitResumeEvent` function needs to be called. One synchronization point is at the destination of the transaction before the actual access is performed and one is when the transaction has returned to the initiator. Using these synchronization points the RM handles resource conflicts and performs timing corrections which is explained in the following section.

2) Resource Conflict Handling: In TLM$^+$ a complete data block is transferred atomically as one TLM$^+$ transfer. Hence, it cannot be interrupted by a transfer of an initiator with a higher priority. This would lead to a wrong timing behavior and could also result in wrong functionality.

Figure 7 shows an example of a resource conflict where two initiators are accessing the same resource during an overlapping time period. Here the CPU (I1) with priority 0 initiates a data transfer to the SIF at 0 ns. Each resource (CPU, BUS, SIF) is requested for 10 ns. At the SIF module the transaction is suspended and resumes at 30 ns.

The DMA (I2) with higher priority than the CPU starts its transfer to the memory at 10 ns. Each resource (DMA, BUS, MEM) is requested also for 10 ns. The transfer of the DMA gets suspended at the memory module and will resume at 40 ns.

This example describes a resource conflict at the BUS which is requested from the CPU transfer and from the DMA transfer. Hence, because of the higher priority of the DMA transfer the resource model needs to consider the conflict by scheduling the resume event of the CPU transfer to a later point of time:

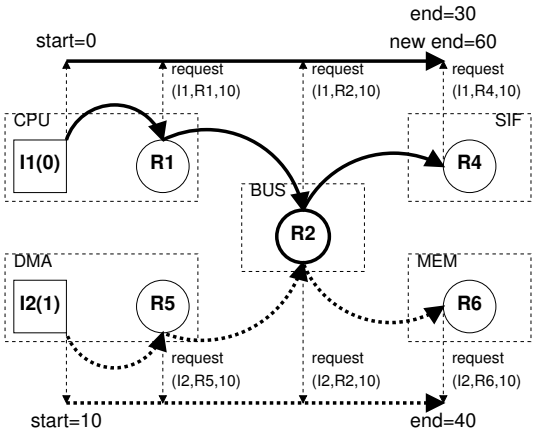

Fig. 7. Resource Model Conflict Handling

The new resume time of the CPU transfer is rescheduled from 30 ns to 60 ns.

With OSCI TLM2 the concept of the quantum keeper is introduced where context switches can be reduced by using a time quantum during which concurrent processes can accumulate time values. One process can perform several subsequent transactions and accumulate the transaction times until the accumulation has reached the time quantum. Then the accumulated time gets synchronized with the simulation time. Hence, a big quantum improves the simulation speed but if it is chosen too big it can lead to wrong simulation results. In contrast to that the RM ensures that TLM$^+$ data transfers are directly synchronized but also resolves resource conflicts by controlling the emission of the various resume events as described earlier.

VI. Application Example

In this section the methodology is demonstrated on an industrial UMTS modem VP (see Fig. 8) that is being used on different abstraction levels for different use cases:

- The fully accurate VP (see Table I) operating with I/Q-samples is being used to develop and verify algorithms distributed across SW and HW. As this includes non-ideal effects like e.g. noise, reflection, etc. low-level data is required.
- In an intermediate version internal data transfers (e.g. from and to memory) are performed on a higher abstraction level as described in section V-A (Block mode VP in Table I).
- An accelerated version with data abstraction on layer1 (air) is being used to develop control software (Data abstraction VP in Table I). In this case, the non-ideal effects are only of minor interest and thus data abstraction (see Section V-B) is used. The data abstraction applied is described in more detail below.
- Further acceleration is achieved by replacing the CPU and DSPs with native code execution. This part is skipped in the application example.

In the next section abstraction of data transfer across layer1 (air) interface for UMTS will be discussed briefly.

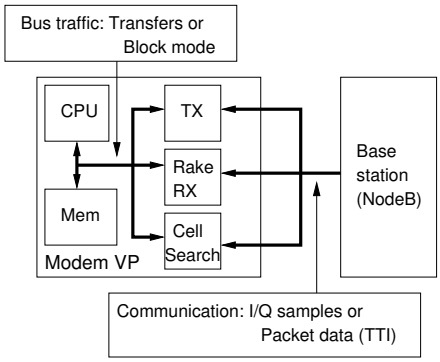

Fig. 8. UMTS Communication Example

A. Data Abstraction

Within UMTS data transfer from the basestation (NodeB) and the user equipment (UE, e.g. mobile phone) is via air. The data is then mixed down into the low-frequency baseband and further signal processing takes place on the I/Q-samples provided in the baseband. Before a transfer the raw data provided to the NodeB is transformed in several steps in order to provide error correction, identification, etc. The same steps are reverted in the UE (see Figure 9). When not interested in the exact operation of the encoding and decoding algorithms themselves it is sufficient to transfer the data and some additional control information as indicated as "Data Abstraction" in Figure 9. Within the data abstraction interface the I/Q-data is

