FAKULTÄT FÜR !INFORMATIK

High-level System Modeling with SystemC and TLM

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Technische Informatik

ausgeführt von

Christian Widtmann
Matrikelnummer 0125145

am:
Institut für Technische Informatik

Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas STEININGER
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Martin DELVAI

Wien, 19. 01. 2009

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

Traditional methodologies increasingly fail to tackle the challenge of contemporary embedded system design. The drive towards shorter product life cycles and time-to-market necessitates an increase of productivity. Electronic System Level Design (ESL) addresses this issue by modeling and abstraction. The Transaction Level Modeling (TLM) standard, based on the system modeling language SystemC, is targeted at the design of fast virtual system prototypes that allow early hardware/software co-development as well as architectural and performance exploration. This thesis presents the methodology of ESL design based on transaction models. It describes the features of SystemC, the concepts of the thereupon defined modeling standard and the tools available to the designer. In the course of a case study the Advanced Encryption Standard (AES) is refined from its mathematical description to an architectural model. The design flow includes a number of modeling steps that illustrate the capabilities of the different modeling styles. Particular focus is laid on linking theoretical concepts to their practical implementation.

The discussion works out the individual attributes that make a certain modeling style useful for a particular use case, as it is perceived during the case study. Performance figures conclude the analysis and illustrate the simulation performance of the various modeling styles.

Zusammenfassung

Der Drang zu kürzeren Produktlebenszyklen und -einführungszeiten sowie die ständig steigende Komplexität von Embedded Systems erfordern einen Anstieg der Produktivität, dem traditionelle Designmethoden immer weniger gewachsen sind. Zur Bewältigung dieses Problems verwendet Electronic System Level Design (ESL) Modellierung und Abstraktion. Der auf der Modellierungssprache SystemC basierende Transaction Level Modeling (TLM) Standard zielt auf die Erstellung virtueller Prototypen, die bereits früh parallele Entwicklung von Hard- und Software sowie Analysen von Architektur und Leistung ermöglichen.

Diese Arbeit präsentiert die Methodologie von auf TLM basierendem ESL Design. Sie beschreibt die Fähigkeiten von SystemC, die Konzepte des darauf aufbauenden Standards sowie die Werkzeuge, die dem Entwickler zur Verfügung stehen. Im Zuge einer Fallstudie wird der Advanced Encryption Standard (AES) von einer mathematischen Beschreibung zu einem Architekturmodell weiterentwickelt. Der Designflow besteht aus einer Reihe von Modellierungsschritten, die die Möglichkeiten der verschiedenen Modellierungsstile aufzeigen sollen. Ein Schwerpunkt dabei liegt auf der Verknüpfung theoretischer Konzepte mit ihrer praktischen Implementierung.

Basierend auf den Ergebnissen der Fallstudie werden jene Eigenschaften der jeweiligen Modellierungsstile herausgearbeitet, auf denen die Eignung für ihre Anwendungsfälle beruht. Abschliessend erfolgt eine Analyse der Simulationsleistung der beschriebenen Modelle.

Danksagung

Diese Arbeit widme ich meinen Eltern, deren unermüdliche Unterstützung mein Studium erst möglich gemacht hat.

Weiters gilt mein Dank . . .

. . . *Univ.Ass. Dipl.-Ing. Dr.techn. Martin Delvai* für seine Betreuung und Anleitung.
. . . *Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger* für Jahre engagierter Lehre.
. . . *Andreas Dielacher* für die produktive und unterhaltsame Studienzeit.

Last but not least danke ich meiner Freundin, *Mag. DI Birgit Schnattinger*, dass sie ihre aus zwei Diplomarbeiten stammenden Erfahrung mit mir geteilt und mich auch sonst in jeder erdenklichen Weise unterstützt hat.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

1.1. Motivation

The design of modern computing systems is evolving into a more and more complex task. The distinction between hardware and software systems has been blurred, and has been replaced by the paradigm of the *Embedded System* which strives for employing the optimal hardware/software tradeoff solution. A good example of a contemporary embedded system is the common cellphone which deploys complex algorithms for a variety of communication tasks, multimedia features and organizer jobs. These capabilities come with respective sets of sometimes contradicting requirements like speed versus power consumption that must eventually be satisfied by the final implementation.

The wide application space is complemented by a vast architectural space that allows an abundance of implementation strategies. The *Computing Universe* by Clive Maxfield and Alvin Brown (Figure 1.1) shows an overview of the various possibilities that range from single processor to multi processor, ASIC and a collection of hybrid approaches. Qualified reasoning about the nature and requirements of the system as whole, rather than as a collection of algorithmic pieces, is a prime prerequisite to devise an optimized architecture among the many possible choices. Traditional design methodologies increasingly fail to handle such reasoning in a cost- and time-effective manner, at that when product time-to-market is the key to success.

Electronic System Level Design lines up to tackle this issue. At the center of its methodology is a hardware/software co-development environment that allows the design of a

Virtual System Prototype (VSP). A VSP is a single or a collection of models that may comprise different levels of abstractions. It describes system behavior as well as architecture, and a variety of use cases can be addressed by the appropriate level of abstraction.

> "ESL methodologies have many starting points, and there is no "right one" for all designs in all design domains." ..."What is important to recognize is that there are "right principles" in an ESL methodology, and these may be implemented in flows incorporating multiple languages and notations." [MBP07]

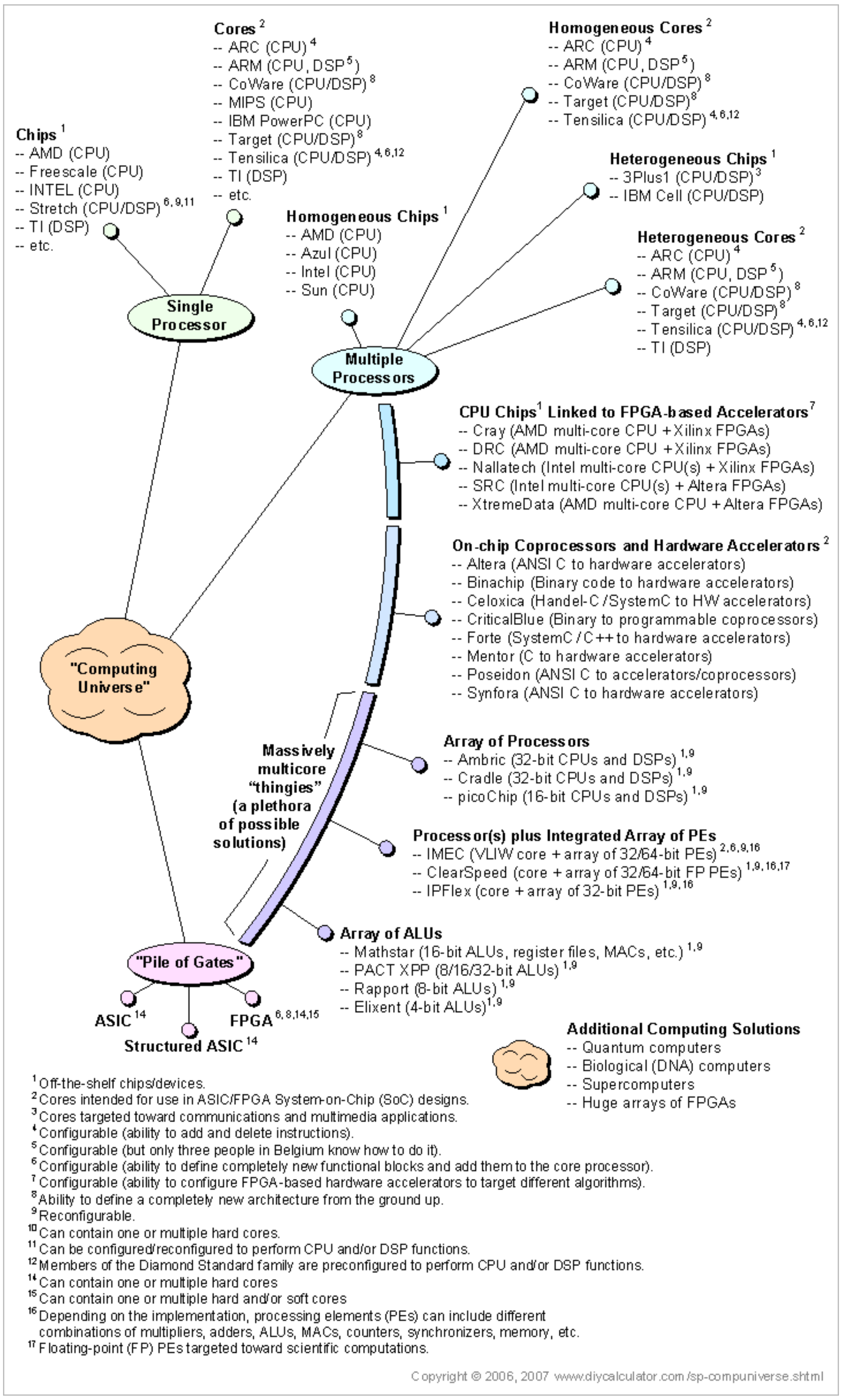
The above definition already indicates that there is a manifold of ESL design flows and tools. While some focus on model-based reasoning about system properties, others concentrate on software development based on realistic models of the underlying hardware. The transition from a traditional (possibly VHDL or otherwise RTL-based) design flow to a more comprehensive ESL design flow initially requires a considerable effort. Adequate tools have to be selected, models have to be written or otherwise acquired, and designers have to adapt to a new style of development. But adopting a consistent and deliberate ESL design flow has many positive effects. It increases quality and reliability of solutions, aids in the production of optimal designs due to the ability to explore and evaluate different strategies, and ultimately reduces time to market primarily by early software development based on Virtual System Prototypes. Therefore this thesis focuses on the understanding and development of models which can be assembled to constitute a Virtual System Prototype in the context of an ESL design flow.

1.2. Goals and organization of this work

The purpose of this thesis is to understand the principles and application of *SystemC* and *Transaction Level Modeling* as tools for *Electronic System Level Design*.

Chapter 2 outlines the fundamental methodology behind ESL and introduces SystemC and the thereupon based TLM standard with the objective of evolving the theoretical rules and definitions into practical approaches that are useful for model design.

The building blocks available for model implementation are described in Chapter 3. It specifies the attributes of individual classes and also illustrates and elaborates con-


Cores [2]
-- ARC (CPU) [4]
-- ARM (CPU, DSP [5])
-- CoWare (CPU/DSP) [8]
-- MIPS (CPU)
-- IBM PowerPC (CPU)
-- Target (CPU/DSP) [8]
-- Tensilica (CPU/DSP) [4,6,12]
-- TI (DSP)
-- etc.
Chips [1]
-- AMD (CPU)
-- Freescale (CPU)
-- INTEL (CPU)
-- Stretch (CPU/DSP) [6,9,11]
-- TI (DSP)
-- etc.
Homogeneous Cores [2]
-- ARC (CPU) [4]
-- ARM (CPU, DSP [5])
-- CoWare (CPU/DSP) [8]
-- Target (CPU/DSP) [8]
-- Tensilica (CPU/DSP) [4,6,12]
Heterogeneous Chips [1]
-- 3Plus1 (CPU/DSP) [3]
-- IBM Cell (CPU/DSP)
Homogeneous Chips [1]
-- AMD (CPU)
-- Azul (CPU)
-- Intel (CPU)
-- Sun (CPU)
Heterogeneous Cores [2]
-- ARC (CPU) [4]
-- ARM (CPU, DSP [5])
-- CoWare (CPU/DSP) [8]
-- Target (CPU/DSP) [8]
-- Tensilica (CPU/DSP) [4,6,12]
-- TI (DSP)
Single Processor
Multiple Processors
CPU Chips [1] Linked to FPGA-based Accelerators [7]
-- Cray (AMD multi-core CPU + Xilinx FPGAs)
-- DRC (AMD multi-core CPU + Xilinx FPGAs)
-- Nallatech (Intel multi-core CPU(s) + Xilinx FPGAs)
-- SRC (Intel multi-core CPU(s) + Altera FPGAs)
-- XtremeData (AMD multi-core CPU + Altera FPGAs)
On-chip Coprocessors and Hardware Accelerators [2]
-- Altera (ANSI C to hardware accelerators)
-- Binachip (Binary code to hardware accelerators)
-- Celoxica (Handel-C / SystemC to HW accelerators)
-- CriticalBlue (Binary to programmable coprocessors)
-- Forte (SystemC / C++ to hardware accelerators)
-- Mentor (C to hardware accelerators)
-- Poseidon (ANSI C to accelerators/coprocessors)
-- Synfora (ANSI C to hardware accelerators)
"Computing Universe"
Massively multicore "thingies" (a plethora of possible solutions)
Array of Processors
-- Ambric (32-bit CPUs and DSPs) [1,9]
-- Cradle (32-bit CPUs and DSPs) [1,9]
-- picoChip (16-bit CPUs and DSPs) [1,9]
Processor(s) plus Integrated Array of PEs
-- IMEC (VLIW core + array of 32/64-bit PEs) [2,6,9,16]
-- ClearSpeed (core + array of 32/64-bit FP PEs) [1,9,16,17]
-- IPFlex (core + array of 32-bit PEs) [1,9,16]
Array of ALUs
-- Mathstar (16-bit ALUs, register files, MACs, etc.) [1,9]
-- PACT XPP (8/16/32-bit ALUs) [1,9]
-- Rapport (8-bit ALUs) [1,9]
-- Elixent (4-bit ALUs) [1,9]
"Pile of Gates"
ASIC [14]
FPGA [6,8,14,15]
Structured ASIC [14]
Additional Computing Solutions
-- Quantum computers
-- Biological (DNA) computers
-- Supercomputers
-- Huge arrays of FPGAs
Copyright © 2006, 2007 www.diycalculator.com/sp-compuniverse.shtml

[1] Off-the-shelf chips/devices.
[2] Cores intended for use in ASIC/FPGA System-on-Chip (SoC) designs.
[3] Cores targeted toward communications and multimedia applications.
[4] Configurable (ability to add and delete instructions).
[5] Configurable (but only three people in Belgium know how to do it).
[6] Configurable (ability to define completely new functional blocks and add them to the core processor).
[7] Configurable (ability to configure FPGA-based hardware accelerators to target different algorithms).
[8] Ability to define a completely new architecture from the ground up.
[9] Reconfigurable.
[10] Can contain one or multiple hard cores.
[11] Can be configured/reconfigured to perform CPU and/or DSP functions.
[12] Members of the Diamond Standard family are preconfigured to perform CPU and/or DSP functions.
[14] Can contain one or multiple hard cores
[15] Can contain one or multiple hard and/or soft cores
[16] Depending on the implementation, processing elements (PEs) can include different combinations of multipliers, adders, ALUs, MACs, counters, synchronizers, memory, etc.
[17] Floating-point (FP) PEs targeted toward scientific computations.

Figure 1.1.: The Computing Universe [MB06]

cepts related to their implementation and usage. It focuses on creating a preliminary conception of how the constructs of SystemC and TLM are practically applicable, in particular in relation to the outlined methods.

Chapter 4 deals with an exemplary modeling case study. It shows the capabilities of the respective modeling styles, their traps and pitfalls and the implications on productivity for both implementation and simulation. One particular focus is placed on a consistent design flow, starting from the algorithmic level and traversing through several modeling stages.

The simulation results of the models designed using different modeling styles are presented in Chapter 5. Additionally, the following questions are answered.

- Which insights can be gained by employing a certain modeling style?
- How well do the TLM use cases and their mapping on modeling styles match with practical experience?
- To what extent can HW/SW partitioning be addressed by TLM modeling and which modeling style is most appropriate?

The thesis concludes with Chapter 6.

Chapter 2.

The methodology of TLM-based ESL design

2.1. Electronic System Level Design

Electronic System Level Design (ESL) is by [MBP07] defined as . . .

> ". . . the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner."

There we also find a definition from Wikipedia dated July 2006, which has since then been modified to . . .

> "Electronic System Level (ESL) design and verification is an emerging electronic design methodology that focuses on the higher abstraction level concerns first and foremost." [Wik08][1]

In principle ESL spans a multitude of abstraction levels, beginning with high-level algorithmic representation of a system and ending with a description that is detailed enough to be a *link to implementation* like RTL. Common to all is the treatment of a whole system using a high-level language such as C++, MATLAB, Petri-Net techniques etc. and a gain of speed and efficiency by abstracting away details that are considered irrelevant to the intended task. Thus, ESL design is often related to creating *models* of the system under consideration.

[1]Last edit at the time of writing: 22 October 2008

> "Basically a model is a simplified abstract view of the complex reality. It may focus on particular views, enforcing the "divide and conquer" principle for a compound problem." [Goo05]

The related activities include high-level behavioral synthesis of whole systems by *Electronic Design Automation (EDA)* tools, elevating their level of applicability above the *Register Transfer Level (RTL)*. Modeling of complex systems at the beginning of the design phase allows architectural exploration, hardware/software tradeoff analysis and software development while the target hardware is not yet available. Most of these tasks are performed manually today. Therefore a main goal that ESL strives to achieve is the creation of a consistent design flow that starts at the algorithmic level and concludes with a link to implementation. Due to the wide range of tasks involved ESL employs complementary methods, each suited to specific subtask(s). One possibility through which these methods can be connected to each other is the use of a *Virtual System Prototype (VSP)* as central piece of the methodology. A cross-section of the commercial ecosystem surrounding ESL can be found in [MBP07].

In particular, ESL design can be conducted using the language[2] *SystemC* as an abstract system modeling language. Its intended application fields span from high-level modeling to behavioral synthesis and RTL, which makes it appropriate for the generation of models that can serve as VSP. To aid this process and to avoid uncontrolled growth of incompatible IP interface and modeling styles, the *Open SystemC Initiative (OSCI)* has complemented SystemC with the *Transaction Level Modeling Standard (TLM)* (Figure 2.1). It uses function calls, rather than signals or wires, for inter-module communication. Its goal is to create a common standard for the generation of high-level models and to leverage the full potential of the unique features offered by SystemC. This thesis employs Version 2.2.0 of the SystemC language [IEE03] and Version 2.0 draft 2 of the OSCI TLM standard [Ope07b].

[2]Strictly speaking SystemC is not a language of its own but rather a library. Nevertheless, due to the specific capabilities that it introduces, we will refer to it as language in this work.

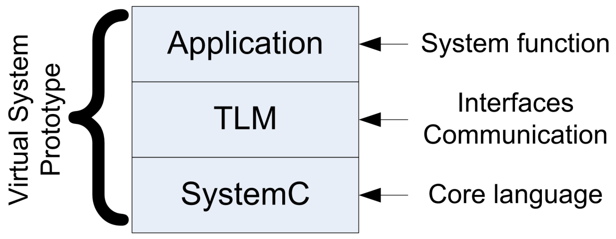


Figure 2.1.: Virtual System Prototype with SystemC and TLM

2.2. SystemC

The *Open SystemC Initiative*, founded 1999 by *CoWare*[3]and *Synopsys*[4], is a nonprofit organization working on the definition of an industry-standard system-level hardware modeling language. SystemC was approved as IEEE standard 1666 in 2005. Version 1.0 resembles HDLs like VHDL or Verilog and provides features like concurrency, bit-accuracy and timing. Being based on C++, it is free of many limitations that are imposed by strongly-typed language such as VHDL. Version 2.0 and above focus on more abstract concepts like *interfaces* and *channels*, and emphasize the modeling aspects of SystemC.

While similarly to VHDL only a certain subset of SystemC is synthesizable [Ope04], it leverages the strengths and flexibility of C++ to enable more efficient modeling and design before going into synthesis or implementation. These strengths include the *Standard Template Library*, *Object Orientation* and *Templates*. Figure 2.2 illustrates the overlaps of SystemC with other languages as well as the additional capabilities. SystemC is rather a system modeling language but still includes many features that are offered by a HDL and is also applicable in verification scenarios. The sum of these qualities makes it well suited for ESL and TLM tasks.

Executable models are created by compiling the model code into a binary which already includes the OSCI reference simulator kernel and thus eliminates the need for an additional simulation tool. The compilation can be conducted by any regular C++

[3]http://www.coware.com/
[4]http://www.synopsys.com/

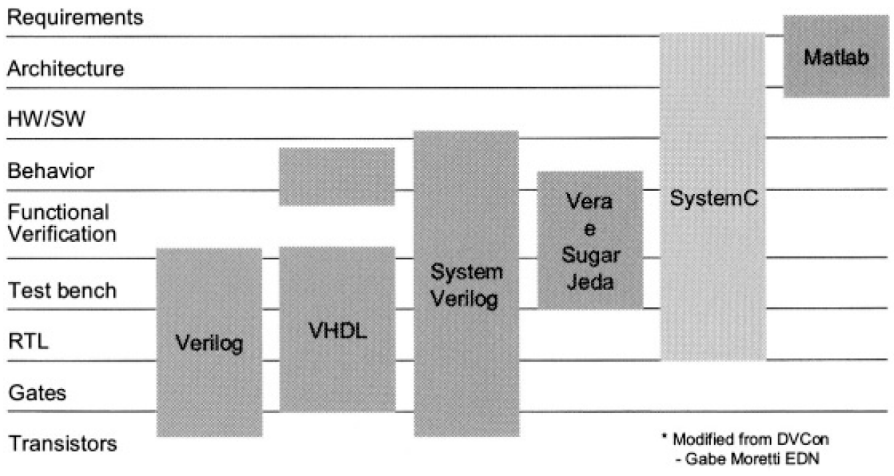


Figure 2.2.: Comparison of SystemC against other languages [BD05]

compiler, making it possible to combine normal C++ with SystemC code. Output and debugging can be performed by simple shell output or by writing out waveform data for later review by an external viewer. The free OSCI reference simulator does not include any IDE or IP library which is a potential motivation to migrate to one of the many commercially available tools. This thesis uses the reference simulator for a focus on SystemC and TLM without bias by a commercial tool or IDE.

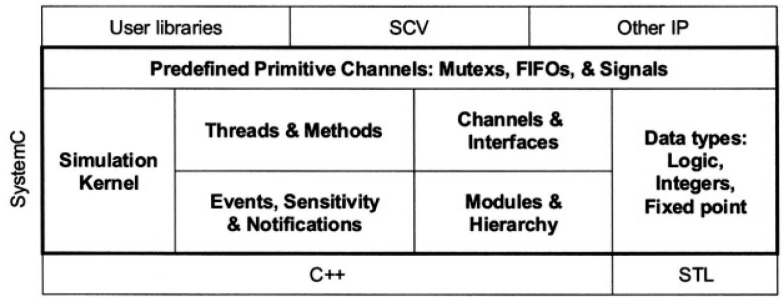

User libraries
SCV
Other IP
Predefined Primitive Channels: Mutexs, FIFOs, & Signals
SystemC
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types: Logic, Integers, Fixed point
Events, Sensitivity & Notifications
Modules & Hierarchy
C++
STL

Figure 2.3.: Architecture of the SystemC language [Ope07b]

Figure 2.3 depicts the building blocks of the SystemC language that we will now contrast to VHDL, with which the reader is assumed to be familiar to some degree. The common features root in the area of hardware design and related simulation semantics, enabling SystemC and VHDL to be used together in mixed-language simulators such as *Mentor Graphics Modelsim*[5].

[5]http://www.mentor.com/ | http://www.model.com/

2.2.1. Processes and sensitivity

The basic unit of simulation behavior is, in VHDL and SystemC alike, the execution thread. While in VHDL all these threads are subsumed under the term *process*, SystemC distinguishes between three types. A *thread* is allowed to call *wait* for an event or time during its execution and thus suspend it, while a *method* is not allowed to do this and is always executed from its beginning to its end. A *cthread* is a special type of thread that must be sensitive to a *clock signal* and is currently relevant for synthesis rather than for simulation [BD05]. A clock signal in turn is a special type of signal defined by the SystemC language.

In additional to explicit waits, SystemC and VHDL support *static sensitivity*. Along with the declaration of a process, a list of events to which it shall be sensitive is enumerated. The firing of such an event will trigger the execution of all processes that are sensitive to it. A method will be executed in its entirety, while a thread is possibly only resumed at the point where it has been suspended. SystemC additionally supports *dynamic sensitivity* for both threads and methods, enabling them to change their sensitivity list during simulation. Also, while in VHDL all processes are executed once at simulation start, this can be prevented in SystemC by calling *dont_initialize()* right after process declaration, which is especially useful with methods.

2.2.2. Interfaces, (ex)ports and signals

Ports and signals in VHDL and SystemC are based upon the *evaluate-update paradigm*. It means that a process will, after having written a new value to a signal, still read the old value from this signal as long as it has not *yielded* (i.e. handed simulation control back to the scheduler), in contrast to a variable that instantly reflects the new value.

An *interface* is an abstract class that contains no data members but only purely virtual functions which need to be implemented by another class inheriting from the interface [Wil06]. Using the intuition of a signal and the definition of an interface, we will now define the term *channel*, which can comprise communication mechanisms of varying complexity, from single wires to whole bus protocols. Channels are a basic mean of communication between two processes at the same or different levels of hierarchy.

> "A SystemC channel is a class that implements one or more SystemC interface classes and inherits from either sc_channel or sc_prim_channel[6]. A channel implements all the methods of the inherited interface classes. A SystemC interface is an abstract class that inherits from sc_interface and provides only pure virtual declarations of methods referenced by SystemC channels and ports. No implementations or data are provided in a SystemC interface." [BD05]

A *signal* in SystemC is a class that inherits from and implements an interface that offers similar capabilities like those of a signal known from VHDL (in particular the evaluate-update paradigm). It is therefore also a channel. Signal implementations are supplied for SystemC data types, bit-accurate data types (comparable to those in VHDL) as well as native C++ data types. Any user-defined class can be used as signal if the class implements the functions required by the signal interface, which offers increased flexibility compared to VHDL.

A *port* is a pointer to a signal (or channel) outside the current module that enables the processes inside this module to talk to processes of another module that is again connected to that external signal by one of its ports.

An *export* allows a module to provide an interface to its parent module. It forwards interface method calls to the channel to which it is bound. An export defines a set of services (as identified by the type/interface of the export) that are provided by the module containing the export [IEE03], i.e. an export is used for advertising internal interfaces for access from outside. They are available in SystemC since Version 2.1.

While ports declare interfaces that are required at the module boundary (and thus need to be bound), exports declare interfaces provided at a module boundary and may remain unbound [Sys04].

Figure 2.4 illustrates how port and export can be used together. It shows a part of the example design *lt_min_system* which is included with the TLM standard download. The *traffic_generator* module writes data to the *lt_initiator* module by calling the functions of the associated port. The available functions are determined by the type of the port, in this case it is the interface class *rw_if* which offers simple read and write functions. They are implemented in the lt_initiator and made available for external binding through an

[6]Inheriting from sc_channel or sc_prim_channel is a structural requirement similar to the fact that a SystemC module needs to inherit from sc_module.

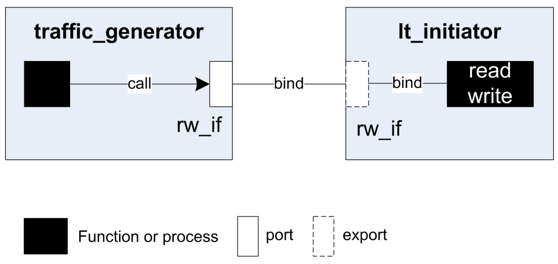


Figure 2.4.: Port / export example

export of the same type rw_if. This export is then bound to the port of same type, so the port effectively calls the function implementation to which the export is bound.

2.2.3. Events

One feature specific to SystemC is the possibility of using atomic events. These events constitute an entity of their own, while in VHDL, the smallest such unit is a whole signal which already includes a variety of associated events. Often these events are an unnecessary surplus, for instance when only simple synchronization is required which in principle needs only a single event.

A SystemC event knows three types of notifications. The first one is the *instant notification*. It triggers all processes that listen to the event as soon as the process that notified the event yields (and therefore still in the same delta cycle[7]). The second, *delayed notification*, puts the notification off by a delta delay[8]. Therefore the trigger becomes effective only when all processes that can run in the current delta cycle have yielded and the simulator has advanced to the next delta cycle, but has not yet advanced simulation time. And third, a notification annotated by a span of simulation time greater than zero delays the notification until the annotated amount of simulation time has passed.

Single events are important for efficient synchronization in models that reside on a higher level than (synthesizable) RTL, and are one of the main reasons for the potential increase in simulation performance of abstract SystemC models (see Section 5.5).

[7]A simulation cycle that is performed at the same simulation time as the previous one is called a delta cycle.[Ele00]

[8]A delta delay is an infinitesimally small delay that separates events occurring in successive simulation cycles but at the same simulation time.[Ele00]

2.2.4. Implications on simulation performance

Due to the progress in modern compiler technology, SystemC is not by itself faster than VHDL [9]. If the simulation takes place at RTL level or the simulated code is even synthesizable, the simulation performance of SystemC is comparable to that of other HDLs [BD05, RDL05]. The picture is different for simulation at a higher level of abstraction. The features of SystemC can result in increased simulation performance if the following rules are considered . . .

- Keep the number of events and context switches to a minimum by (un)timed techniques. Use single events rather than signals for synchronization.
- Choose the right data type for the required task as well as for the used machine. Native C++ data types simulate faster than SystemC or VHDL data types.
- Take advantage of the STL and other well-defined libraries.
- Use pointers instead of copying data.
- Use ports with exports to avoid reliance on explicit multilevel paths and to permit direct function call interfaces for TLM without introducing additional context switches (in contrast to port-to-port mappings). [Sys04]

The next section describes how the features available through the SystemC language can be implemented in a modeling standard that increases simulation performance by embedding them in appropriately defined levels of abstraction and modeling styles.

2.3. Transaction Level Modeling

The following definition of *Transaction Level Modeling (TLM)* in the context of SystemC is given in [Gro02]. It not only states the abstract definition but also illustrates the design paradigms by which TLM can be realized in practice and the positive consequences.

[9]Synthesis for instance, regardless of the language, usually requires a certain level of detail, a certain usage and number of events and processes, certain data types.

"TLM is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. Communication mechanisms such as busses or FIFOs are modeled as channels, and are presented to modules using SystemC interface classes. Transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange.

At the transaction level, the emphasis is more on the functionality of the data transfers - what data are transferred to and from what locations - and less on their actual implementation, that is, on the actual protocol used for data transfer. This approach makes it easier for the system-level designer to experiment, for example, with different bus architectures (all supporting a common abstract interface) without having to recode models that interact with any of the buses, provided these models interact with the bus though the common interface."

2.3.1. TLM with SystemC

The TLM 1.0 standard (introduced in 2005) defines a set of APIs for transaction-level communication, but does not define the content of the transactions or how they should be managed within a system. The OSCI tries to address this yet unstructured area by Version 2 of the standard, to enable consistent interaction between tools and thus consistent usage of these tools.

In the context of Figure 2.3, TLM constitutes a user library based on the classes offered by SystemC. A key concept is the declaration of interface classes (as defined in Section 2.2.2) and calling the functions of these interfaces as mean of communication in contrast to long-winded synchronization procedures including maybe several signals or events. These are the *Core TLM2 interfaces* depicted in Figure 2.5 and described in Section 3.2.

The TLM library is relying on templates that may in principle assume any user-defined data types. The standardization of version 2 was targeted at the context of a *memory-mapped bus (MMB)* and the default template data types and their values have been defined with this context in mind. In particular, this includes the *generic payload (GP)* as basic transaction class transported by interface calls. Only a reference to the transaction, which in turn contains only a pointer to the associated data buffer, is passed. This alleviates the need for extensive copying and increases simulation performance significantly. The *phase* of the transaction represents the current state of the protocol state machine used for communication. It enables the notion of *timing points* that mark the transition to the next phase. The different use and availability of phases are important for the definition of a *modeling style*.[10]

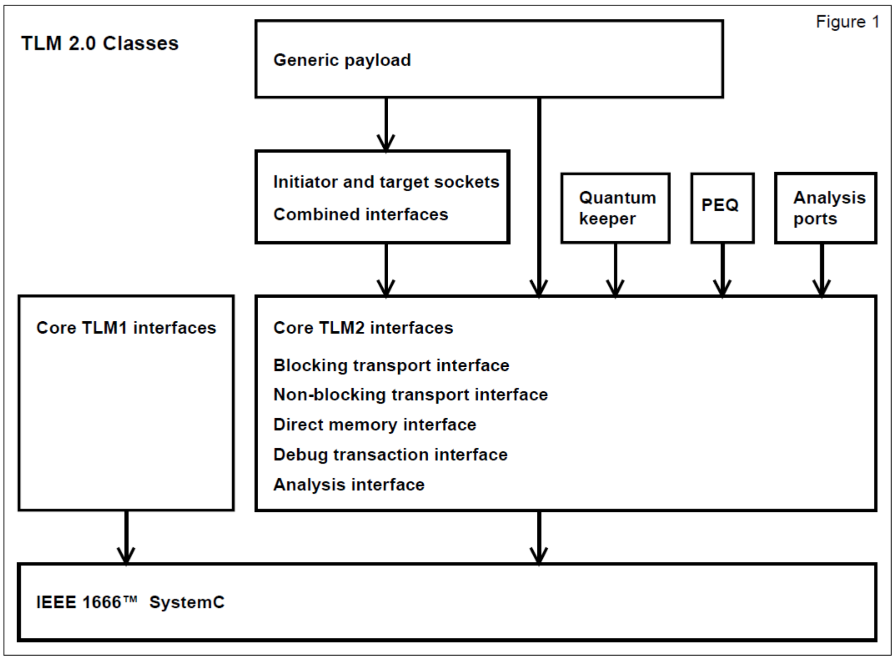


Figure 2.5.: Cornerstones of the TLM standard [Ope07b]

2.3.2. Modeling styles

The following definition of the term *coding style* can be found in [Ope07b].

> "A coding style is a set of programming language idioms that work well together, not a specific abstraction level or software programming interface.

[10]In the context of the MMB communication protocol, synchronization phases are tailored to simple read and write requests and responses. For modeling a more complex protocol state machine like e.g. packet based network protocols, different type definitions might be required. Since the TLM standard defines its modeling styles based on the (theoretically replaceable) MMB data types, they would need to be re-defined as well, if these data types were altered.

TLM2 recognizes several coding styles, which should be used as a guide to model writing."

The same document uses the term *modeling style* in an inconsistent but semantically equivalent fashion. This thesis therefore consistently uses the term *modeling style* to refer to both modeling and coding style in the sense of [Ope07b].

A modeling style is represented by a set of interfaces together with an (optional set) of classes. It defines usage and behavior guidelines that enable the model designer to create interoperable models with certain features, as offered by the modeling style. The most important feature is the capability to model time (in particular delays between phase transitions) more or less differentiated, if at all. The tradeoff is usually the one between (simulation or design) speed and accuracy.

Several use cases that shall be tackled by TLM have been defined, each of which implies the required data and timing accuracy. In order to provide an efficient modeling framework for the individual use cases, TLM provides three modeling styles named *Untimed (UT)*, *Loosely-timed (LT)* and *Approximately-timed (AT)*. The combination with additional template classes results in the features and accuracy that is required for an intended use case.

Figure 2.6 illustrates the usability of certain modeling styles for individual use cases and their associated interfaces and classes. For a more detailed discussion of these classes and socket-related communication see Chapter 3.

Untimed (UT) modeling style

An untimed model may be viewed as a functional model that can minimally consist of just one single execution thread. By definition, an untimed model has no notion of time, just like the interfaces defined to form this modeling style (although syntactically they could contain wait statements for both time and events).

A single-thread model is sometimes referred to as *algorithmic* model rather than *transaction* model, since such a model would require communication between several concurrent execution threads. If several execution threads are employed in an untimed model, they synchronize explicitly and hard-coded by waiting on explicit events rather than time delays. Such a model can be executed without any advance of simulation time. This style is useful for evaluating the algorithmic description of an application,

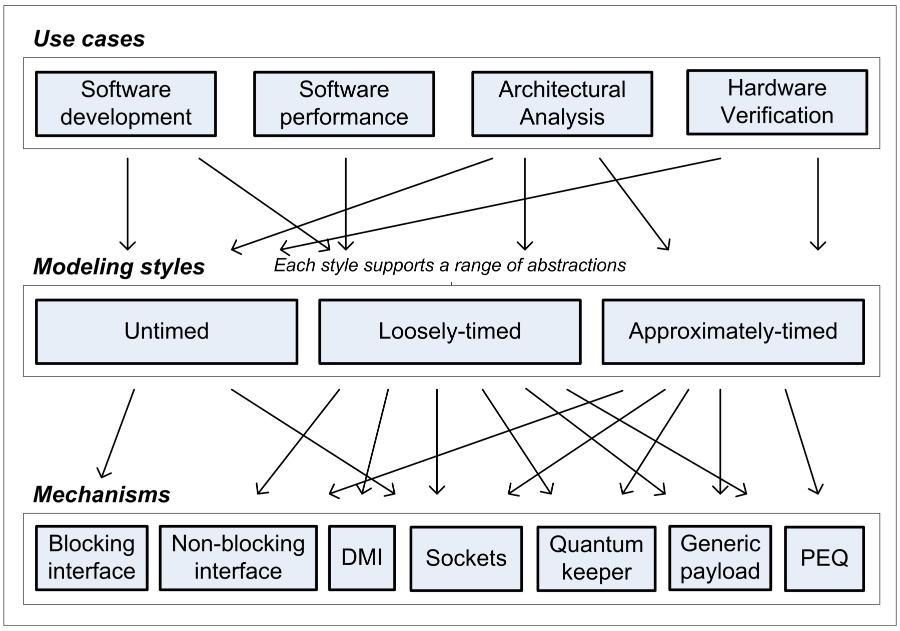


Figure 2.6.: TLM use cases (extended from [Ope07b])

its partitioning into blocks and the resulting communication and data dependencies between the blocks, while yet omitting concurrency and structured communication.

As Figure 2.6 depicts, this makes untimed modeling style interesting for software development on a rapid untimed model of the underlying hardware that requires only functional correctness, architectural analysis of the application and in relation to this, for hardware verification. Performance analyses are not possible due to the missing notion of time.

Loosely-timed (LT) modeling style

The loosely-timed modeling style uses a set of interfaces that allows the annotation of delays and the structuring of communication into two phases, *begin request* and *begin response*. Requests and responses are not attributed with a duration, but the transaction ends with the begin response phase. However the delay between request and response can be specified. A loosely-timed model is aware of time, in particular of a global simulation time as delivered by the *sc_time_stamp* call, and a time that is local to the respective module or processing thread which may be ahead of the global simulation time. It does not necessarily rely on an advance of global simulation time to produce a reply.

Temporal decoupling (TD) is an optional feature of the loosely-timed modeling style. It means that an execution thread runs ahead of the current global simulation time, keeping track of the local time offset, until it has exceeded its local *time quantum*. This quantum denotes the span of time that a thread is allowed to run uninterrupted. A temporally decoupled thread may run beyond a point where a non-decoupled thread would have already yielded[11]. The decoupled thread eventually yields and permits another process to run for its respective quantum. The global simulation time is only advanced when all threads that are currently scheduled to run have used up their quantum, and the simulation time is then advanced by this quantum (or less if some threads were using a local quantum that is shorter than the global quantum). This scenario requires that no thread needs any explicit synchronization whatsoever, possibly at the cost of simulation accuracy, but at the largest increase in simulation performance that can be gained by temporal decoupling due to the low number of context switches. Since it is an attribute of a thread it is not required that all threads of a model are temporally decoupled, although this is recommended by the standard in [Ope07b].

In order to use temporal decoupling without compromising simulation accuracy, threads must be able to *synchronize on demand* if they encounter an unresolved data dependency that does not permit them to run ahead safely until the end of their quantum.

In a loosely-timed model without temporal decoupling, delay information and functional result of a request can be known instantly and annotated to the transaction right way (without synchronization). If the target is not yet able to construct the response, it can indicate to the initiator to wait for a later callback because the target requires an advance of simulation time. In both cases the delay is, in contrast to temporal decoupling, implemented as soon as the transaction has been completed.

So while interfaces and phases stay the same, there are several ways in which the loosely-timed modeling style can become manifest. Common to all of them is that a model should in the best case not require to yield or advance global simulation time to serve a request. Such fast models are well suited for software development on virtual prototypes of the underlaying hardware platform. The loosely-timed coding style supports the modeling of timers and course-grained process scheduling, sufficient to model the booting and running of an operating system [Ope07b]. Under certain conditions it can also be used as modeling style for a coarse architectural and hardware/-

[11]Such points are for instance the resolution of a data dependency or the implementation of a delay as required by a modeling style not using temporal decoupling.

software tradeoff analysis, as described in Section 4.5.5, and is also useful for the design process in evaluating concurrency schemes and delay distributions. Concurrency schemes themselves cannot be derived by TLM or SystemC but require other forms of analyses. Furthermore the loosely-timed modeling style can aide the implementation process by fast models which can by a few function calls model delays at their interfaces that have been derived by analyzing the internal workings of more detailed models, for instance, an already available IP core or real hardware.

Approximately-timed (AT) modeling style

The approximately-timed modeling style syntactically uses the same interfaces as the loosely-timed modeling style, but provides additional phases for the model designer to structure the life-cycle of a transaction. Requests and responses consist of a begin and an end event. This enables finer modeling of timing and is a prerequisite for the realistic modeling of pipelined hardware. It makes the approximately-timed style useful for contention / arbitration modeling, architectural exploration and performance analysis.

Annotated delays are implemented by calls to the scheduler (waits or delayed notification of events). The modules do not make use of precognition as in the loosely-timed temporally decoupled case. The syntactically equal interfaces permit interoperability with loosely-timed models, possibly requiring an adapter module to compensate for the two additional phases.

The availability of four phases allows the definition of the following delays . . . [KHA05]

Accept delay Annotates the minimal interval between two subsequent request begins or response begins. In essence the accept-delay constraints the bandwidth of a block. During this period a slave module is busy with the processing of a request or a master module is busy with the processing of a response.

Response delay Annotates the interval between a request begin and a response begin i.e. the module latency.

In general it is sufficient to model these two delays to implement an approximately-timed module. By simply changing the accept and response delays, it is possible to switch between hardware- and software-style behavior in a flexible way. It is stated in

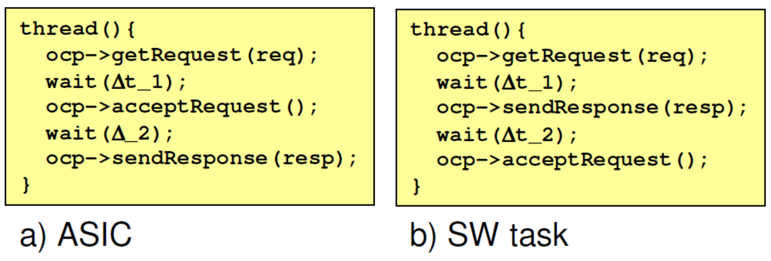
[KHA05] that for a pipelined ASIC, the request accept delay is supposed to be smaller than the response delay, while for software it is the other way round. Figure 2.7 illustrates this classification of behavior based on accept and response delay as done in [KHA05]. The circumstance that this aspect is not treated thoroughly enough in the TLM standard Version 2 draft 2, and that the definition in Figure 2.7 uses a different syntax and is not consistent to the OSCI TLM standard, gives rise to the problem addressed in the following section.

2.3.3. Classification of behavior by phase transitions

The *Open Core Protocol International Partnership (OCP-IP)* cooperates with the OSCI in its effort for industry-wide modeling standards. It focuses on the decoupling of IP computation behavior from communication behavior to promote IP design and reuse in a plug-and-play fashion. In [KHA05] the OCP-IP published concepts that, while embracing the same principles like OSCI TLM, take a more practical approach in their definitions. OCP-IP outlines an UT-like *Functional View* advertised for algorithm/specification, a LT-like *Programmers View* advertised for SW development and an AT-like *Architects View* advertised for hardware/software tradeoff analysis and architectural (platform) modeling.

Figure 2.7 summarizes the definition given in [KHA05]. It should be noted that, while both sources discuss the same TLM semantic, OCP-IP does not use the same syntactical elements as the OSCI. The *getRequest* function call corresponds to the transmission of a request begin, calling *acceptRequest* corresponds to the phase transition to request end and *sendResponds* amounts to a response begin and end in OSCI terms. In the ASIC case, Δt_1 matches the accept delay while in the software case it represents the response delay. In the first case the sum of Δt_1 and Δt_2 constitutes the response delay and in the second case the accept delay.