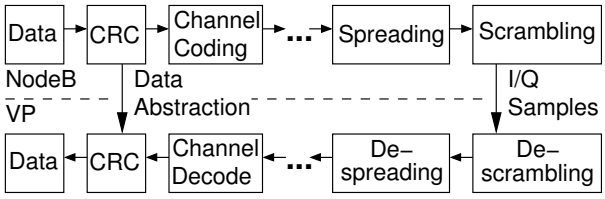

Fig. 9. Transformation steps and data abstraction in UMTS RX path.

not transferred but rather the raw data including some control information like scrambling code, spreading factor, etc. Thus the UE has all information available in the I/Q-data.

	Full accurate VP	Block mode VP	Data abstraction VP
Simulation speed [10ms frames/sec]	0.031	0.201	3.310
Speedup	1.0	6.5	106

TABLE I
SIMULATION PERFORMANCE COMPARISON

Table I compares the simulation performance measured in UMTS frames (10 ms) simulated per second wall-clock time (e.g. 1 frame/second means a factor of 100 compared to real time) for a standard test-case including e.g. cell search, BCH/PCCPCH decoding, etc.

It can be seen that just by abstracting the data transfer a speed-up of more than a magnitude can be achieved. The more data transfer happens, the higher the speedup and thus the speedup heavily depends on the application scenario.

VII. Conclusion and Outlook

In this paper we introduce the TLM$^+$ modeling style which enables abstraction techniques which go beyond the current TLM abstraction, thus enabling faster VP simulations for early SW validation in complex SoC platforms. We achieved the higher abstraction by employing three different abstraction techniques. Namely, merging directly at the HW/SW interface moving from bit true to content true data representations, and by a dedicated resource model for obtaining a high degree of timing accuracy when compared to TLM. Experimental results on an industrial mobile phone platform application have also shown that moving to the TLM$^+$ abstraction can yield up to hundred orders of magnitude faster simulations when compared to regular TLM platforms with ISS-core models.

Currently, this work is being applied to further industrial use cases, in order to obtain more experimental results and to analyze the feasibility of the approach in an industrial design environment.

The work presented in this paper is partially funded by the German Federal Ministry of Education and Research within the context of the SANITAS project (01M3088).

References

[1] OSCI, *Standard TLM 2.0.1*, http://www.systemc.org, July 2009.
[2] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina, "Mixed SW/SystemC SoC Emulation Framework," in *proceedings of IEEE International Symposium on Industrial Electronics*, 2007.
[3] S. Sonntag, M. Gries, and C. Sauer, "SystemQ: Bridging the gap between queuing-based performance evaluation and SystemC," *Design Automation for Embedded Systems*, vol. 11, 2007.
[4] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS Modeling for System Level Design," in *proceedings of Design, Automation and Test in Europe Conference*, 2003.
[5] R. Le Moigne, O. Pasquier, and J. P. Calvez, "A Generic RTOS Model for Real-time Systems Simulation with SystemC," in *proceedings of Design, Automation and Test in Europe Conference*, 2004.
[6] H. Yu, A. Gerstlauer, and D. D. Gajski, "RTOS Scheduling in Transaction Level Models," in *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2003.
[7] S. Yoo, G. Nicolescu, L. Gauthier, and A. A. Jerraya, "Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design," in *proceedings of Design, Automation and Test in Europe Conference*, 2002.
[8] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. A. Jerraya, "Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer," in *proc. of Design, Automation and Test in Europe Conference*, 2003.
[9] A. Bouchhima, S. Yoo, and A. A. Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model," in *proceedings of Asia and South Pacific Design Automation Conference*, 2004.
[10] A. Bouchhima, P. Gerin, and F. Pétrot, "Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation," in *Asia and South Pacific Design Automation Conference*, 2009.
[11] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-Performance Timing Simulation of Embedded Software," in *proceedings of Design, Automation and Test in Europe Conference*, 2008.
[12] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel, "Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation," in *proceedings of 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2008.
[13] W. Ecker, S. Heinen, and M. Velten, "Using a Dataflow abstracted Virtual Prototype for HdS-Design," in *proceedings of Asia and South Pacific Design Automation Conference*, 2009.