The OSCI TLM standard [Ope07b] states that the phase transition to response begin implicitly includes a request end. Although the rule was in the first place meant to enable interfacing of loosely-timed initiators with approximately-timed targets, it is nonetheless valid regardless of the context. This however poses a contradiction to the event order depicted in Figure 2.7, where in the case of software a request is accepted only after the response end.



```
thread(){
  ocp->getRequest(req);
  wait(Δt_1);
  ocp->acceptRequest();
  wait(Δ_2);
  ocp->sendResponse(resp);
}
```

a) ASIC

```
thread(){
  ocp->getRequest(req);
  wait(Δt_1);
  ocp->sendResponse(resp);
  wait(Δt_2);
  ocp->acceptRequest();
}
```

b) SW task

Figure 2.7.: Hardware and software behavior as defined by OCP-IP [KHA05]

Although OCP-IP and OSCI rules are thus incompatible for direct interfacing, the OCP-IP approach for defining approximately-timed simulation behavior was adopted due to its intuitiveness. For instance, consider the case of a software function. It delivers its full response before a second call to it is possible (due to the sequential nature of software), which is modeled by the late request acceptance. Therefore a reformulation of OCP-IP function calls into OSCI phase transitions that matches this behavior and preserves the OSCI rules was required, and is depicted in Figure 2.8, 2.9 and 2.10. In addition to software and (pipelined) hardware, the category of non-pipelined hardware is defined.

Software Request accept delay > response delay. It is assumed that an initiator first needs to process a response before it can issue a new request. Communication overhead for software can be neglected because context switches etc. can be modeled by appropriate delay values in computation modules.

Non-pipelined hardware Request accept delay = response delay. It is assumed that the initiator response accept delay is smaller than the target response delay i.e. the initiator can process a response before the target can send the next one. In practice the target can potentially communicate while doing the next workload. If this is not the case, non-pipelined hardware is effectively like software. This depends on the actual implementation of the hardware model. Communication delays are introduced by separate modules.

Pipelined hardware Request accept delay < response delay. The target request accept delay resembles the delay of the slowest pipeline stage (not necessarily the first), the response delay models the latency. Obviously, pipelined hardware can communicate while processing.

Consequently, modeling the difference between hardware and software is effectively

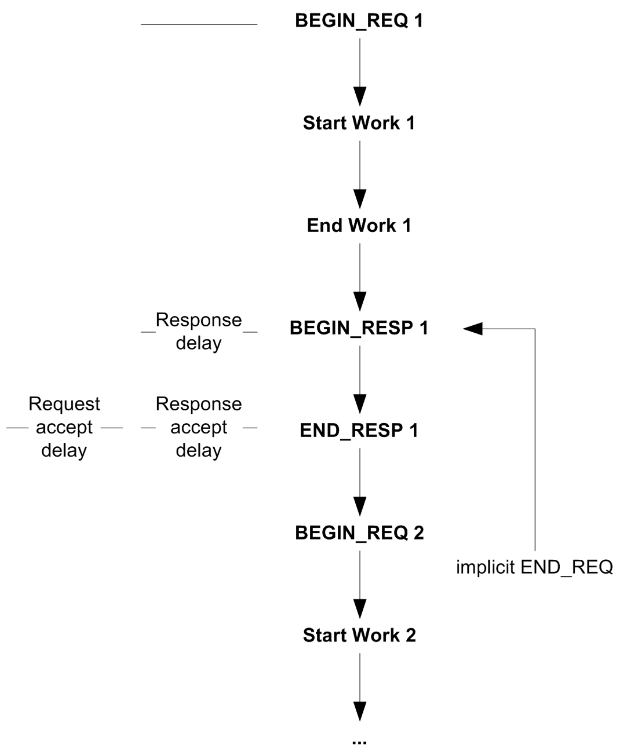


Figure 2.8.: OSCI phase transitions for software behavior

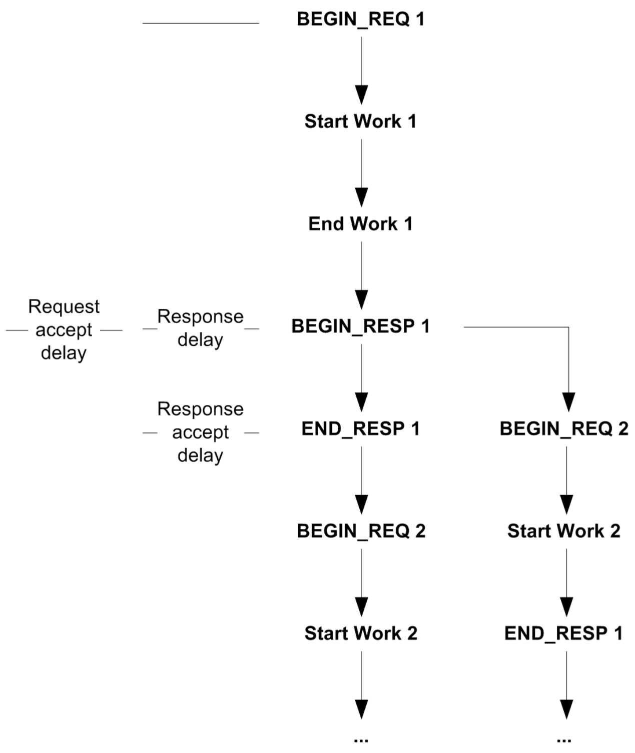


Figure 2.9.: OSCI phase transitions for non-pipelined hardware behavior

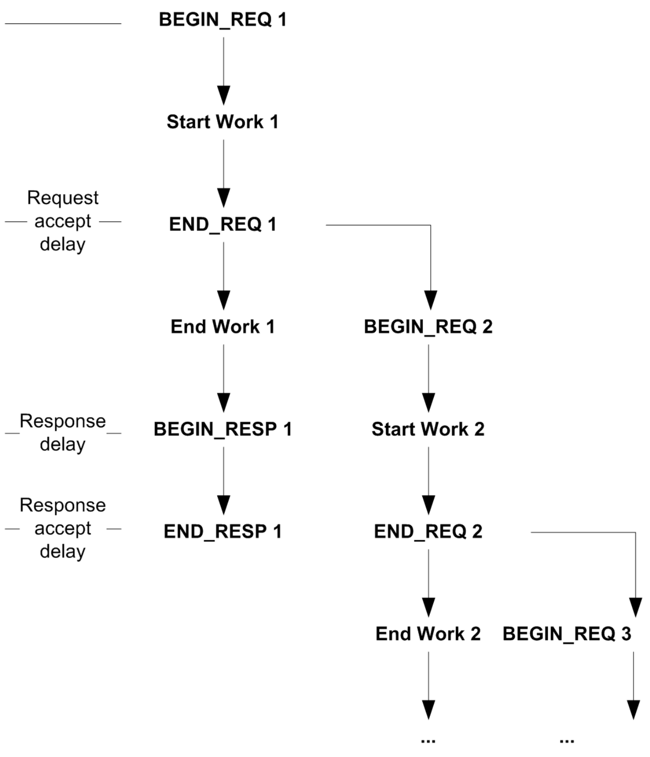


Figure 2.10.: OSCI phase transitions for pipelined hardware behavior

only possible with the additional delays available through the approximately-timed style. In practice, the difference between hardware and software shows in the length of individual delays and the causality of event notifications (concurrent vs. sequential).

The authors of [KHA05] discourage shared ownership of communication and computation delay parameters by the same module and advocates dedicated communication nodes for the modeling of communication delays. In particular, it is stated that the accept delay (i.e. the minimum delay between two successive request begins) of a block should only be related to its computation and should not include communication delays. Section 5.2 discusses the necessary constrains on loosely-timed modeling that allow coarse hardware/software tradeoff analysis without modeling communication delays by separate modules.

Chapter 3.

Transaction Level Model development

The concept of modularity is not only visible in the implementation of a model, but also in the structure of the TLM standard itself. As Figure 2.6 shows, the starting point is one of several use cases that represent the context that a model is targeted at (e.g. software development, hardware verification etc.). A use case requires certain features and thus an appropriate modeling style. The features of a modeling style depend on the inclusion of certain classes supplied by the standard and the way that these classes are employed to implement a model, following the rules that apply to the respective modeling style.

The knowledge of these classes and the illustration of their usage is the focus of this chapter. We cover only those classes that are important for the understanding of what the TLM standard can do, and only in such depth that the reader becomes familiar with the concept. For a more in-depth coverage of the code, see [Ope07b].

3.1. Basic modeling classes

The classes described in this section are, in contrast to the interfaces described in Section 3.2, fully implemented in the TLM standard and can be used out of the box.

3.1.1. Generic payload (GP)

The *generic payload* class has been designed as standard transaction for *memory-mapped buses (MMB)* and represents the data structure towards which the development of the current TLM standard was primarily geared to. Assuming the context of a MMB, it comprises attributes such as a pointer to an associated data buffer, a length value for the data buffer, an address that is the start of the target memory area in focus and a command attribute stating the purpose of the transaction (read or write). Success or the type of error is indicated by the setting of status members. Apart from these basic features, more elaborate ones like byte enables, data stream attributes and user-customized extensions are supported. For our modeling purposes, the basic features mentioned so far suffice.

Sockets are an important aspect of the OSCI TLM communication model. It is usually through a socket that a module is able to communicate with other modules. Sockets encapsulate and implement the interfaces that are discussed in Section 3.2 and 3.3. They can be *initiators* or *targets*. An initiator issues a request (in form of a transaction reference), while a target accepts and processes the request.

The data buffer can assume any user-defined type or class. To set the transaction data pointer to the user data buffer, it needs to be cast to a character pointer (char*) on the initiator side (this is required by the TLM standard transport function declaration), and cast back to the user data type at the target side. The GP only provides the char pointer, but imposes no other restrictions on the transported data structure in addition to the need to cast.

3.1.2. Payload event queue (PEQ)

The PEQ is used for timed modeling and represents a type of FIFO. Unlike a regular FIFO that sorts its content by the input order, the PEQ sorts by the time value that is put into the queue along with each item. It is is not restricted to generic payload transactions but can hold any data type.

The PEQ's output event is notified once for each item that is inserted. Thus, waiting for this event yields a temporal ordering. This work uses the class *MyPEQ*, a redesign of the standard class *tlm_peq* included with the standard's code examples. It was chosen because the documentation indicated that MyPEQ would most likely replace tlm_peq in the next standard version [Ope07b].

```
// MyPEQ put function
void notify(transaction_type& trans, sc_core::sc_time& t)
{
    mScheduledEvents.insert( std::make_pair(t + sc_core::sc_time_stamp(),
         &trans) );
    mEvent.notify(t);
}
```

3.1.3. Quantum keeper (QK)

The *quantum keeper* is a class that is used solely in temporally decoupled models. Although it has not been employed in this work, it is mentioned for completeness.

When individual threads are allowed to run ahead of the local simulation time (i.e. not releasing control of the simulation and continuing in a local time warp), the QK can be used to manage the maximum quantum of simulation time that the current thread may run ahead. Before initiating a new action, the current thread should check with the QK to see if it may still continue to execute, or if the quantum has been exceeded. Decoupled threads that never encounter an unresolvable data dependency (i.e. whose temporally decoupled request can always be answered) are thus kept from running forever.

The QK provides all necessary functions to manage the quantum for each thread, and is an important tool to leverage the increased simulation performance of the LT modeling style in combination with temporal decoupling.

3.2. Core TLM2 interfaces

The interfaces described in this section are abstract classes (as in Section 2.2.2) defined by the TLM standard and have to be implemented in order to be usable. The implementation can be trimmed to fit the model's and application's needs, but has to adhere the rules defined by the standard. Many data types related to communication and user payload are templates and can be replaced by the user to fit his needs. The enumeration data types described here (phase values, return values etc.) are the ones declared by the TLM standard for a *memory-mapped bus* based on *generic payload* transactions.

Two types of interfaces have been identified. On the one hand those that are intended to model the operational behavior of a system and its application, i.e. the behavior that the system exhibits while performing its intended function. They are thus named *operational* interfaces. On the other hand the standard provides interfaces that are meant for meta-functions like simulation setup and analysis. We will call those *supplementary* interfaces.

All interfaces are unidirectional. The target only reacts to the initiator's requests, similar to a master/slave relation. If a target is required to autonomously issue own requests to a module which acts as its initiator, it needs an additional pair of interface sockets. This effectively turns the former target into the initiator and vice versa, for that additional pair of sockets.

3.2.1. Forward and backward path

Despite their names, forward and backward path constitute one interface that goes from one initiator to one target. The forward path describes the chain of calls that starts at the initiator and ends at the target that is meant to receive a transaction. One way to communicate the reaction of the target back to the initiator is through the return value of the function call, after possibly altering the information associated with the transaction (e.g. phase or delay). If the target is not able to supply such a reaction right away, it can, by its return value, indicate to the initiator that it has been reached by the request. It wold then reply at a later time by calling the backward path, which goes through the same modules and sockets as the forward path, but in reverse order.

For some interfaces the backward path is not a mean for reply to a request, but offers an independent function, like the DMI pointer invalidation described in Section 3.2.2. It depends on the interface's intention if there is a backward path and if its interfaces offer complementary or independent functions in relation to those of the corresponding forward path.

3.2.2. Operational interfaces

Blocking transport interface

The blocking transport interface is the simpler of the two transport interfaces. It does not include a backward path like the non-blocking transport interface, but upon return, the transaction must be processed. There is no structuring of communication into phases and no delay annotation. In contrast to the non-blocking version, the blocking interface allows calls to *wait* within the transport function. Consequently it should be called from a SystemC thread rather than a method, since it is usually unknown to the initiator whether a wait will occur within the target's transport function implementation. Since a later response can thus be implemented by calling wait in the forward path implementation, there is no need for a backward path.

This interface is characteristic for the untimed modeling style, and a good initial step towards the implementation of an algorithm into a model, because it yet hides the additional complexity involved with the use of structured communication and time. In code, this is expressed by the fact that the blocking transport call takes a transaction reference as only function argument and has no return value (*void*).

```
// Blocking transport forward function
template <typename TRANS = tlm_generic_payload>
virtual void b_transport(TRANS& trans) = 0;
```

Passing the transaction object by reference implies that there is only one transaction object and all operations are carried out on that very instance. A transaction includes a pointer to the data payload that it encapsulates. All data copy operations are performed by dereferencing this pointer rather than by real copy operations. The same is valid for the non-blocking transport interface described in the next section.

The actual transaction type can be user-defined and is not restricted to the default generic payload. For more details, see [Ope07b]. The models created in the course of this work employ the default templates.

Non-Blocking transport interface

The non-blocking transport interface introduces the notion of time, begin and end phases of requests and responses, and enables more realistic modeling of communication and intertwined computation. The fact that it is forbidden to call wait during the execution of such a transport call requires that the non-blocking interface defines a backward path for cases when the target needs to yield control to process the request (and thus needs to call back at a later time). As a benefit, a non-blocking interface can also be called from a SystemC method, not only from a thread. It is used in the more difficile loosely- and approximately-timed modeling styles.

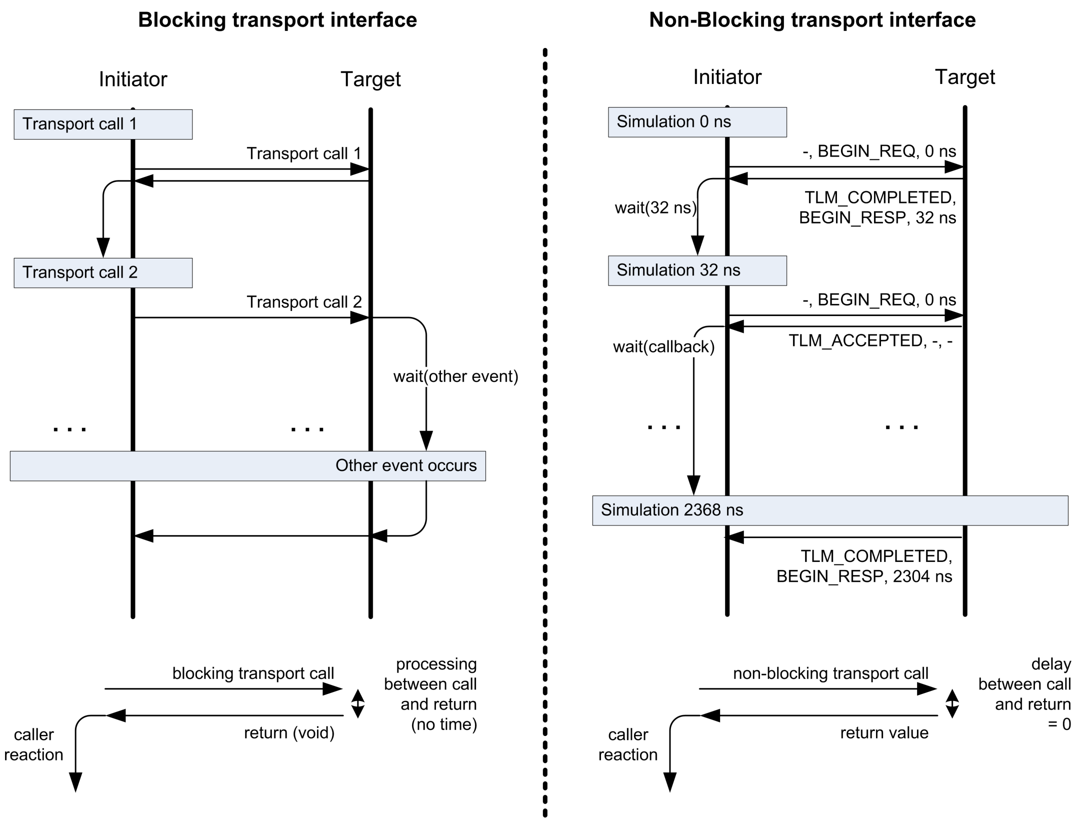


Figure 3.1.: Basic blocking and non-blocking transport behavior

Figure 3.1 contrasts the basic cases for the blocking and non-blocking transport interfaces. To the left, the initiator using the blocking interface issues the first transport call and the target instantly (which in the untimed style means without any wait for an external event) processes the transaction and returns. For the second call, the target transport function needs to wait for some event to occur and only afterwards the target finishes processing the transaction and returns. No synchronization values or annotations are provided.

To the right, an initiator issues its first request which is instantly (i.e. without an advance of simulation time) processed by the target which returns the annotated delay.

The initiator implements the delay and issues a second call. This time the target indicates the initiator to wait for a callback by the return value. Before returning though, the target would usually notify some external worker thread that will later perform the callback to the initiator. The transport function itself must not call wait. The synchronization values and annotations provided at transport function calls and returns will now be discussed.

The following phases mark the life cycle of a transaction in the context of a memory-mapped bus. Changes to transaction attributes may only take place along with phase transitions, also called *timing points*. In theory it is possible to use any phase enumeration data type in a user model. Since however the definition of the loosely- and approximately-timed modeling styles is intertwined with the default phases, an exchange would include a significant modification to the modeling styles as well.

```
// Standard TLM transaction phases
enum tlm_phase { BEGIN_REQ, END_REQ, BEGIN_RESP, END_RESP };
```

To leverage the potential for simulation speed increase, the LT style includes only the two begin phases which reduces the number of function calls and context switches. The more accurate AT style incorporates all four phases, attributing a distinct length to request and response as well as the time in between them, in contrast to LT that models requests and responses as singular events.

The return value of the transport call represents the action that the callee has taken with regard to the caller's request. It may have rejected the transaction, accepted it without changing anything yet, accepted it and updated the state or phase of the transaction, or completed it (but not necessarily semantically successful, this is indicated by special attributes). In the case of LT, an update automatically completes the transaction since there is only one phase to update to.

```
// Synchronization values returned by nb_transport
enum tlm_sync_enum { TLM_REJECTED = 0, TLM_ACCEPTED = 1, TLM_UPDATED = 2,
    TLM_COMPLETED = 3 };
```

It should be noted that phase changes of a transaction always apply to the same transaction instance. A transaction is created once with the phase BEGIN_REQ and, until completion, this object shall not be deleted or recreated. Like the transaction, the mentioned data structures are passed to the transport call by reference and thus always the same object instance is manipulated.

```
// Non-blocking transport forward function
template <typename TRANS=tlm_generic_payload, typename PHASE=tlm_phase>
virtual tlm_sync_enum nb_transport(TRANS& trans, PHASE& phase,
     sc_core::sc_time& t) = 0;
```

The time argument annotates a delay to the call that shall postpone the phase transition, if there is one. The target shall behave as if it had received the call only after this delay has passed in relation to the current simulation time. Similarly, if the target increases the delay during its processing, the initiator shall, upon return of the call, behave as if it had received the return value after the annotated span of time. It is the caller's responsibility to implement the simulation time advance upon return from the transport call.

The delay can be greater than zero already for the begin of a request (see Section 3.4.1) which effectively asks the callee to predict its future. The callee can either reply or only accept the request and wait until the delay has passed (*synchronize on demand*). For instance, a target with several reading and writing masters may have to establish R/W consistency and prevent out of order execution, or can accept the potential inaccuracy for increased simulation speed.

Similar to the transaction type, also the phase argument can assume an arbitrary user-defined enumeration type. For more details, see [Ope07b]. The models described in this work have been created with the default templates.

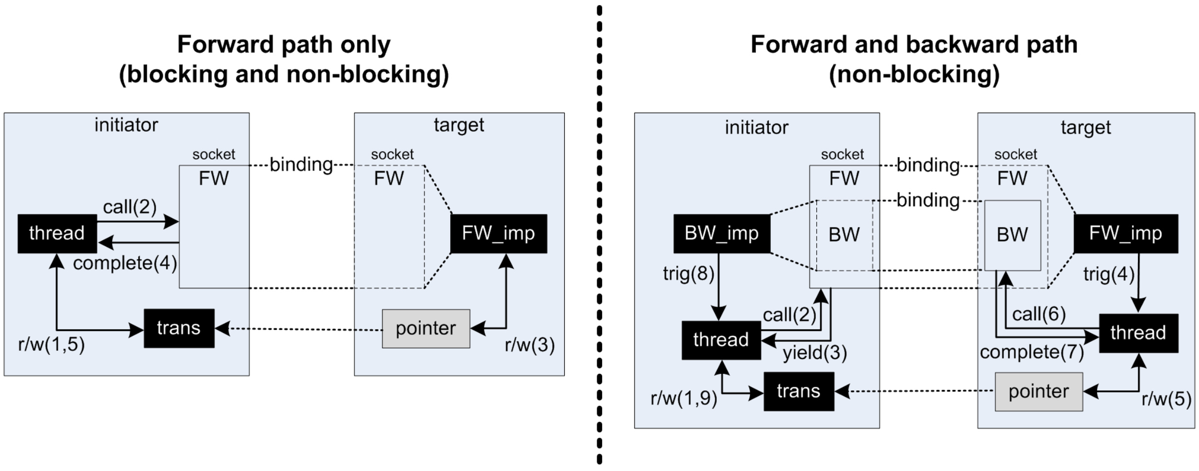


Figure 3.2.: Transport call through forward and backward path

Figure 3.2 illustrates the sequence of events that occurs during the use of a transport interface. In the left case, the initiator thread creates a new transaction (1) and then calls the forward interface (2), transmitting the reference to the transaction object. The transport function implementation processes the transaction by reading and/or writing to the reference (3) and returns *TLM_COMPLETED* in the case of a non-blocking interface (4). The blocking interface function of type *void* is assumed to always complete and simply returns (4).

In the right case, instead of completing the transaction after creation (1) and transmission (2), the transport function implementation tells the initiator to yield (3) and wait for a callback since the processing is not carried out immediately. The forward transport function then triggers the target thread (4) which processes the transaction (5). The thread subsequently calls the backward interface implementation in the initiator (6) which indicates the transaction completion to the target (7) and then notifies the suspended initiator thread (8)that can now access the completed transaction (9).

Direct memory interface

The DMI interface allows models to communicate with other models without constantly going through a transport interface, reducing the number of interconnect components involved and thus reducing the number of function calls, events et cetera. It returns a direct pointer to the target module memory that can be directly read/written, optionally annotating a latency. It can be used to speed up simulation of memory read/writes by bypassing the chain of transport calls, at a loss of simulation accuracy since a simple latency annotation can generally not be as accurate as the actual interconnect behavior.

The initiator requests the DMI pointer for a certain address of the target's memory. The target announces in its reply to which region of its memory it grants the desired type of access (read/write/both), and which latencies are involved for read and write. This indication is performed by setting the data members of a data structure whose reference is sent by the initiator along with the initial request. If this request is successful and the data members are set, the initiator can hence use the DMI pointer for direct read and writes. The pointer stays valid beyond the lifetime of the DMI request and has to be invalidated by the target through the backward path.

```
// DMI forward function
virtual bool get_direct_mem_ptr( const sc_dt::uint64& address, DMI_MODE&
    dmi_mode, tlm_dmi& dmi_data ) = 0;
```

The request for the DMI pointer initially passes through the transport chain, along the forward path, and delivers the target's response when returning. This procedure resembles that of the transport interface depicted in Figure 3.2, left. The transport transaction is simply replaced by the DMI request and its data pointer by the pointer to the DMI data structure. There is no callback in the backward path, but only the invalidate function.

```
// DMI backward function
virtual void invalidate_direct_mem_ptr( sc_dt::uint64 start_range,
     sc_dt::uint64 end_range ) = 0;
```

3.2.3. Supplementary interfaces

Debug transaction interface

This interface lets debuggers access the storage of a model. Debug calls follow the same forward path as the transport calls used for normal transactions to allow the same address translations, but without side effects like waits, delays, notifications or others. They enable instantaneous reads and writes to/from the module.

```
// Debug forward function
virtual unsigned int transport_dbg( tlm_debug_payload& r ) = 0;
```

The debug interface is used for quick initialization at the start of the simulation, to peek the data contents of modules during simulation, or to alter their value by non-intrusive means. It does not define a backward interface.

Analysis interface

The analysis interface's purpose is the non-intrusive duplication of (user-defined) transactions to subscribers for later analysis/evaluation. It uses a dedicated kind of port that can transport an arbitrary type of transaction and can thus contain any type of information which the user desires for his analyses. In contrast to normal sockets, the binding of one analysis initiator to several analysis targets is possible.

This interface is not part of the convenience sockets defined by the TLM standard, and is thus not a necessary prerequisite for their implementation.

3.3. Basic TLM sockets

The interfaces described in Section 3.2 can, without any further steps, be used to create models without infringement of their functionality. To do this, it is necessary that a model class inherits from these interfaces and implements them. But for ease of use and reusability, the individual interfaces are grouped into *combined interfaces* and then joined with SystemC ports and exports (Section 2.2.2) in the TLM standard *basic sockets*. They therefore require a port binding comparable to the one already known from RTL SystemC or VHDL. In contrast to the latter two, TLM knows three different types of socket binding.

3.3.1. Combined interfaces and basic sockets

The following *combined interfaces* are defined based on the *core interfaces* described in Section 3.2.

tlm_fw_b_transport_if Inherits from the blocking transport, the forward DMI and the debug interface

tlm_bw_b_transport_if Inherits from the backward DMI interface

tlm_fw_nb_transport_if Inherits from the non-blocking transport, the forward DMI and the debug interface

tlm_bw_nb_transport_if Inherits from the non-blocking transport and the backward DMI interface

These combined interfaces characterize sockets as a whole. An untimed initiator socket (where untimed implies using the blocking transport interface) for instance implements the class *tlm_bw_b_transport_if* which contains all related backward interfaces. Since however the blocking transport interface does not provide a backward path, the only appropriate interface is the backward interface class of the DMI interface.

Most interfaces have asymmetric forward and backward paths because the two paths do not offer the same functionality. For instance, the DMI forward path requests memory access while the backward path invalidates it. The non-blocking transport's forward and backward interfaces however consist of the one same function header as listed in Section 3.2.2, since both paths offer the same functionality i.e. the indication of a phase transition. The modeling style in turn regulates which phase transactions are allowed to take place via a certain path and in which way.

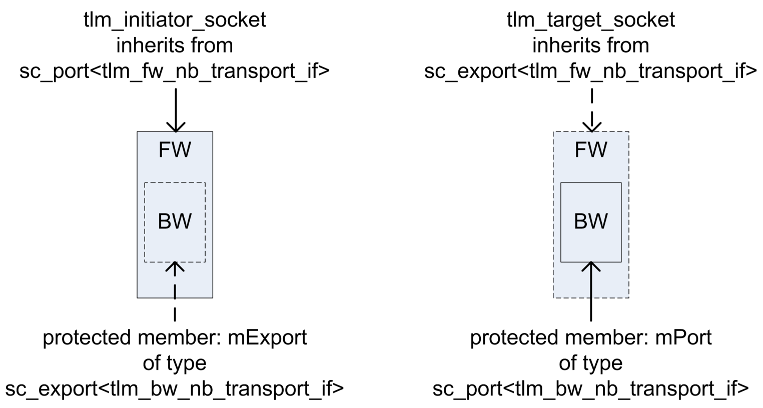

tlm_initiator_socket
inherits from
sc_port<tlm_fw_nb_transport_if>
FW
BW
protected member: mExport
of type
sc_export<tlm_bw_nb_transport_if>
tlm_target_socket
inherits from
sc_export<tlm_fw_nb_transport_if>
FW
BW
protected member: mPort
of type
sc_port<tlm_bw_nb_transport_if>

Figure 3.3.: TLM standard basic sockets

As depicted in Figure 3.3, the two prime types of TLM standard basic sockets are the initiator and the target socket. They both consist of one port and one export. In classical mapping, when socket is mapped to socket, the port is mapped onto the export (respectively). In a separate step, the respective exports are mapped onto the implementation (compare Section 2.2.2 and Figure 2.4). The port is the interface that is superficially called by a model, while the body that is really executed is the export to which that port is mapped. Therefore the target socket's export is of the forward interface type, while its port is of the backward interface type, and vice versa (for details, see Section 3.3.2).

The brackets indicate that port and export can in principle be of any interface type, as long as they match for mapping. In practice, the TLM basic sockets use either the blocking or the non-blocking interface types to determine the type of modeling desired (timed or untimed).

Since the elements of this section are abstract classes, the implementation must be provided by the model.

3.3.2. Socket binding

Due to the overloading of the binding operator known from SystemC, applying it on sockets can have three different meanings. Binding 1 and 2 are the minimal bindings required to receive a functioning model, while binding 3 is only required for a hierarchical design.

Binding 1 - Export to implementation

The binding depicted in Figure 3.4 is called on a socket. The argument has to be an object that inherits from and implements the same interfaces like the ones used for declaring the socket. A SystemC module inheriting from and implementing an interface is a channel (Section 2.2.2). The call effectively binds the socket export to the channel, or rather to the implementation of the export interface type (since a port is just a pointer to a channel).

The argument must not be a socket itself. The nature of the following two binding methods and the fact that a socket itself inherits from an interface would prevent the compiler to determine a socket-to-socket or a socket-to-implementation binding is desired.

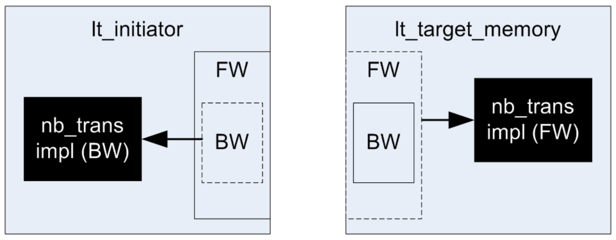


Figure 3.4.: Socket export to implementation binding

For example, *lt_initiator* inherits from the backward interface and includes an *initiator_socket* member. Calling *initiator_socket(*this)* in the *lt_initiator* constructor binds the backward export of the socket member to the local implementation. The same applies symmetrically to *lt_target_memory*.

Binding 2 - Initiator to target

The binding illustrated in Figure 3.5 is the classical binding of an initiator to its target. It is called on the initiator socket with the target socket as argument and binds the initiator port to the target export, and the target port to the initiator export. It is typically used for binding socket-to-socket at the same level of hierarchy. As a result, and in combination

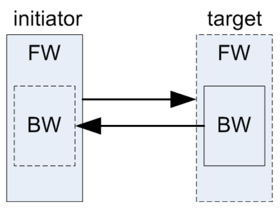


Figure 3.5.: Initiator to target socket binding

with binding 1, calling either port effectively calls the function bound to the export of the opposite socket. Multiple binding of sockets is not possible.

For example, the initiator socket is bound to its target counterpart by *initiator(target)* which comprises binding the initiator forward port to the target forward export and the target backward port to the initiator backward export.

Binding 3 - Hierarchical binding

Figure 3.6 shows the binding used to connect sockets up and down through the model hierarchy between child and parent socket. It is called on a socket with another socket of the same type as argument. It binds port to port and export to export.

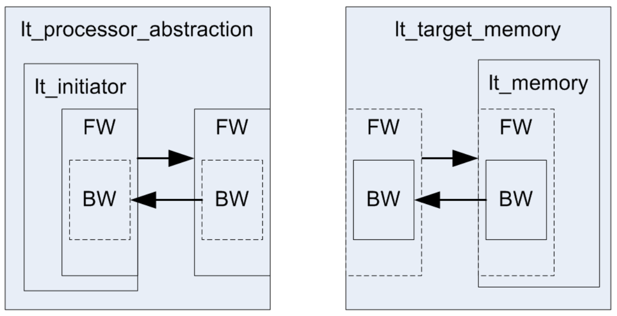


Figure 3.6.: Hierarchical socket binding

Since the called socket is always bound to the argument socket, the order of binding is important. When binding initiator to initiator, the child must be bound to the parent by calling *child(parent)*, going up the hierarchy. For target to target binding, the parent must be bound to the child by *parent(child)*, going down. In combination with binding 2 this means that going upward means binding port to port, then port to export at the top level, and export to export going down the hierarchy.

3.4. Model development in practice

So far we discussed only the classes that are already provided by the standard itself and available as soon as the TLM header is included in a design. With the exception of the classes described in Section 3.1, the standard supplies only declarations and no implementation. Included in the TLM2 draft 2 download archive, based on these declarations, comes a collection of example designs that can either be directly employed in a design or serve as blueprint for the implementation of user designs.

Classification of modules

Considering the different contexts encountered, we can distinguish the following categories of modules that constitute a system model. As we will see in Chapter 4, the context of the module can have implications on the general design approach.

Computation module A module whose prime task is to transform data, usually received and transmitted by its interface sockets or other interfaces.

Leaf module A computation module whose function, in particular the completion of a transaction, does not depend on nested transactions. A nested transaction is initiated between receiving a request and completing the associated response.

Non-leaf module The complement of a leaf module, i.e. the module requires additional nested communication to complete a request.

Communication module A module whose prime task is to manage the communication between two computation modules, requiring only the address information of a transaction.

Interconnect module A communication module which relays only one class of transactions (i.e. preserving the transaction type) without changing the data payload of the transaction. The *SimpleBus* (Section 3.4.2) is such an interconnect module.

Bridge A communication module which relays more than one class of transaction (possibly translating between the transaction types) without changing the data payload of the transaction.

3.4.1. Modeling options in LT and AT

The following standard situations are representative for the communication over the non-blocking transport interface in the loosely-timed (LT) and approximately-timed (AT) modeling style. The nature of the blocking interface and the lack of timing do not permit similar options for the untimed modeling style.

LT with annotation The standard case with precognition. The initiator sends the request with zero delay, the target replies with TLM_COMPLETED. The target knows and annotates the delay that the response is supposed to be delayed compared to the request. The initiator usually implements the delay.

LT with sync No precognition from the target is available. It simply accepts[12] the request without delay annotation, indicating that the initiator should yield. At a later point in (simulation) time, the target performs a callback to send the response that the initiator confirms by returning a TLM_COMPLETED. This concept is similar to *AT with backward path* but includes only 2 instead of 4 phases that are available in AT.

LT with TD Similar to the standard case with precognition. The initiator does not implement the delay annotated by the target right away but sends additional requests, using the accumulated local time offset as initial delay annotation.

AT with backward path The four phase transitions are indicated by dedicated transport calls without annotated delays. The initiator begins the request phase by the first call. The target ends the request phase and starts the response phase by one or two[13] explicit calls. The initiator finally ends the response phase by yet another call. All calls are performed at different simulation times and are accepted by the respective recipients.

AT with annotation An AT target might as well annotate a delay and update the transaction phase during execution of the forward call, indicating the end of the request phase implicitly rather than by an explicit backward call. Similarly, an AT initiator can reply to the response begin by annotating a delay that indicates the response end which completes the transaction without another call. The annotated delays are implemented by the respective recipients.

[12]Accepting means to return TLM_ACCEPTED and preserving the current transaction phase.
[13]A response begin implicitly includes a preceding request end.[Ope07b]

In the case that a LT target's precognition is dependent on the individual request it may be necessary to use *sync on demand* which constitutes a dynamic switch between *LT with annotation* to *LT with sync* when a request can't instantly be completed.

In general, an annotation is not describing a processing delay or a span of time during which something specific happens, but it should be interpreted such that the related phase transition is delayed by that annotated time, without any additional assumptions on the nature or origin of that delay.

For more specific rules on the handling of delays and the calling of interfaces, see Section 4.5.1 and [Ope07b].

3.4.2. Implementing communication through sockets

Two implementation approaches to equipping a module with facilities for communication through sockets could be identified.

Basic socket member and interface inheritance The module class has a basic socket member and inherits from and implements the socket member's export interface, mapping the export onto itself in its constructor.

Basic socket inheritance and subclass interface inheritance The module class inherits from a basic socket and thus becomes a socket itself. It defines a subclass that inherits and implements the basic socket's export interface, and in its constructor maps the export onto the subclass implementation.

A third conceivable approach is a module class that inherits from a basic socket and simultaneously from the required interfaces. The module class would implement the interface functions and map itself as a socket onto itself as an interface implementation. As already mentioned in Section 3.3.2, this approach is not feasible since the compiler would not be able to determine whether the desired mapping is socket-to-socket or socket-to-interface.

The TLM2 draft 2 kit includes a collection of examples that illustrate how targets and initiators (not just sockets but e.g. memories) can be implemented using the first approach, in which the module class inherits from the interface classes. By doing so, the interface functions become part of the module member functions and are thus mixed with the module-specific member functions that are not concerned with communication aspects. Communication and computation functionality is thus intermingled in the same class. While this is a straight forward approach for simple modules which do not require diversified communication capabilities, a more elaborate approach should be taken to promote modularity between sockets and the modules using them.

SimpleSocket

The *SimpleSocket* class is an example design that takes the second approach. It inherits from a basic socket and includes a private subclass that inherits from the basic socket's export interfaces (see Figure 3.3). The constructor of the SimpleSocket class then maps the basic socket's export on the implementation within the private class via binding 1 (Section 3.3.2). By doing so the SimpleSocket's communication behavior is kept well separated from the module's computation behavior. The following code listing illustrates this concept.

```
// SimpleLTTargetSocket inherits from a basic TLM socket
class SimpleLTTargetSocket : public tlm::tlm_target_socket< ... >
{
  // The constructor maps the export to the implementation in mProcess
  explicit SimpleLTTargetSocket(const char* n = "") : ...
  {
    // mapping target forward interface port to implementation
    (*this)(mProcess);
  }

  // Process is a private subclass that inherits from the export interface
  private:
  class Process : public tlm::tlm_fw_nb_transport_if<TYPES>
  {
    // Now comes the interface implementation
    sync_enum_type nb_transport(transaction_type& trans,phase_type& phase,
                                sc_core::sc_time& t)
    {
      ...
    }
  }
}
```

In addition the SimpleSocket class manages function pointers for each method required by its interfaces. As a result the user can decide for each instance if it should use its built-in default behavior (by registering no function pointer), or if a different behavior is desired for the respective module. See Appendix C.1 for illustrative code.

When it is possible, the default behavior replies to a phase transition according to the standard and notifies events that enable the outside module to react to the transition. In principle, it denies any special requests like DMI or debug due to the ignorance about the module around it, and also cannot offer a default behavior for the transport target because the transaction handling in the target is highly application dependent. It is thus advisable to register pointers for all transport functions. Other pointers can be left unregistered if no special action except the TLM-standard default negative answer to the caller is required.

The SimpleSockets (originally included for LT) and their adaption for the UT and AT modeling style will be the sockets of choice for the modeling done in this work. They are referred to as *SimpleUTSocket*, *SimpleLTSocket* and *SimpleATSocket* and can be employed in one of the two following ways.

SimpleSocket and function pointer The module class has a SimpleSocket member, implements functions with the same signature as the ones included in the required interfaces and registers appropriate function pointers.

SimpleSocket and default behavior The module class has a SimpleSocket member and employs the default behavior of the SimpleSocket.

The first approach appears similar to having the module inherit and implement the socket interface. The main advantage is that by using SimpleSocket with function pointers, the user implementation can be named more descriptive and can be chosen more flexibly be reassigning the pointer, or not registering it at all.

SimpleBus

This example design provided by the TLM2 draft 2 manages an array of SimpleSockets (both initiators and targets) by a unique address that is valid only within the bus component. This address enables the SimpleBus to keep track of which pending transaction belongs to which socket pair. It enables cross-over communication from each initiator to any target by generic payload transactions and supports both AT and LT timing mode as well as the dynamic switching between the modes (see also [Gro02]).

The SimpleBus will be the basis for the bus model used in Section 4.6.3.

Chapter 4.

Case Study: Advanced Encryption Standard

This chapter describes the modeling case study of an application from its abstract description to an approximately-timed model supporting architectural exploration. Since previous experience has been established in [AHL+06], the *Advanced Encryption Standard (AES)* has been chosen as example application.

4.1. Description of the algorithm

The *Advanced Encryption Standard*, also known as *Rijndael algorithm*, is a symmetric block cipher that generates cipher text out of plain text by iterative transformations [Nat01]. It employs the same key for encryption and decryption. The encryption involves a certain set of operations carried out on the data (that we refer to as *state*) as well as on the key. The results are the so-called *round state* and the *round key* for a given round number. The combination of the round key with the corresponding round state concludes an *encryption round*.

4.1.1. The standard AES algorithm

We are considering the 128-bit version of AES in the *electronic codebook (ECB)* operating mode, comprising 11 encryption rounds - 1 initial, 9 regular, 1 final. The initial round consists of only the XOR combination of the initial state with the initial key, yielding the first round state. The process of round key creation is referred to as *key scheduling* or *key expansion*, and operates on 16 key bytes that are regarded as 4x4 matrices. Since the calculation of a new round key is only dependent on the previous round key, it is independent of the *data encryption* process that in turn operates on 4x4 matrices of state bytes. Thus, the generation of all 10 round keys can either be carried out beforehand (offline), reusing the stored round keys for each new block of data, or during the encryption (online), creating and storing only the next required round key.

Data encryption

Figure 4.1 depicts the sequence of data encryption operations. [AHL+06]

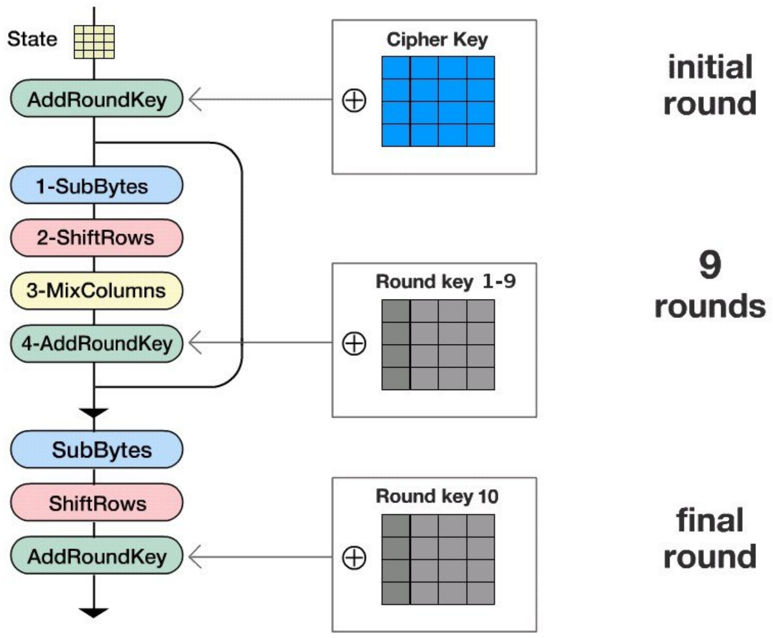


Figure 4.1.: Sequence of AES rounds [Zab03]

1. *Substitute Bytes (SubBytes)* - each state byte is used as address value to look up the substitute value in a so-called S-Box (i.e. a 256 byte array).
2. *Shift Rows* - rotation of state row n by n bytes.
3. *Mix Columns* - multiplication of each state column with a constant 4x4 matrix, containing only the values 01, 02 and 03.

4. *Add Round Key* - XOR combination of state and key, constituting the initial round and ending all other encryption rounds.

Key scheduling

The generation of the next round key comprises the following operations on the current round key, in the order of listing. [AHL+06]

1. *Rotate Word (RotWord)* - rotation of the last column of the current round key by one byte.
2. *Substitute Bytes (SubBytes)* applied to the last column of the current round key.
3. *Rcon* - XOR of the rotated and substituted column with a column taken from a given constant matrix.
4. The first column of the next round key is generated by XOR of the first column and the transformed last column of the current round key.
5. The remaining 3 columns of the next round key are generated by XOR of column i of the current round key with column $i-1$ of the next round key.

4.1.2. The T-Box variant

It is possible to merge the SubBytes, ShiftRows and MixColumns operations of the data encryption process into one single lookup (similar to the S-Box lookup of the standard algorithm) [FD01, MS04, RSQL04]. This extended lookup array is referred to as T-Box. It can be divided into three distinct S-Boxes, called S1, S2 and S3, that each contain different values and anticipate the MixColumn operation. While the data encryption process requires all three S-Boxes, Key scheduling requires only S1 lookups since it does not use a MixColumn operation.

Generating one byte of the next state comprises the lookup of four bytes of the current state (two in S1, one in S2, one in S3) and their XOR combination. To implicitly implement the ShiftRows operation, the diagonal of the current state is used to generate a column of the next state. Rotating the input pattern by one over the same four current state diagonal bytes delivers the respective next state column byte. The input for the subsequent next state column is selected by shifting the diagonal window on the current state horizontally by one. The first generation using the yet unrotated new diagonal as input now generates the last byte of the second next state column (rotated vertically) and so on. Figure 4.2 illustrates the data encryption using the T-Box variant.

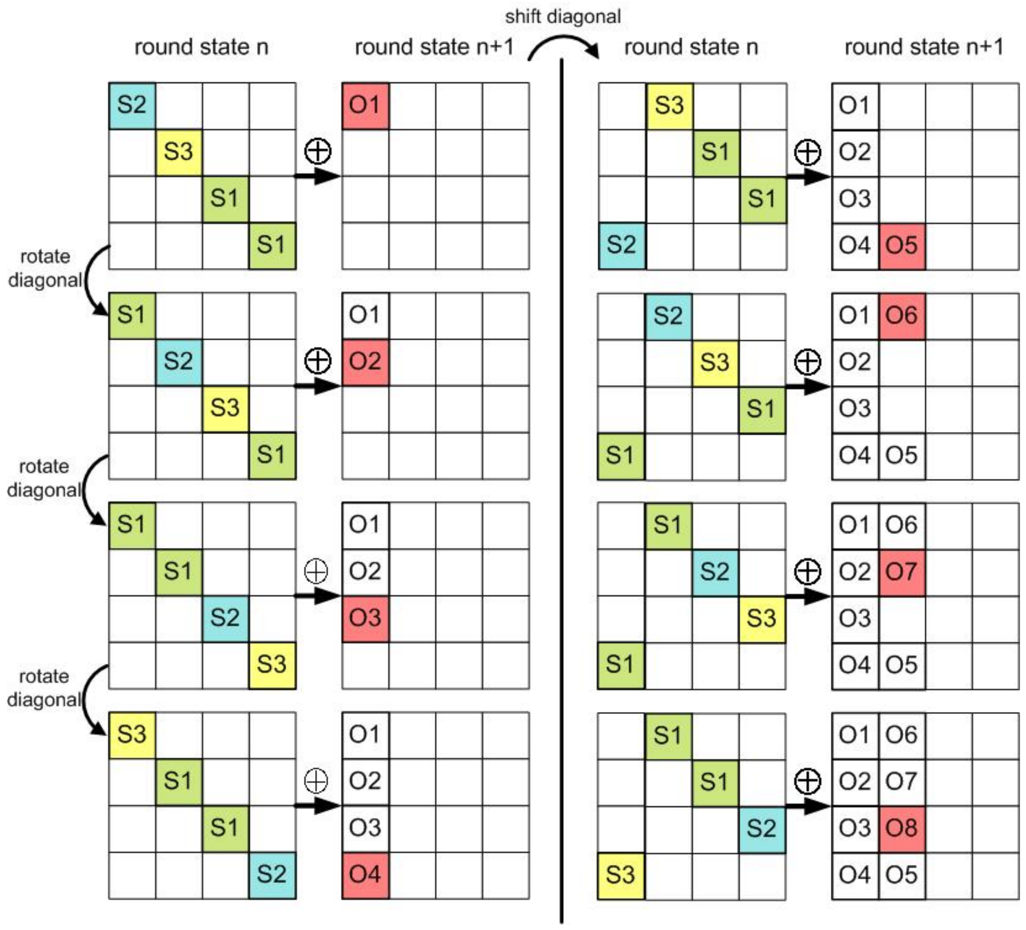


Figure 4.2.: T-Box data encryption [AHL+06]

4.2. Outline of the design flow

The modeling styles of TLM can be applied to a variety of use cases and thus in differing order. The case study follows the design flow described below.

1. Implement the algorithm in *software* to understand the involved computations.

2. Create an *algorithmic* system model based on the software implementation as functional reference and for deriving a partitioning into distinct modules.
3. Introduce a notion of time by refinement to the *loosely-timed* modeling style.
4. Based on that notion of time, conduct a *preliminary hardware/software tradeoff analysis* to gather information for the next refinement steps.
5. Refine to the *approximately-timed* modeling style for *architectural exploration*.

The steps that are required to meet the respective goals will be described in the respective sections.

4.3. Software implementation

Based on the algorithm outlined in Section 4.1, the initial step is its implementation in software, using the round-wise output of [Zab03] as initial lead. The C++ code is then meant to serve as a functional reference throughout the rest of the modeling process and as a first step of understanding the computations that are required by the application. The algorithm is split into coarse subunits, that become manifest in four methods that are called by the main control flow, and which execute distinct sub-algorithms.

- ScheduleKey
- NextState
- NextStateFinal
- AddRoundKey

With the exception of the first round, which only comprises an AddRoundKey, the functions are executed in this order. Since several operations of the standard AES algorithm are wrapped up in lookups, a partitioning of the T-Box algorithm results in more coarse functions. One particular result is the differentiation of the final round from the regular encryption rounds. The core data structure of the implementation are 16-byte buffers (*uint8[16]*). Their manipulation by NextState(Final) requires a temporary buffer because the generation of a new state by the T-Box algorithm requires the old state to remain unchanged until completion. ScheduleKey can be executed directly on the buffer. AddRoundKey could be executed directly as well but is implemented using a temporary buffer. This simplifies the control flow since the result is written back into the original buffer from where NextState(Final) previously read its input.

4.3.1. Control flow considerations

The control flow can be implemented in several ways. One possibility is to preliminarily schedule all round keys and have them instantly available to AddRoundKey (*offline key scheduling*). The downside of this approach is the initial idle time for AddRoundKey, the subsequent inactivity of ScheduleKey and the preliminary assumption of a sequential order of the two functions.

Alternatively, ScheduleKey could run loose without any synchronization. AddRoundKey requests keys on demand and has to wait only if ScheduleKey is yet busy with the requested key. This implicitly already assumes that the two tasks will be carried out concurrently, a decision that is not part of this phase.

Ultimately, the control flow is implemented in such a way that ScheduleKey calculates just one key at a time (*online key scheduling*) and continues with the next key only when the last one has been consumed by AddRoundKey. This intertwining of actions, involving periodic synchronization points, was found to be most appropriate for the modeling task at hand since it was ignorant to whether the two functions execute sequentially or concurrently, a question that was yet to remain open in this phase.

4.3.2. Application-related considerations

In addition to the modeling-related discussion of the control flow, the same application-related assumptions as the ones mentioned in [AHL+06, page 6] are applied. Namely, that the application is assumed to exhibit small bursts of a few data packets, or possibly only one at a time, justifying the use of online scheduling. Offline scheduling would be preferable for long data streams per session key, where the scheduling overhead becomes negligibly small compared to the encryption process.[14]

[14] If these aspects of the application were not fixed in the beginning, they would need to be incorporated into the modeling evaluations. For the sake of simplicity, and due to the focus of this work, these assumptions are maintained.

The resulting set of software methods along with initial insights about the control flow and storage requirements will be the functional core of distinct simulation modules.

4.4. Untimed modeling

The intent of this phase is the implementation of a functional system model using the untimed modeling style. The advantage of performing the first modeling steps in this style is that yet irrelevant details like timing and communication phases can be postponed and focus can be laid on the modular composition and as yet simple communication between the sub-modules. The main difference between the software and the untimed functional implementation is that in software, some data availability requirements can remain hidden, while the untimed system model and its self-contained modules naturally reveal them.

4.4.1. Partitioning into computation modules

Analyzing the usage of the data buffers mention in Section 4.3 reveals the data dependencies among the four functions, which are illustrated in Figure 4.3. An incoming arrow means that a function needs the result of the function where the arrow originates before it can continue its computation. Note that ScheduleKey does not require any more input after being initialized with the initial key.

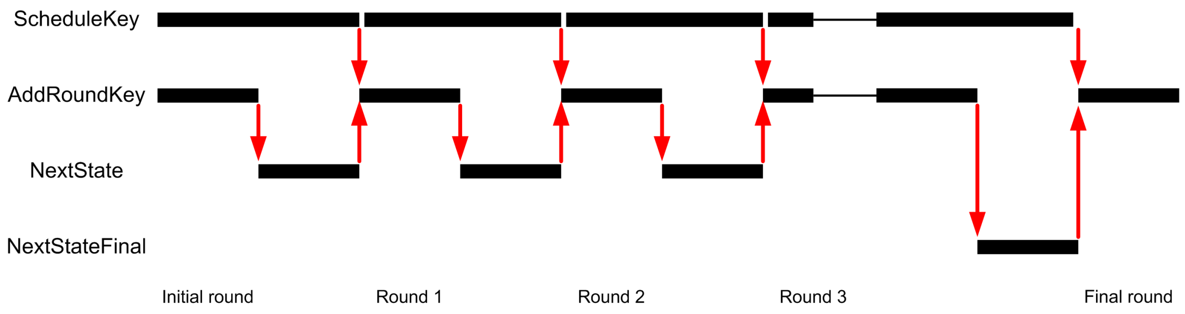


Figure 4.3.: Coarse AES data dependency analysis

One conclusion that can be drawn from Figure 4.3 is that each of the four functions is qualified to constitute a proper computation module. Still, two issues remain. First, NextStateFinal represents the special case of NextState in the final round and is only called once. Thus, it is merged with NextState into one module. Consequently, the round number associated with each data buffer becomes relevant for the NextState module. Second, the initialization with the initial key and state must be performed by some kind of master which, in the software version, was the method that would call the encryption procedure. In the system model this master will be represented as additional module that triggers the encryption by supplying the initial key and state to the encryption system, and that reads the cipher text after completion. The master shall otherwise not be involved in the processing and the interface should be subject to as little change as possible to maintain transparency. The master presents the user application that employs the AES encryption facility in its context.

While NextState and ScheduleKey exhibit enough functionality to justify their implementation as dedicated modules, AddRoundKey is performing only the XOR combination of their results. AddRoundKey is thus the natural synchronization point between these two modules and in addition the first module to perform an operation at the start of a new encryption. AddRoundKey is therefore the interface to the master module and in control of the encryption, implementing the previously derived control flow. Since the untimed modeling phase is intended to confirm the big picture and functional correctness of the algorithm implementation and the module composition, lookups are yet assumed to happen instantaneous and will be treated in a later stage. These considerations yield the system architecture depicted in Figure 4.4.

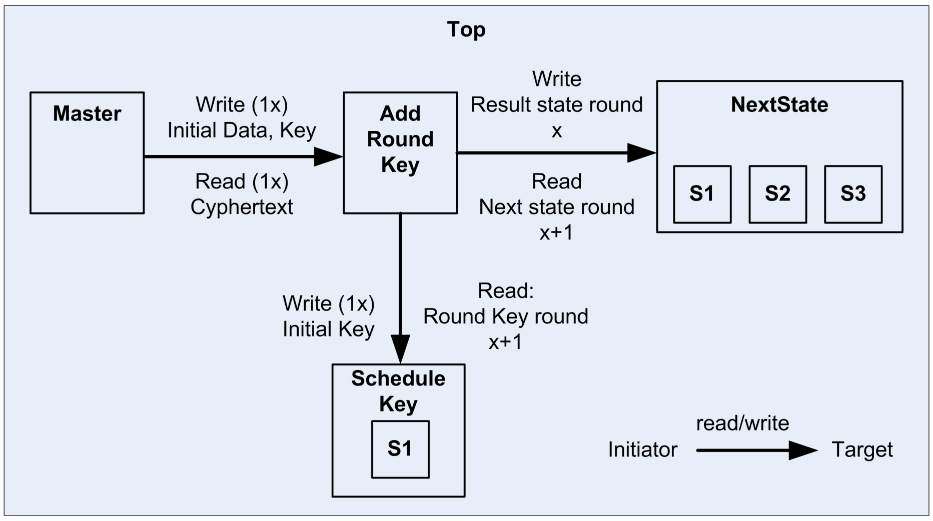


Figure 4.4.: System architecture with instantaneous lookup

4.4.2. Implementing an algorithmic model

Design guidelines

The prime focus of this untimed model is the creation of a functional reference for later timed models, while yet including as little complexity as possible. One goal therefore is to execute the simulation without the use of simulation time i.e. in a finite number of delta cycles.

Although we are not yet modeling communication procedures or communication modules, we already perform communication by *blocking transport* calls (Section 3.2.2). Later stages will involve a shift towards more complex communication. Adhering the principle of separation of communication and computation, communication modules should be kept ignorant of the transactions' semantic.

A conceptual separation between the software description of the functionality and the system model that executes the functionality shall be preserved. It should be possible to include the sub-algorithms (depicted in Figure 4.3) that are the core of the computation modules and that have initially been implemented in pure software in a way that does not require an extensive rewrite. The module implementation shall control the use of the respective thread or function while the function itself is ignorant of its environment.

Data structures

The choice of the transaction class as well as the data structure that is included in a transaction are key design decisions. The default transaction class is the *generic payload* as described in Section 3.1.1. The core data structure of the algorithm is a packet consisting of 16 bytes (either key or state). Since the final round consists of different operations than the other data processing rounds, the round number that a packet corresponds to must be known. There are several approaches to tag such a packet with the associated round number.

1. Create a new transaction class to replace the default *generic payload (GP)*.
2. Create a new transaction class by inheriting from and extending the GP.
3. Extend the GP data payload rather than the GP class itself.

Option 1 involves all the overhead required by the object-oriented design of the TLM facilities, and is only advisable if the communication context of a memory-mapped bus is totally inadequate for the task at hand. Since the model operates on a read/write basis, this is not the case.

Option 2 involves less overhead, but would introduce application-specific attributes at a level where they are not appropriate. The design rules in [Ope07b] state that any extension of the GP class with additional member variables must not be relevant to a target's functional operation. This however would be the case for the round number, if it were added as distinct class member to a new transaction class.

Rather, the data payload member of the default GP class will transport an extended data structure. This leaves the GP class itself unchanged (since the data payload is just a pointer to an external data structure) and corresponds to option 3. The data structure whose pointer is included in the GP transaction is implemented in a class *aes_payload* that comprises the 16-byte data matrix member and an additional member value containing the associated round number. Note that because the round number is relevant only internally within the AES system, the master module issues transactions carrying the basic 16-byte data member.

Implementation

A *top* module instantiates the system modules, performs the port binding and triggers a packet stream by performing a notification in its simulation control thread. A new notification starts another data stream. The master generates the stream by repeatedly calling *b_transport* to AddRoundKey to deliver the initial key and data of each new packet.

Within AddRoundKey, the thread Encrypt which contains the encryption control can be initiated in one of two ways. Either Encrypt is implemented as a function that is called by the second transport interface call which completes the initialization. So upon return from the second transport call, the cipher text is available for instant read by the third transport call. Or Encrypt is implemented as a simulation thread and the second transport call notifies an event to which Encrypt is sensitive. Because the master has not yet yielded, allowing no other thread to run yet, the third transport (cipher text read) happens before Encrypt has started and during the third call the master thread is suspended until a done-event is notified by Encrypt. The main difference is that in the first case, the encryption remains in the context of the master thread, while in the second case, the master yields simulation control to the scheduler which thereupon schedules Encrypt (see Appendix C.2 for an illustrative code). Therefore only the first version is an algorithmic model, which is defined to have only one thread of execution[15].

Encrypt performs the encryption along the control flow outlined in Section 4.3.1. It calls NextState and ScheduleKey by sending transactions containing all relevant information. It first writes the initial key to ScheduleKey, subsequently sends a read transaction for each new round key, and sends a pair of transactions (first write then read) to NextState for each round. The write transaction includes the result of the last AddRoundKey while the read transaction retrieves the next round state. The handling of the round number in the transaction data structure requires no intelligence by the worker modules. Since simulation time never has to be advanced, the simulation terminates after *zero time*.

Model breakdown

The following classes are used for implementing the untimed algorithmic model.

- Generic payload
- SimpleUTSocket
- Blocking transport interface

The model consists only of leaf modules.

4.5. Loosely-timed modeling

Based on the untimed model described in the previous section, the next step is to introduce the concept of time. The loosely-timed model shall be able to generate and implement an advance of simulation time by adding up the delays of an operation sequence and call, at some point, wait or a delayed notification. Communication shall be structured into the two phases *BEGIN_REQ* and *BEGIN_RESP*, possibly performing computation before the phase transition.

[15]Since the simulation control thread of the top module is not part of the application, the system model is still considered to be an algorithmic model.

4.5.1. Relevant design rules for timed models

The following rules from [Ope07b] summarize the guidelines for handling delays in timed models. For a given modeling style, some rules may be further restricted (e.g. ban of temporal decoupling in AT).

1. The nb_transport method shall not call wait, directly or indirectly.
2. An nb_transport call on the forward path shall under no circumstances directly or indirectly make a call to nb_transport on the associated backward path, and vice versa.
3. A timing annotation using the sc_time argument shall delay the phase transition, if there is one.
4. On return from nb_transport, it is the responsibility of the caller to behave as if it had received notification that the transaction will change state at time sc_time_stamp() + t. In other words, the time argument is used to annotate latency to the nb_transport call, and it is the caller's responsibility to realize that latency.
5. On return from nb_transport, the caller has three options for implementing an annotated latency. It can run in temporally decoupled mode, it can put the transaction into a payload event queue (or similar), or it can call wait(t) (assuming the caller is a thread process).
6. Transactions may be pipelined. The initiator could call nb_transport to send another transaction to the target before the delay returned from the first call had elapsed. It is the responsibility of the initiator to account for the delays as it wishes.

4.5.2. Model migration from UT to LT

In the simplest case, the transition from the untimed to loosely-timed style is performed by replacing the calls to the *void b_transport* function with a transaction argument by calls to *nb_transport* functions with transaction, phase and delay annotation arguments which always returns TLM_COMPLETED and the associated delay. Doing this for our untimed model, we get a sequence as depicted in Figure 4.5, left.

In accordance with rule 5 the master calls the transport with zero offset, expects an instant completion and implements the delay explicitly by wait for both reads and writes.

Since the model still uses Encrypt as a function call and follows the same control flow as the untimed version it is still executing the subtasks sequentially. Rule 1 requires that the encryption is carried out without calling wait, because it happens in the context of the master's read request and thus in AddRoundKey's transport interface implementation. So the delays of all intermediate steps are added up and, eventually, explicitly implemented by the master. No other fundamental changes to the untimed model are yet applied. This results in a situation that deserves special consideration.

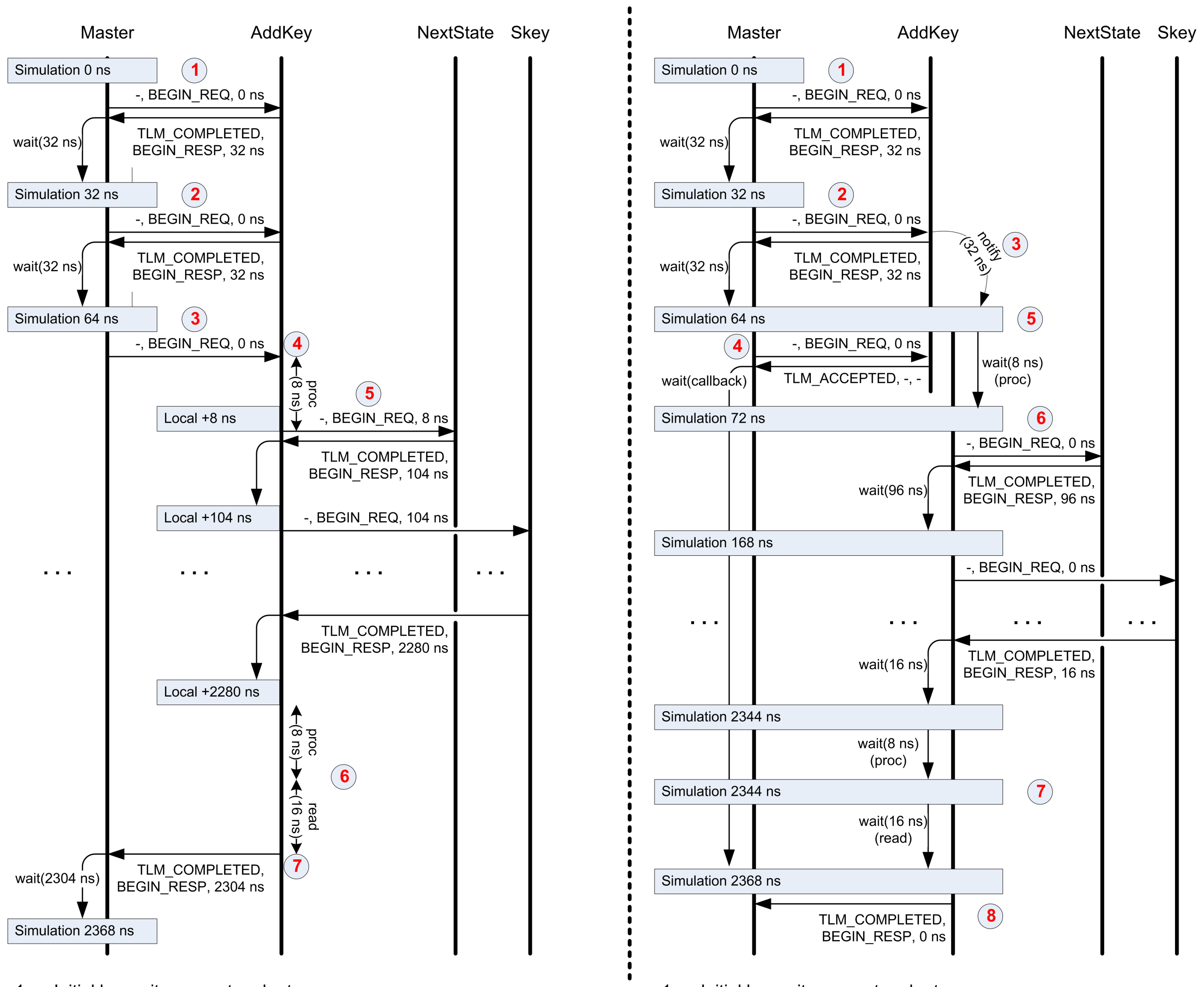
4.5.3. Removal of temporal decoupling

The fact that the same accumulating delay value is used as the annotation for the nested transport calls from AddRoundKey to NextState and Skey represents a case of temporal decoupling. This poses no problem because the modules know the time they require for completion of their time-invariant computation and the model complies with rules 1, 4, 5 and 6.

Since this is not the general case we remove the temporal decoupling. This requires AddRoundKey to call wait after each transport, which would violate rule 1 if it happened during the master's transport call. Encrypt is therefore implemented as a thread that is notified by the master's read call. By explicitly implementing the delays annotated from the other modules, Encrypt can perform its transport calls with zero offset. The master's read request can not anymore be completed before returning from the transport call, so it must now support sync on demand by handling a return value of TLM_ACCEPTED and providing a callback transport function.

When AddRoundKey receives the read request, it decides dynamically if it is able to complete it by returning the cipher text or if it only accepts the transaction and indicates the master to yield. AddRoundKey remembers the choice and performs a callback at the end of the processing.

The standard rules are unclear whether just a forward call or also a backward call with non-zero offset would constitute temporal decoupling. To be on the safe side, AddRoundKey explicitly implements the accumulated delay and performs the callback with zero offset. The alternative is a non-zero callback offset and a delayed notification by the master's backward transport function to the callback event. In the current model this is effectively the same behavior, but with an additional assumption on the behavior



1 Initial key write request and return
2 Initial state write request and return
3 Cypertext read request
4 Start of processing by calling Encrypt function from within transport function implementation
5 Temporally decoupled transport calls
6 Encrypt function returns after „proc“ delay
7 Transport function calls back with cypertext read response

1 Initial key write request and return
2 Initial state write request and return
3 Transport function notifies Encrypt thread with delay
4 Cypertext read request and return
5 Encrypt thread begins to run
6 Transport calls from Encrypt to Nextstate and ScheduleKey
7 Implementation of processing and read delays by Encrypt
8 Encrypt thread calls back with cyphertext read response

Figure 4.5.: Loosely-timed encryption with/out temporal decoupling

or the master. Figure 4.5, right depicts the sequence for the conservative solution. Please note that both sequences are still sequential and thus transport calls to NextState and ScheduleKey are not carried out concurrently. We will introduce concurrency in the next section.

4.5.4. Introduction of concurrency

The previous models were still based on the sequential nature of the initial software implementation As can be seen in Figure 4.3 however, several actions can be executed concurrently to one another. Already the addition of a second thread to remove temporal decoupling in the previous section introduced the notion of concurrency between Master and AddKey as depicted in Figure 4.5, right. There already concurrency had to be considered to maintain the same simulation behavior, by delaying the encrypt event notification by the write delay. Otherwise this delay would have been lost compared to the non-concurrent temporally decoupled case. Temporal decoupling was not causal to the sequential nature of the first model though. Rather, removing temporal decoupling by introducing a second thread implied the concurrency issues.

SystemC by itself cannot achieve something like a concurrency analysis. What it can be used for is to model and evaluate a certain concurrency scheme, but this scheme has to be derived by different techniques. We can use the scheme that is already hidden in the data dependency analysis depicted in Figure 4.3. The term *scheme* refers to the fact that it indicates what task can or cannot be carried out concurrently with others, which is the first requirement for evaluating execution times and different concurrency schemes to achieve well-distributed load or performance scenarios.

In principle there are two possibilities that enable introducing and taking advantage of concurrency.

- Perform computation and communication in separate threads coordinated by delayed notifications and waits for events.
- Perform several concurrent communication activities (e.g. to different target modules) in separate threads.

To cope with several concurrent requests to the same computation module, a busy flag is introduced. A queue structure for requests (like the *payload event queue* of

Section 3.1.2) is needed to store transactions for later processing. Temporal decoupling or instant replies without yielding can become very tricky, or even impossible.

Introducing concurrency into our model

An initial concurrency scheme has been identified by data dependency analysis (Figure 4.3). Every action (both computation and communication) that shall be concurrent with any other action requires an own thread. Enabling AddRoundKey to communicate with NextState and ScheduleKey concurrently therefore requires two communication threads. As additional benefit, this enables NextState and ScheduleKey to perform their computations concurrently to each other as well without requiring a thread of their own. They can implement their respective computation as functions that are called by their transport interface implementations. The communication threads in AddRoundKey subsequently implement the delays annotated by these computation calls after return from the transport call. AddRoundKey waits for the completion of both communications and continues only after state and key have been synchronized. So in addition to the Encrypt thread, AddRoundKey now includes two additional communication threads.

The fact that NextState and ScheduleKey not necessarily require dedicated threads to perform their work is beneficial to the simulation speed. However, concurrency also means that a target which is able to process only one transaction at a time can still receive several requests from different initiators simultaneously. To minimize the number of assumptions on the behavior of any module a *queue+callback* approach combined with a busy flag is applied. To keep the modules as simple as possible, the target transport functions unconditionally queue the received transactions in a PEQ and return TLM_ACCEPTED while, depending on the busy flag, an independent worker thread on the other end of the queue performs the computation and callback. Therefore NextState and ScheduleKey each now include a worker thread to perform their function. See Appendix C.3 for illustrative code.

This separation of request from work/response requires more synchronization with the SystemC scheduler at a gain of increased flexibility, reusability and simplicity.

The busy flag is managed by a dedicated second thread and serves as indicator of the current busy status of the target, both internally for the worker thread and, if required, externally (including tracing). The worker thread assembles the delay values in zero time (in case of a leaf module) and notifies the busy thread which implements them. If the module is dependent on unknown delays from other modules, the busy thread itself must wait on appropriate events that are delivered by worker and communication threads. The at first glance useful distinction between computation and communication busy times that should allow better analyses of the concurrency scheme comes with certain complications, as described in Section 5.2.

4.5.5. Facilities for basic HW/SW tradeoff analysis

Using fast loosely-timed models (and their busy flags) can yield preliminary information about potential bottlenecks. Constantly active and thus critical components are candidates for concurrent hardware implementation. Uncritical components are candidates for sequential software, possibly sharing a common CPU. To gain this kind of insights from a model in an efficient way, it must offer certain capabilities beyond indicating busy and idle status. We will now introduce them into the model.

Easy timing variation

One main goal of hardware/software tradeoff analysis, architectural exploration and other model based evaluations is to determine the best execution time / communication time relation between the concurrent tasks or the criticality of the execution time of certain computations. Allowing easy variation of these parameters enables efficient simulation of different timing combinations and should be given special attention.

A straight forward approach is the use of a global header that contains a set of static variables which includes all delays present in the simulation. They can either be coarse for each module's computation, specialized for certain fine grain atomic actions that build the computation, or a mix of the two. The individual modules employ these variables in their wait and notification statements.

The master module issues data packets to AddRoundKey for encryption. The top module notifies the master to start a new encryption. To simulate two timing schemes, the top module sets the new delay scheme before the next notification to the master.

This approach assumes complete white-box control over the modeling and that all modules include the same header. A more flexible OOP approach could pass a reference to the module constructor that points to the shared variable, while the module internally waits for the reference rather than directly for the shared variable.

Signal tracing

The analysis interface (Section 3.2.3) enables very sophisticated and automated ways of simulation analysis. If just a simple evaluation is desired, tracing variables (not only signals) and their values, similar to VHDL, can already suffice. The busy flags are good candidates for such tracing.

The OSCI reference simulator does not include a waveform viewer but writes tracing information into a file for later review. Variables that shall be traced need to be public to be accessible. Tracing is set up before the start of the simulator and changing the tracing setup during simulation (create an additional tracing file, add or remove variables etc.), results in a segmentation fault. Therefore two timing schemes cannot be written to two separate trace files. A long wait between two schemes is recommended for easier reading.

There seems to be no possibility to trace strings or enumeration types. Tracing strings yields only the first character as binary, tracing enums yields their integer encoding. The trace function for enums is reported deprecated. Also, the *Value Change Dump (VCD)* format has, according to Verilog standard [IEE01], no definition of strings or enums and saves all values as bit vectors that are interpreted accordingly by a waveform viewer such as *Wave VCD Viewer*[16] (freeware for Windows), SynaptiCAD Wave-Viewer (Windows)[17] or *GTKWave* (Linux).

4.5.6. Bottleneck identification

Employing the outlined facilities for tracing and analysis, we will now derive and assign execution times to the various computation and communication steps between the modules to identify the most critical component. The focus is on a qualitative rather

[16]http://www.iss-us.com/wavevcd/index.htm
[17]http://www.syncad.com/

than a quantitative assessment and no optimizations related to implementation are assumed. Consequently, a XOR operation carried out in one module is assumed to take as long as in any other module. Table 4.1 lists the subtasks under consideration.

Task	required operations
Write/read of 16 data bytes	16 reads/writes (algorithm operates byte-wise)
NextState regular round	4x (4 lookups + 12 XOR) = 16 lookups + 48 XOR
NextState final round	4x (4 lookups + 4 assign) = 16 lookups + 48 assign
ScheduleKey round	1 lookup + 17 XOR
AddRoundKey	16 XOR

Table 4.1.: Subtasks of the AES algorithm model

The operations used above are now refined to the instructions that would be required to implement them on an abstract general purpose CPU (listed in Table 4.2). Each instruction is considered as 1 ns delay which corresponds to an operating frequency of 1 GHz. However this is just an aid to derive delays that do not presume unbalancing optimizations to the implementation of any part of the algorithm. Rather the goal of this analysis step is to determine the part that should be optimized in the first place. Any other scenario that preserves this balance would be appropriate as well. Also note that the structure of the model does not yet include a communication structure that would enable congestion modeling and that S-Box lookup is still ideal. Therefore this reasoning must not be confused with already presuming an implementation approach.

The two operations required for write/read are considered symmetric since a transfer medium (either a register file or a bus) is assumed to which the transfered value has to be written and read in both cases, but possibly reversed order. The assign operation is performed only after data has been retrieved by a read operations and is thus already available.

Operations	attributed delays
Write/read	load + store / store + load = 2 ns
XOR	2 operand loads + XOR + result store = 4 ns
Assign	store = 1 ns
Lookup	start address load + offset load + indirect read + store = 4 ns

Table 4.2.: Delays attributed to operations

Consequently, the subtasks are annotated with the following delays listed in Table 4.3.

Task	assigned delay
Write/read of 16 data bytes	32 ns
NextState regular round	256 ns
NextState final round	112 ns
ScheduleKey round	72 ns
AddRoundKey	64 ns

Table 4.3.: Subtasks of the AES algorithm model

The derived delays already indicate that NextState is the most complex subtask and will be the bottleneck. This is confirmed by Figure 4.6 which illustrates that ScheduleKey and AddRoundKey both have to wait a significant amount of time due to the long runtime of NextState. While the derived delays might have already been enough to support that assumption, this may not be the case for more complex algorithms or concurrency schemes.

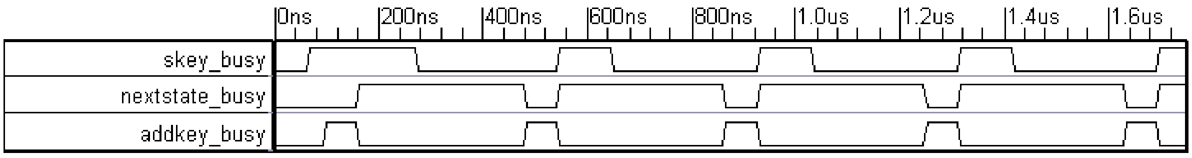


Figure 4.6.: Busy times of AES algorithm subtasks with original delays

The designer can already now evaluate what effect for instance a fast implementation of the NextState XOR operation would have. Assigning only 1 ns to this operation as compared to the original global value results in 112 ns total time for a NextState regular round. Figure 4.7 illustrates the improvement. The overall computation time drops from 3888 ns to 2592 ns. Therefore it is possible to already derive indicators about which part of the implementation should be particularly optimized. Please note however that this does not yet presume an implementation already but is only a qualitative statement that can be used as guideline.

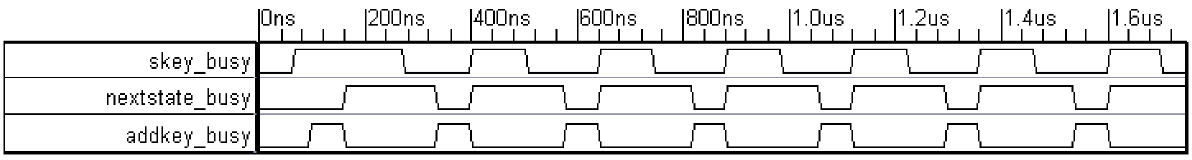


Figure 4.7.: Busy times of AES algorithm subtasks with altered delays

4.5.7. Concluding model implementation

NextState has been identified as the critical component. This hints to a possible subsequent implementation in hardware. In particular the S-Boxes that are critical for the performance of NextState would then be realized in one or more ROM memories. They should thus be moved to separate modules for an analysis of the best memory and communication structure. To ease the modeling refinement for such an analysis it is first carried out based on a loosely-timed model, and only then refined it to an approximately-timed model.

Each external S-Box is called through its own socket. This allows maximum flexibility with respect to the external bus and memory organization. If each S-Box transport call can be assumed to return instantly without competition for the target (which is the case assuming a replicated S1-Box), an initiator requires no dedicated thread for concurrent communication with several S-Boxes, but just call and wait in NextState/ScheduleKey is sufficient. If no such assumptions are made, one thread and 2 events per S-Box are necessary. The price for this increase in flexibility is a higher number of transport calls, events and context switches.

Since a lookup always comprises 4 bytes, they can be merged into one read transaction. The data buffer initially contains the 4 values that are to be substituted. Upon completion, it contains the substituted result. Lacking a concrete communication medium implementation to consider, this abstract form of communication is valid and complies with the standard.

The S-Box modules are implemented using a PEQ in the same style as described in Section 4.5.4. The initial loosely-timed model structure with instantaneous lookup is depicted in Figure 4.4. The final model structure with dedicated lookup modules is shown in Figure 4.8. All other intermediate steps described in this section had no effect on the block structure of the model. If the associated delays are set to zero, there is no difference from a model with instantaneous lookup.

Model breakdown

The following classes are used for implementing loosely-timed models.

- Generic payload

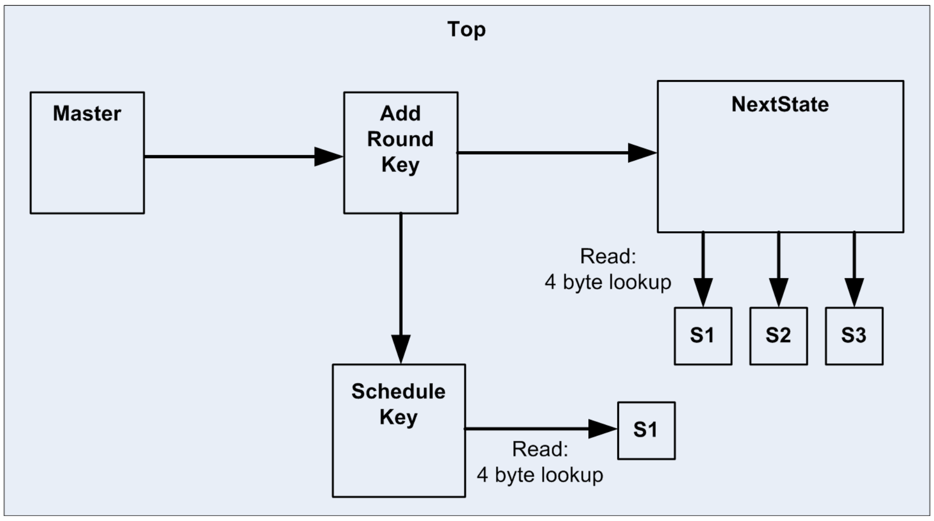


Figure 4.8.: System architecture with dedicated lookup modules

- SimpleLTSocket
- Non-blocking transport interface (forward and backward)
- Payload event queue

The model consists of leaf and non-leaf modules which implement the loosely-timed style with both annotation and sync.

4.6. Approximately-timed modeling

4.6.1. Migration and timing considerations

The approximately-timed modeling style enables a finer modeling of communication and computation by additional end phases for request and response. These become particularly interesting because of their usability for architectural and hardware/software tradeoff analysis. In contrast to the loosely-timed style, they resolve the need to impose certain restrictions on model behavior (i.e. always sync and callback, see Section 5.2).

Unlike the migration from untimed to loosely-timed, no interfaces have to be exchanged due to their syntactic equivalence to the ones used in the approximately-timed style. An AT initiator works with an LT target, assuming that the target replies to a BEGIN_REQ with TLM_COMPLETED and that the initiator is robust enough to handle the three implicit phase transitions being coincident. An LT initiator requires an inserted adapter module to filter the callbacks of an AT target that might employ phases which an LT initiator is not required to understand. The target SimpleLTSocket can therefore be reused in the AT model, while the initiator has to be marginally extended to handle the additional phases.

The concepts of request/response accept delay and response delay have been established in Section 2.3.2. Both delays in theory start at the begin of a request. In practice, the response delay (maybe even the request accept delay) of a target can only be known in advance if the module is a leaf node and not dependent on communication with another module. In our practical models, the response delay cannot start until after the request phase has ended (see Figure 2.8, 2.9 and 2.10). Therefore, the theoretical response delay as defined in the documentation is constituted by the request accept delay and the effectively implemented response delay which begin only after request acceptance (similar to Δt_1 and Δt_2 in Figure 2.7).

$$ResponseDelay_{theoretical} = RequestAcceptDelay + ResponseDelay_{implementation}$$

In the approximately-timed style, delays are accumulated less often then in the loosely-timed style. The focus is shifted to explicit synchronization with more frequent zero delay annotated transport calls. Based on [KHA05], an approximately-timed computation module may not include delays related to communication, rather these should be contained in dedicated communication modules.

Consequences for implementation

The practical considerations from Section 4.6.1 combined with Section 2.3.3 yield the following implementation guidelines for our AT models. See Section 2.3.3 for the description of intervals between events and the classification into software, non-pipelined hardware and pipelined hardware modules.

A software module accepts a request without annotation or callback, returning an implicit END_REQ along with BEGIN_RESP. No later point is possible because of a rule in [Ope07b], which is actually intended for LT/AT interfacing but is nevertheless valid. Only after the response has been accepted, it starts processing the next request. This assumes that the initiator first needs to receive/process the response before issuing the next request (Figure 2.8).

A non-pipelined hardware module also accepts a request without annotation or callback and returns END_REQ implicitly along with BEGIN_RESP, but already starts processing the next request before receiving the response acceptance. Care must be taken to delay the next response begin until the old response acceptance has been received (Figure 2.9).

A pipelined hardware module returns END_REQ either implicitly by annotation if it is a leaf module or by callback if the result is dependent on a nested transport call (which is unlikely though due to the nature of pipelining). The accept delay should be equal to the slowest stage. After the accept delay, the next request processing can be started (Figure 2.10).

Communication delays shall be introduced by dedicated communication modules except in the case of two communicating software modules. In this case, the function call / context switch overhead can be modeled by increased request accept delay (call) and response accept delay (return).

If a request/response buffer is desired, i.e. the module is not pipelined and a request should be accepted before the module can process it, the buffer should rather be placed in the communication module, because this aspect is part of communication rather than computation.

It is the responsibility of the initiator to delay the next request until it has received the request acceptance or alternatively in the responsibility of the communication module to delay a new request until a request acceptance for the last request has been returned by the computation module.

4.6.2. Model implementation

The master/AddRoundKey interface remains loosely-timed to stress the interoperability aspect between LT and AT. Based on the implementation guidelines of Section 4.6.1, we develop the models in the following way.

Computation modules

A computation module is implemented by two communication methods (responsible for handling begin and end of the response) and one worker method (leaf module) or thread (non-leaf module), combined with a *C++ standard template library* worker queue and a worker event. The transport call puts the transaction into the worker queue, instantly notifies the work event (after checking for a zero delay to detect undesired temporal decoupling) and returns TLM_ACCEPTED. Once the worker is idle it fetches the transaction from the queue and starts processing it. Only one transaction is processed at a time, others are queued. This concept matches the behavior of software or non-pipelined HW. For pipelined behavior, an additional request handling method, a different synchronization among the methods and additional queues would be required. In the following, only software or non-pipelined hardware is modeled.

END_REQ is delivered implicitly by the BEGIN_RESP callback. The worker notifies the begin response method with the response delay that was accumulated during its processing. Once a response acceptance is received (by either method, depending if the initiator annotates or calls back), the transaction is removed from the queue.

In case of a leaf module that is not dependent on communication with another module, the worker can be a method which notifies the response method with the processing time as offset. If additional communication or waits occur, the worker thread is implementing part of the delay by waits and the last part by a notification offset. See Appendix C.4 for illustrative code.

Communication modules

The bus interconnect module includes 2 modes: *single* and *multi* trans. Single means that the bus will accept the next transaction only after receiving an end response event from the old one. Multi means that this response is not necessary for the next request to be transmitted. Into both directions, the bus implements a size-dependent delay before passing the transaction on. Only one transaction into each direction can be processed by multi, a limitation which results from the 2-threaded design (request, response). For more flexibility, a more elaborate thread-structure would be required.

It has been assumed that regardless whether the interconnect module is possibly able to manage several pending transactions (as in the multi transaction mode), only one can actually be transmitted at any time. This means that an initiator or interconnect module monitors the status of the last delivered transaction and should not start a new one while the target is busy, which indicated by the lack of response in form of a transition to the END_REQ phase.

The coincidence of END_REQ with BEGIN_RESP (and thus the possibility that no distinct END_REQ is ever transmitted) caused complications. The possible requirement of completing a nested transaction would deadlock communication over the common single-transaction bus for the following reason. To start BEGIN_RESP of the enclosing transaction that is dependent on the nested transaction, this nested transaction would obviously need to be completed first. The nested transaction however cannot start before an END_REQ (explicit or implicit) of the enclosing one was received, since without seeing an END_REQ for the enclosing transaction, the single transaction bus remains busy and thus unavailable for the nested transaction. Thus, an interconnect model that is designed for only one transaction at a time (i.e. a simple ROM interface) would deadlock. Also, a non-leaf module employing an initiator SimpleATSocket should therefore not wait for END_REQ (and dismiss the possibly conincidence with BEGIN_RESP). The default behavior of the SimpleATSocket would need to be modified such that both EndRequestEvent and BeginResponseEvent are notified when BEGIN_RESP is received (see Section 3.4.2).

These concepts are applied to the *SimpleBus* of Section 3.4.2 to generate a modified version, the *aes_SimpleBus*.

4.6.3. Architectural exploration in practice

As described in Section 2.3.2, one purpose of the AT modeling style is to enable efficient and fast architectural exploration and hardware/software tradeoff analysis. We will now focus on this aspect.

Alternation between hardware and software behavior

To enable an easy simulation of several architectural configuration, every module can alternate its behavior between hardware and software. The associated configuration parameter determines whether potentially concurrent actions are executed in a way that corresponds to concurrent hardware or sequential software, for example whether three concurrent transactions are to be notified at the same time, followed by a wait for the conjunction of all three done events, or if the three notifications and waits are performed in an interleaved fashion (see Appendix C.5 for illustrative code). This concept is applied to potentially concurrent computation as well as communication. A prerequisite this alternation of behavior is the separation of each of these actions into distinct threads and methods.

The simulation of several hardware/software behavior configurations can be merged into a single run by changing the behavior configuration parameter in the top module between two successive encryption runs, similar to setting a new delay scheme as described in Section 4.5.5.

Model 1: Software solution

We first set a configuration that models a pure software implementation. In essence, it comprises the loosely-timed model refined to approximately-timed transport interfaces and enhanced by the mentioned facilities for a switch between hardware and software behavior. There are no communication nodes yet, all delays represent the busy time of processing modules and all modules call each other directly. Each lookup socket has its own dedicated lookup module (compare Figure 4.8).

All parameters are set to software behavior. Only one action, communication or computation, is performed at any time. The resulting delays model the computation and call/return (context switch) timing that software is known to exhibit[18]. The model block structure corresponds to the once depicted in Figure 4.8. Keep in mind that the delays for the external S-Box lookups can be set to zero to model a fast lookup as opposed to an external memory lookup.

The delay scheme for an assumed abstract software implementation has already been outlined in Section 4.5.6, Table 4.2. Now this scheme is applied to evaluate the practical implementation in a 1 GHz general purpose CPU, which yields a runtime of 7504 ns for the encryption of one packet. The fact that two similar models with equal delay assignments exhibit so different results can be explained by the fact that the model in

[18]Setting all parameters to hardware behavior would model a non-pipelined ASIC with point-to-point connections between the submodules.

Section 4.5 served the goal of evaluating a concurrency scheme and thus made massive use of concurrency, while the software model of this section models a software implementation and is sequential in nature. So while the two models somewhat resemble each other structurally, their respective purposes do not.

Model 2: Microcontroller with simple ROM interface

We now employ our bus component for the first time. All lookups are performed by transactions to one single memory module. The calls are transmitted over the shared bus, to which all lookup sockets are connected.

Since NextState has three lookup sockets that shall remain ignorant of the communication architecture, they are mapped to an interconnect component that serializes the calls and modifies the addresses for uniqueness within the memory that represents all three S-Boxes. ScheduleKey modifies the address of its only lookup-socket by itself. This is the only modification that is required to reflect the new model structure in computation modules, and could, if the model were not white-box accessible, be performed by another interconnect component. The only two initiators connected to the bus are therefore the one from the interconnect component and one from ScheduleKey. The only target connected is the memory module (depicted in Figure 4.9).

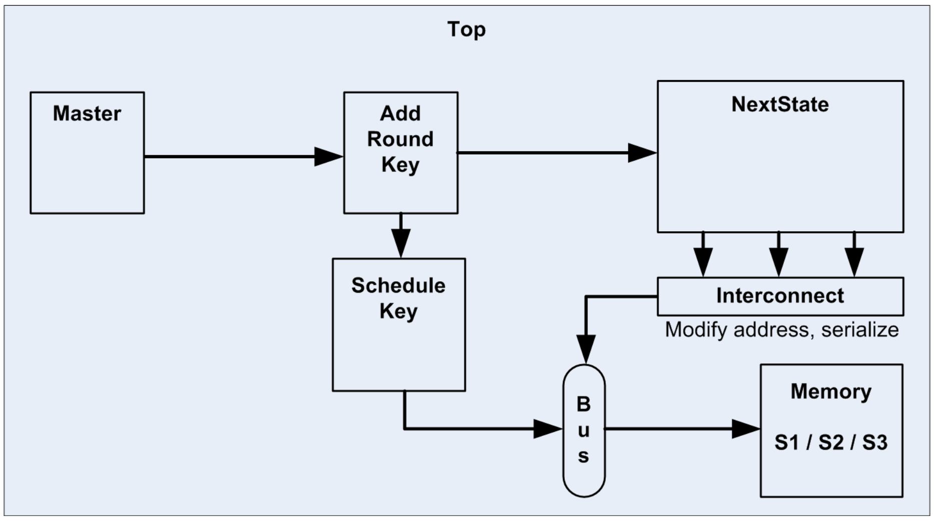


Figure 4.9.: System architecture with shared lookup module

All computation modules are switched to software to model sequential software executed on a microcontroller. Consequently, we keep the point-to-point connections among the computation modules. Since the memory has no potentially concurrent behavior, switching it from software to hardware has no actual effect.

Since only lookup transactions travel over the bus and have no nested dependency, the structure in principle works with all combinations of hardware/software switch settings as well as single/multi bus settings. Due to the usually simple nature of a ROM interface, single is preferable.

The delay scheme for an assumed abstract software implementation has already been outlined in Section 4.5.6, Table 4.2. In contrast to model 1 however which assumes a fast CPU, model 2 models a comparably slower (500 MHz) microcontroller with memory bus congestion as potential bottleneck. Therefore all delays related to computation are doubled. The lookup delay which is only partly related to the computation speed is changed to one I/O store operation plus memory delay and represents the lookup of one byte. A short delay of 1 ns per byte is added to a transaction when it traverses the assumed fast bus.

Operations	attributed delays
Write/read	load + store / store + load = 4 ns
XOR	2 operand loads + XOR + result store = 8 ns
Assign	store = 2 ns
Lookup CPU	start address load + offset load + indirect read + store = 8 ns
Lookup MC	store + memory delay = 2 ns + 4 ns = 6 ns
Bus transfer	1 ns per byte

Table 4.4.: Delays attributed to operations in the microcontroller model

Applying the microcontroller delay values to this model yields a runtime of 13056 ns for the encryption of one packet. This increase should be contrasted to the 15008 ns that the same computation would have required if it would have been performed by model 1 at 500 MHz. The assumed fast memory interface alleviated the impact of the operating frequency reduction.

Model 3: ASIC comprising a Network-on-Chip (NOC)

We now connect all sockets to the bus, and let all transactions travel across it. All modules are switched to hardware behavior to model an ASIC, the crossbar connection mimics a Network-On-Chip. This configuration only works with a multi trans bus setting due to the resulting nested transactions (see Section 4.6.2). A re-design of the bus would be necessary to bring it closer to a true network-on-chip behavior.

It should be particularly noted that the only change necessary to create this version was to re-map the modules, after they have been generally made fit for communication over a shared bus as in model 2. A reconfiguration to any communication structure can in principle be performed, as long as the modules use the same addresses and the bus uses the same address resolution to determine the transaction target. The resulting system structure is depicted in Figure 4.10.

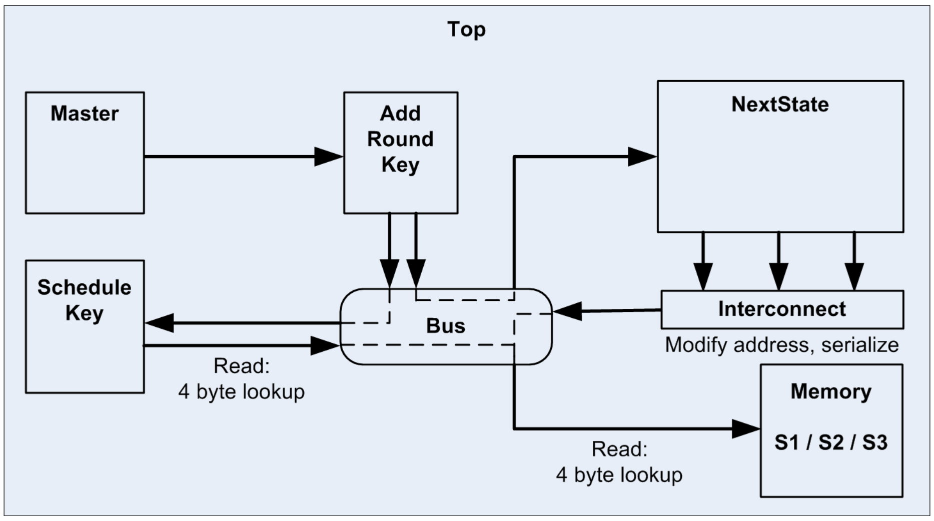


Figure 4.10.: System architecture with bus topology

The NOC is again assumed to run at 500 MHz and consist of ASICs who are optimized such that each operation requires only one clock cycle per byte. Communication between modules is assumed to be slower than the internal computation. A higher read-/write delay models the coordination overhead of the network infrastructure and a delay of 1 ns per byte is added to a transaction to model the assumed fast network transfer. This results in the delay scheme listed in Table 4.5.

The NOC employs maximum concurrency and encrypts a packet in 7488 ns, which is a significant decrease compared to the microcontroller or slow software model. Please note that these are rather exemplary and rough evaluations. More realistic analyses require a more profound knowledge of the considered hardware components like the CPU instruction set and memory architecture or the NOC communication architecture to leverage their full potential, which is out of the scope of this work.

Operations	attributed delays
Write/read	4 ns
XOR	2 ns
Assign	2 ns
Lookup	2 ns
Network	1 ns per byte

Table 4.5.: Delays attributed to operations in the NOC model

Model	simulated encryption time
Software 1 GHz	7504 ns
Software 500 MHz	15008 ns
Microcontroller 500 MHz	13056 ns ns
NOC 500 MHz	7488 ns

Table 4.6.: Results of the exemplary architectural analysis

Table 4.6 illustrates the results gathered during this exemplary architectural analysis.

Model breakdown

The following classes are used for implementation of approximately-timed models.

- Generic payload
- SimpleATSocket
- Non-blocking transport interface (forward and backward)
- Standard Template Library queue

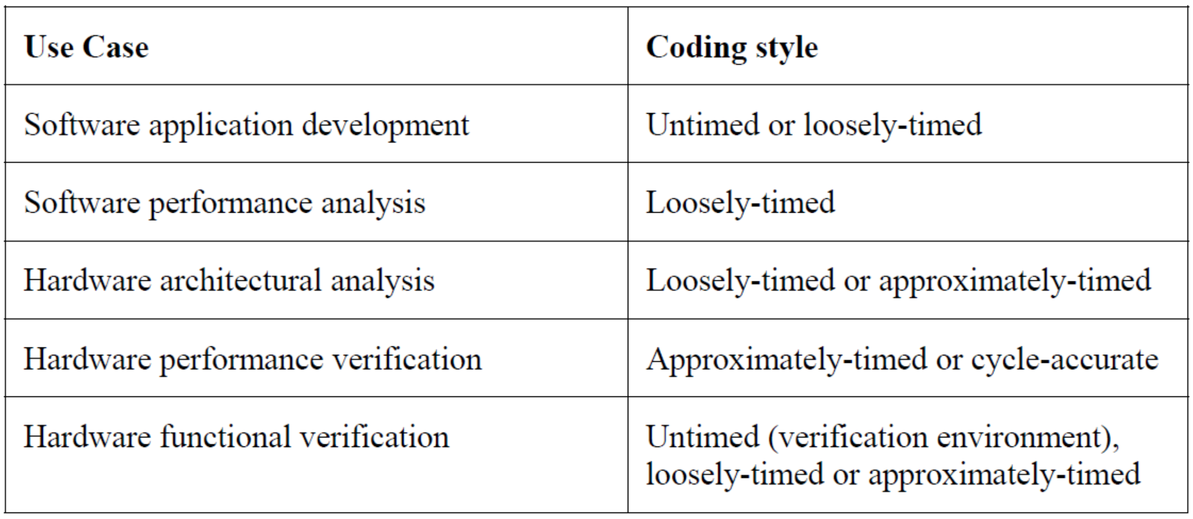
The models consist of leaf and non-leaf modules which implement the approximately-timed style with both annotation and backward path.

Appendix B constitutes a concluding index or the identifiers that have been used in the modeling process, to assist in the understanding of the developed code.

Chapter 5.

Results and discussion

Figure 5.1 shows the summary of use cases at which TLM is targeted and the modeling styles that are designed to meet them. This chapter will discuss the experiences made during the case study described in Chapter 4 and present facts and figures on the resulting models.



Use Case	**Coding style**
Software application development	Untimed or loosely-timed
Software performance analysis	Loosely-timed
Hardware architectural analysis	Loosely-timed or approximately-timed
Hardware performance verification	Approximately-timed or cycle-accurate
Hardware functional verification	Untimed (verification environment), loosely-timed or approximately-timed

Figure 5.1.: Mapping between TLM use cases and modeling styles [IEE03]

5.1. Application comprehension by algorithmic modeling

An algorithmic model is defined by Section 2.3.2 as an untimed model containing only one execution thread, consequently performing its function per instant calls that are not afflicted with timing. Such a model was designed and described in Section 4.4 and found to be a good tool for a better understanding of an application that is initially given as an abstract, possibly mathematical description. It is an appropriate first step in the design process. Particularly the resulting division into blocks and revealed data dependencies laid the basis for the subsequent timing afflicted modeling.

From an implementation point of view it was advantageous to create a functional golden model whose core functions could be embedded in subsequent timed models, without requiring changes to the functions themselves and therefore eliminating a potential point of error.

The well defined and separated interface to the AES system by its master module facilitate its use as part of a greater development project involving AES just as another subsystem, for the use cases depicted in Figure 5.1. These include the development of software based on an untimed system model that offers its services at such interfaces, as well as hardware functional verification due to the correctness of the model.

It was in particular this last use case for which the untimed AES model was used in this work, namely to identify errors introduced by the refinement process.

5.2. Delay modeling scopes of LT and AT

The different timing modeling capabilities of LT and AT were of particular interest. The theory has been outlined in Chapter 2 and 3. The practical implication that showed during the modeling phase is that it is not natively possible in LT to differentiate between delays that are produced by communication and delays that are produced by computation processes. This has been tried by incorporating facilities for hardware/software tradeoff analysis as described in Section 4.5.5, using distinct busy flags to indicate the current status of a module, what it is waiting for or occupied with, to enable such analysis while still using the speed of LT and come to conclusions about which module would be best implemented as hardware or software.

To determine the different types of business however it is required to determine their duration or at least their begin and end. It was found that this was not generally possible without imposing restrictions on the usage and interpretation of the delays annotated to transport calls. For instance, the delay annotated by a target to a backward path callback had to be non-zero (in contrast to implementing the delay in the target and performing the callback with a zero delay) to enable the initiator that was waiting for the callback to differentiate between the time that is has been waiting for the callback to arrive and the communication time required to receive that callback. This distinction would have been desirable to determine how long a module has to wait for a callback due to bus congestion or other causes. Without that non-zero annotation though, the actual communication busy time gets merged into the preceding waiting time because of missing or simultaneous event triggers in the initiator SimpleSocket. The fundamental contradiction is that for such a differentiated analysis, a request or response would somehow have to be divided into partial delays that together constitute the whole request and response delay. By definition though, request and response are singular events in the LT modeling style and any alternation to this paradigm results in imposing restriction on the interpretation of LT timing values.

The conclusion is that with LT only a coarse architectural and tradeoff analysis is possible that attributes delays to certain functions, but without the capability of differentiating them more exactly. However the need for additional events within the initiator for modeling these differentiated delays already hints to the AT style which offers semantically comparable additional phase end events natively.

The demand of [KHA05] to separate communication from computation delays by separate modules in contrast to mixing them in the same module consequently raises the question whether communication delays can be correctly modeled at all without an interconnect component. The modeling conducted in this work shows that it is possible on an abstract level, but even there not advisable, since it would complicate the individual module and impose restraints on the usage of implicit and explicit delay values as explained above. One exception is the modeling of software behavior. While it is natural to model busy times by busy flags to ease the analysis of effects that internal changes have on the externally visible timing, they should not influence or create assumptions on the way that the timing is implemented. Combined with that LT is targeted at only coarse architectural modeling or more software related use cases while AT is advocated as a more hardware related style, AT should be used when (hardware) communication structures are to be evaluated. Since the delay definitions related to AT are only concerned with the overall throughput and latency of the respective model without any differentiations, the conclusion is that the recommendations of [KHA05] are coherent.

5.3. Adequacy for HW/SW partitioning

Among the goals set for this work was to answer the question to what extent the problem of hardware/software partitioning can be addressed by the described TLM tools. The results comprise the following key points . . .

- SystemC or TLM cannot natively tackle the task of concurrency analysis, which is an important prerequisite for deriving a hardware/software partitioning.
- The use of an untimed model as initial step does reveal the insights mentioned in 5.1 through the analyses necessary for the model implementation, not by running the model.
- The loosely-timed modeling style can be used for a coarse analysis and the derivation of qualitative statements about a proposed partitioning. It can be employed for a quick simulation of a variety of delay schemes.
- For a more detailed performance analysis of a chosen scheme, the loosely-timed style does not offer sufficient capabilities. The attempt to introduce them into the model at the LT stage resulted in potentially problematic restrictions that infringe the interoperability with other models and rather resembled a step towards the approximately-timed style.
- The mechanisms for switching between hardware and software behavior are not influenced by the migration from LT to AT. When observing the recommendations of [KHA05], AT enables the distinct modeling of communication versus computation by correct usage of the associated request and response delays.

- Consequently, the conclusion can be drawn that AT is an appropriate style for evaluating a certain hardware/software architecture.

LT and AT are both useful for architectural modeling, depending on the required accuracy. While LT can be used for more qualitative analyses, AT is better suited for the evaluation of more refined structures. Since time accuracy is important for performance evaluations, AT or even cycle-accurate models are suitable for this use case. More diversification in modeling must on the other hand be paid for by an increased number of events and context switched, as illustrated in Section 5.5. These conclusions match the use cases depicted in Figure 5.1.

5.4. Software development focus of LT

SystemC and TLM have been advertised as new technology for generating virtual prototypes of (hardware) architectures and to enable early software development based on these prototypes while the real hardware is yet unavailable. Especially the loosely-timed modeling style has been targeted at related use cases, primarily due to its ability to use much fewer events and context switches than an approximately-timed or RTL model, in particular when TD can be used in its full extent. Section 5.5 illustrates the decrease of events when pure LT is used in contrast to more detailed styles.

From a more qualitative perspective, LT can, as already established, not differentiate well between the causes of delays. This statement is targeted at the internal workings of a system model. In the case of software development or performance evaluation, this software employs only the interfaces of the underlying LT model that it is developed for. While it might be important for model itself and the derivation of the correct interface timing, the distinction between different sources of delays is not so much relevant for the external software when its performance is evaluated.

What on the other hand is important is the increase of simulation speed, that enables e.g. the booting of an operating system within the model in reasonable time. Also, since an LT model comprises less details than an AT model, it might be available before the AT model and enable an earlier start of software development.

It is also possible to abstract an LT model from an AT or RTL model to use it for more rapid development than with the original one. This latter case seems to be a more attractive field for companies these days, since it enables them to add more value to their IPs by opening them to this use case. One example is a fast model of a CPU IP, or an ISS that is embedded in the overall system model by including it in a LT module.

In contrast to a design flow that maps the algorithm to already existing LT or AT platform models, the flow in this work employs LT models early for coarse hardware/software tradeoff analysis, before the selection or implementation of AT models. The goal is to derive qualitative data on which the selection / implementation can then be based. This is not a contradiction but an intermediate phase that, based on the context of the project, may very well be advisable. The experiences made in the modeling process indicate that LT is appropriate for these intended use cases.

5.5. Performance analysis

This section illustrates the performance of the various modeling styles and models.

5.5.1. Setup for data gathering

The figures in table 5.1 refer to the simulated encryption of a 1 Mbyte data stream (8192 packets à 128 bit). The stream is triggered by a single notification from the top module to the master, and the simulation is stopped by the master after reading the last cipher text packet.

Time values are retrieved by the following function.

```
#include <sys/time.h>
struct timeval tv;
int getTime(void)
{
    gettimeofday(&tv,NULL);
    return (int)((tv.tv_sec*1000) + (tv.tv_usec/1000));
}
```

The current time value is recorded immediately before and after the simulation start. The measurement discards time components that have only constant influence on the execution time, like the loading of the binary or the instantiation of the model object. Tracing has been disabled.

```
int starttime = getTime();
sc_core::sc_start();
int endtime = getTime();
cout << endtime - starttime  << endl;
```

The models are compiled with *GCC 3.4.6 20060404 (Red Hat 3.4.6-3)* along with the arguments *-O3 -march=nocona*. The employed operating system is a *Redhat Linux*, running on a *Dual core Intel(R) Xeon(TM)* 64-bit CPU with 3.20GHz.

The resulting runtime value is the average of 100 model executions. It was noted that the compiler arguments not only significantly reduced the absolute runtime of the models but also the variance among the individual runs.

5.5.2. Results and interpretation

Model	threads/methods[19]	events[20]	context switches[21]	runtime [ms]
Software	1/0	0 (0/0/0)	0	26
Algorithmic	1/0	0 (0/0/0)	1	24
LT seq. TD	1/0	3 (0/3/0)	24577	83
LT seq.	2/0	6 (3/3/0)	385026	200
LT conc.	8/0	18 (15/3/0)	1728522	801
LT conc. flags	16/0	36 (29/7/0)	8372249	3871
AT software	12/7	41 (27/14/0)	8060942	2786
AT NOC	14/7	55 (27/22/6)	11591696	6812

Table 5.1.: Facts'n figures about selected models

As can be seen in table 5.1, the runtimes of software and the algorithmic model match closely. This seems reasonable because the algorithmic model performs only one context switch followed by the exact same computation like the software implementation, with only the transport calls as additional overhead. However the fact that the runtime of the algorithmic model is smaller than the software model, and not a little larger as would be expected, is inconclusive. It can only be presumed that the compiler performs a better optimization of the algorithmic model compared to the software

[19]The number of software threads is 1. For models, the additional thread that only triggers the simulation start is not considered since it is not actively participating in the remaining model execution.

[20]The number of events does not include the start simulation event used by the top module. The format is total(module/socket/bus) and includes events associated with PEQs.

[21]The number of context switches covers wait-calls for events and delays as well as the execution of methods in modules, sockets and buses.

implementation. The result still confirms though that the execution times are roughly the same magnitude.

The mere translation from algorithmic (untimed) to loosely-timed with the associated introduction of events and synchronization instantly increases the number of context switches by several orders of magnitude. This underlines the fact that untimed models should be applied wherever time is of no relevance.

The removal of temporal decoupling again increases the number of context switches by a factor of more than 15 and the runtime by a factor of about 2.5, illustrating the power of this optional loosely-timed technique.

As mentioned in Section 4.5.2, the initial two loosely-timed models still exhibit a sequential software behavior like the algorithmic model. A maximum parallelization results in yet another increase of context switches due to the additional events and threads required to model concurrent computation (see the rise of threads and events in the module category).

A comparison between the increase of context switches and the increase of runtime indicates that they rise by approximately the same factor as long as the modeling style is not significantly changed. This is illustrated by the aligned increases of the not temporally decoupled loosely-timed models.

The decrease of runtime from the last loosely-timed to the first approximately-timed model is unexpected but could be related to the reduction of threads by a forth, which are more costly than methods. We can also see how the performance of the loosely-timed model with busy flags enters the dimension of the approximately-timed models, which supports the conclusion from Section 5.3 that the additional overhead required for architectural exploration negates the performance gain of the modeling style.

The inconclusive performance relation among the approximately-timed models gives rise to the supposition that the runtime does not always increase in simple relation with the number of context switches and processes, but that also structural changes within the model are relevant. Keep in mind that the loosely-timed models all had the same basic shape that was only extended by additional lookup modules, while the block diagrams of the approximately-timed models changed significantly between the software and the NOC model. While the compiler optimizations reduced the variance of runtime results, they could contribute as well to a disguise of performance relations.

In general the performance figures indicate the significant differences that the various modeling styles exhibit, and illustrate the importance of choosing the right modeling style for the intended use case.

Chapter 6.

Conclusion and outlook

This thesis has introduced the idea of TLM-based modeling as a tool for *Electronic System Level Design*. It outlined the language SystemC upon which the TLM methodology is based and focused on translating its theoretical concepts and guidelines into practical implementation approaches. Where the OSCI standard was not a sufficient basis, the terms and definitions of the OCP-IP were applied or adapted. This led to a set of implementation guidelines that were either extracted from documentation or abstracted from included example designs, and that were the basis for the implementation of an exemplary case study.

This case study was carried out based on a design flow which delivered insights required to evaluate the methodology's potential. The discussion was focused on the usefulness of the different modeling styles and their applicability to certain use cases. The most prominent among them were hardware/software partitioning and tradeoff analysis as well as architectural exploration. The facilities necessary to perform such analyses were established and implemented, and the applicability of the loosely- and approximately-timed modeling styles was discussed. It was concluded that the AT style is the first choice for the kind of analyses in focus. While the LT style shows potential for them as well, its main focus of either abstract or software related use cases was discussed and illustrated. A short architectural exploration of the case study concluded the design flow and the focus of this thesis.

TLM-based *Electronic System Level Design* is a very broad field of application and several ways to extend this work are conceivable. A next step on the employed design flow would be to apply the results of the architectural exploration to implement a model that consists of IP cores rather than abstract models. Alternatively refinement could continue down to the RTL level or serve as input to a commercial behavioral synthesis tool. Furthermore the model could be centered around an extensible processor IP, based on which parts of the application can be implemented in software.

A new case study could contrast this design flow that has a theoretical focus to the one outlined in [KHA05]. After initial algorithmic modeling it progresses to an AT-like hardware modeling phase and then abstracts these hardware models to allows prototype based software engineering, while additional refinement results in a hardware implementation that can then run the developed software.

This variety of links to follow illustrates the potential that *Electronic System Level Design* and *Transaction Level Modeling* offer to evolve today's embedded system design flows.

Appendix A.

Acronyms

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
AT	Approximately-timed
DMI	Direct Memory Interface
GP	Generic Payload
HDL	Hardware Description Language
ISS	Instruction Set Simulator
LT	Loosely-timed
MMB	Memory-Mapped Bus
NOC	Network On Chip
OCP-IP	Open Core Protocol International Partnership
OOP	Object Oriented Programming
OSCI	Open SystemC Initiative
PEQ	Payload Event Queue
QK	Quantum Keeper
RAM	Random Access Memory
ROM	Read-Only Memory

RTL Register Transfer Level

STL Standard Template Library

TLM Transaction Level Modeling

UML Unified Modeling Language

UT Untimed

VHDL Very High Speed Integrated Circuit Hardware Description Language

VSP Virtual System Prototype

Appendix B.

Identifiers

SysC function declared and defined by the SystemC language

TLM UT function declared by the TLM standard for untimed modeling

TLM LT/AT function declared by the TLM standard for loosely / approximately timed modeling

Socket Impl function declared and defined by the individual socket implementation

App function declared and defined by the individual user application implementation

TLM + Socket Impl function/class defined by the *SimpleSocket* implementation, having the same function signature(s) but a different name as a function declared by the TLM standard, to implement this function through a function pointer

TLM + App function/class defined by the individual user application implementation, having the same function signature(s) but a different name as a function declared by the TLM standard, to implement this function through a function pointer

Declared means the declaration of the name and signature of a function, *defined* means the actual implementation of that declaration.

Fpoint = Functon Pointer

Function name	SysC	TLM UT	TLM LT/AT	Socket Impl	App	Category
sc_core::sc_module	X					SytemC basics
sc_core::sc_module_name	X					
sc_core::sc_event	X					
sc_core::sc_time	X					
SC_HAS_PROCESS(. . .)	X					SystemC macros
SC_THREAD(. . .)	X					
SC_METHOD(. . .)	X					
wait(. . .)	X					SystemC sync
notify(. . .)	X					
sc_trace(. . .)	X					SystemC tracing
tlm_fw_b_transport_if		X				TLM interfaces
tlm_bw_b_transport_if		X				
tlm_fw_nb_transport_if			X			
tlm_bw_nb_transport_if			X			
b_transport(. . .)		X				TLM functions
nb_transport(. . .)			X			
transport_dbg(. . .)		X	X			
get_direct_mem_ptr(. . .)		X	X			
invalidate_d._m._p.(. . .)		X	X			
tlm_generic_payload		X	X			TLM datatypes
tlm_generic_payload_types		X	X			
tlm_debug_payload		X	X			
tlm_dmi		X	X			
tlm_dmi_mode		X	X			
tlm_phase			X			
tlm_sync_enum			X			
tlm_initiator_socket		X	X			TLM sockets
tlm_target_socket		X	X			
MyPEQ			X			TLM queue
SimpleUTInitiatorSocket		X		X		SimpleSockets
SimpleUTTargetSocket		X		X		
SimpleLTTargetSocket			X	X		

SimpleLTInitiatorSocket			X	X		
registerBTransport(. . .)				X		Fpoint reg.
registerNBTransport(. . .)				X		
registerInvalidateDMI(. . .)				X		
registerDebugTransport(. . .)				X		
registerDMI(. . .)				X		
REGISTER_XXX				X		Fpoint macro
setInvalidateDMIPtr(. . .)				X		Fpoint setters
setTransportPtr(. . .)				X		
setTransportDebugPtr(. . .)				X		
setGetDMIPtr(. . .)				X		
simple_socket_utils				X		SocketID
set***UserId(. . .)				X		
getEndEvent()				X		Event getter
masterBTransport(. . .)		X			X	App. transport
addkeyBTransport(. . .)		X			X	
addkeyNBTransport(. . .)			X		X	
masterNBTransport(. . .)			X		X	
nextstateNBTransport(. . .)			X		X	
skeyNBTransport(. . .)			X		X	
Initiate ()					X	App. threads
Encrypt()					X	
NextStateThread()					X	
NextStateFinalThread()					X	
ScheduleKeyThread()					X	
WorkerThread()					X	
WorkerMethod()					X	
ManageBusyFlagThread()					X	
S1/2/3ComThread()					X	
SKeyComThread()					X	
NextStateComThread()					X	
BeginResponseMethod()					X	AT comm. only
EndResponseMethod()					X	
NextState(. . .)					X	App. core

NextStateFinal(. . .)					X	
ScheduleKey(. . .)					X	
Lookup(. . .)					X	
PrintKey(. . .)					X	App. output
PrintNextState(. . .)					X	
PrintResultState(. . .)					X	
PrintArray(. . .)					X	
aes_payload					X	App. data

Table B.1.: Index of identifiers

Appendix C.

Codelisting

C.1. SimpleSocket transport function pointer

SimpleLTTargetSocket forward transport function pointer management - condensed

```
class SimpleLTTargetSocket : public tlm::tlm_target_socket< ... >
{
  ...

  //======================================================================
  //  REGISTER_XXX methods (registering function pointers)
  //======================================================================
  template <typename MODULE>
  void registerNBTransport(MODULE* mod,
  sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::
      sc_time&), int id)
  {
    mProcess.setTransportPtr(mod, static_cast<typename Process::
        TransportPtr>(cb));
    mProcess.setTransportUserId(id);
  }
  ...

  //======================================================================
  //  Subclass "Process" (executing function pointers or default behavior)
  //======================================================================
  private:
  class Process : public tlm::tlm_fw_nb_transport_if<TYPES>
  {
```

```
//======================================================================
//  Type definitions
//======================================================================
public:
  typedef sync_enum_type (sc_core::sc_module::*TransportPtr)
  (transaction_type&, tlm::tlm_phase&, sc_core::sc_time&);
  ...
//======================================================================
//  Constructors and destructor
//======================================================================
  Process(const std::string& name) :
    ...
  {
  }
//======================================================================
//  Setter functions for UserID and function pointers
//======================================================================
  void setTransportUserId(int id) { mTransportUserId = id; }
  void setTransportDebugUserId(int id) { mTransportDebugUserId = id; }
  ...
  void setTransportPtr(sc_core::sc_module* mod, TransportPtr p)
  {
    if (mTransportPtr)
    {
      std::cerr << "non-blocking forward call already registered" << std
          ::endl;
    }
    else
    {
      assert(!mMod || mMod == mod);
      mMod = mod;
      mTransportPtr = p;
    }
  }

//======================================================================
//  Target non-blocking forward interface
//  (executing function pointers or default behavior)
//======================================================================
  /// Target non-blocking transport forward interface
  sync_enum_type nb_transport(transaction_type& trans,phase_type& phase,
                              sc_core::sc_time& t)
    {
      if (mTransportPtr)
      {
        // forward call to function pointer
        assert(mMod);
        simple_socket_utils::simple_socket_user::instance().set_user_id(
            mTransportUserId);
        return (mMod->*mTransportPtr)(trans, phase, t);
      }
      else
      {
        std::cerr << "no non-blocking transport registered" << std::endl;
        assert(0); exit(1);
      }
    }
    ...

  //======================================================================
  //  Member variables (Process)
  //======================================================================
    private:
      const std::string mName;
      sc_core::sc_module* mMod;    ///< pointer to module holding functions
      TransportPtr mTransportPtr; ///< pointers to module functions
      int mTransportUserId;       ///< user IDs
      int mTransportDebugUserId;
      ...
    };
    ...
};
```

Register macros used in models (simple_socket_utils)

```
#define REGISTER_NBTRANSPORT(socket, process) \
  socket.registerNBTransport(this, &SC_CURRENT_USER_MODULE::process, 0)
#define REGISTER_NBTRANSPORT_USER(socket, process, id) \
  socket.registerNBTransport(this, &SC_CURRENT_USER_MODULE::process, id)
```

C.2. Event (untimed) model vs. call (algorithmic)

AddKey constructor, transport and encrypt function (event) - condensed

```
//==========================================================================
// Constructor and destructor
//==========================================================================
ut_aes_addkey::ut_aes_addkey (                ///< constructor
        sc_core::sc_module_name  module_name   ///< module name
    ) :                                        ///< initializations
     ...
   {
    // Register target forward interface functions to replace default
    // Transport default:     error
    REGISTER_BTRANSPORT(m_master_socket, masterBTransport);
    SC_THREAD(Encrypt);
   }
...
//==========================================================================
// Method performing communication, preparing and initiating computation
//==========================================================================
void ut_aes_addkey::masterBTransport(transaction_type& trans)
{
  ...
  switch (command)
  {
    default:
    {
      break;  ///< error
    }
    case tlm::TLM_WRITE_COMMAND:
    {
      if (address == KEY_INIT_ADDRESS)
      {
        ...
        m_key_initialized = true;
      }
      else if (address == STATE_INIT_ADDRESS)
      {
        ...
        m_state_initialized = true;
      }
      else
      {
        ...  ///< error
      }
      // Start computation once initialized
      if ((m_state_initialized == true) && (m_key_initialized == true))
      {
          m_StartEncryptEvent.notify(); //< event notification
      }
      break;
    }
    case tlm::TLM_READ_COMMAND:
    {
      if (address == CYPHER_READ_ADDRESS)
      {
        ///< data ready if bool is already true or wait returns
        if (m_encrypt_done == false)
          wait(m_EncryptDoneEvent);
        ...
        m_encrypt_done = false;
      }
      else
      {
      ...  ///< error
      }
      break;
    }
  }
  // Set parameters to indicate all is well
  trans.set_response_status(tlm::TLM_OK_RESPONSE);
}
//==========================================================================
// Method performing computation
//==========================================================================
void ut_aes_addkey::Encrypt()
{
  ...
  while(1)
  {
    /* Wait for wakeup call from transport procedure */
    wait(m_StartEncryptEvent);
    ...  ///< Encryption
    /* Reset control variables for possible next encryption */
    m_encrypt_done      = true;
    m_state_initialized = false;
    m_key_initialized   = false;
    /* Notify waiting transport procedure that encrypted data is ready */
    m_EncryptDoneEvent.notify();
  }
}
```

AddKey constructor, transport and encrypt function (call) - condensed

```
//==========================================================================
// Constructor and destructor
//==========================================================================
ut_aes_addkey::ut_aes_addkey (                    ///< constructor
        sc_core::sc_module_name  module_name      ///< module name
    ) :                                           ///< initializations
      sc_module           (module_name)
      , ...
   {
    // Register target forward interface functions to replace default
    // Transport default:     error
    REGISTER_BTRANSPORT(m_master_socket, masterBTransport);
   }
...
//==========================================================================
// Method performing communication, preparing and initiating computation
//==========================================================================
void ut_aes_addkey::masterBTransport(transaction_type& trans)
{
  ...
  switch (command)
  {
    default:
    {
      break;  ///< error
    }
    case tlm::TLM_WRITE_COMMAND:
    {
      if (address == KEY_INIT_ADDRESS)
      {
        ...
        m_key_initialized = true;
      }
      else if (address == STATE_INIT_ADDRESS)
      {
        ...
        m_state_initialized = true;
      }
      else
      {
        ...  ///< error
      }
      // Start computation once initialized
      if ((m_state_initialized == true) && (m_key_initialized == true))
      {
          Encrypt(); ///< function call
      }
      break;
    }
    case tlm::TLM_READ_COMMAND:
    {
      if (address == CYPHER_READ_ADDRESS)
      {
        ...  ///< perform read
        m_encrypt_done = false;
      }
      else
      {
      ...  ///< error
      }
      break;
    }
  }
  // Set parameters to indicate all is well
  trans.set_response_status(tlm::TLM_OK_RESPONSE);
}
//===========================================================================
// Method performing computation
//===========================================================================
void ut_aes_addkey::Encrypt()
{
    ...  ///< Encryption
    /* Reset control variables for possible next encryption */
    m_encrypt_done      = true;
    m_state_initialized = false;
    m_key_initialized   = false;
}
```

C.3. Loosely-timed target busy flag management

Busy flag management in lt_aes_lookup - condensed

```
//==========================================================================
// Constructors
//==========================================================================
  lt_aes_lookup::lt_aes_lookup ( ... ) : ///< initializations
      ...
   {
    REGISTER_NBTRANSPORT(m_lookup_socket, lookupNBTransport);
    SC_THREAD(ManageBusyFlagThread);
    SC_THREAD(WorkerThread);
   }
//==========================================================================
//  Target non-blocking forward interface
//  (m_lookup_socket)
//==========================================================================
/// Target non-blocking transport forward interface
/// performing the communication, preparing and initiating computation
lt_aes_lookup::sync_enum_type       ///< result
lt_aes_lookup::lookupNBTransport(   ///< nb_transport
  transaction_type&          trans, ///< transaction
  phase_type&                phase, ///< transaction phase
  sc_core::sc_time&          time   ///< elapsed time
  )
{
    // Pass tranction and arrival time to PEQ for WorkerThread
    m_TransQueue.notify(trans, time);
    return tlm::TLM_ACCEPTED;
}
//==========================================================================
//  Internal module functions
//==========================================================================
/// Thread setting the busy-flag, calling wait for the time of processing,
      and resetting the flag when done
void lt_aes_lookup::ManageBusyFlagThread(void)
{
    while(true)
    {
      wait(m_StartBeingBusyEvent);
      cout << this->name() << " is now busy for " << (m_BusyComTime +
          m_BusyProcTime) << endl;
      m_isBusy     = true;
      m_isBusyCom  = true;
      m_isBusyProc = false;
      wait(m_BusyComTime);
      m_isBusy     = true;
      m_isBusyCom  = false;
      m_isBusyProc = true;
      wait(m_BusyProcTime);
      m_isBusy     = false;
      m_isBusyCom  = false;
      m_isBusyProc = false;
      cout << sc_core::sc_time_stamp() << " - " << this->name() << " is
          now idle" << endl << endl;
      m_StopBeingBusyEvent.notify();
    }
}
/// Thread performing computation and transaction handling
void lt_aes_lookup::WorkerThread(void)
{
  while(true)
  {
    wait(m_TransQueue.getEvent());
    bool done = false;
    while (done == false)
    {
      transaction_type* trans_p = m_TransQueue.getNextTransaction();
      if (trans_p == 0)
      {
        done = true;
        break;
      }
      sc_core::sc_time CallbackTime  = sc_core::SC_ZERO_TIME;
      m_BusyComTime  = sc_core::SC_ZERO_TIME;
      m_BusyProcTime = sc_core::SC_ZERO_TIME;
      ...
      switch (command)
      {
```

```
    default:
    {
      ... ///< error
      break;
    }
    case tlm::TLM_WRITE_COMMAND:
    {
      ... ///< error
      break;
    }
    case tlm::TLM_READ_COMMAND: ///< read request - return the result
    {
      if (m_isBusy == true) ///< busy flag is true, wait until last
           processing is done to read consistent memory
      {
        wait(m_StopBeingBusyEvent);
      }
      unsigned int address1 =  address        % 256;
      unsigned int address2 = (address >>  8) % 256;
      unsigned int address3 = (address >> 16) % 256;
      unsigned int address4 = (address >> 24) % 256;
      switch (m_LookupType)
      {
        case(t_S1):
          data_p[0] = S1[address1];
          data_p[1] = S1[address2];
          data_p[2] = S1[address3];
          data_p[3] = S1[address4];

          m_BusyComTime  += global_delay_sbox_read;
          m_BusyProcTime += global_delay_sbox_lookup;
          CallbackTime   += global_delay_sbox_read +
               global_delay_sbox_lookup;
          break;
        case(t_S2):
          ...
          break;
        case(t_S3):
          ...
          break;
        case(t_RCON):
          ...
```

```
              break;
          }
          m_StartBeingBusyEvent.notify();
          break;
        }
     }
     // Set parameters to indicate all is well at callback
     trans_p->set_response_status(tlm::TLM_OK_RESPONSE);
     tlm::tlm_phase phase = tlm::BEGIN_RESP;
     (void)m_lookup_socket->nb_transport(*trans_p,phase,CallbackTime);
  }
}
```

C.4. Approximately-timed target example

AT lookup memory model as non-pipelined hardware (at_aes_memory.h) - complete

```
#include "SimpleSocket/simple_at_target_socket.h"
#include "utils/MyPEQ.h"
#include "aes_lookup.h"
#include "aes_constants.h"
#include <stdint.h>
#include <queue>

class at_aes_memory
  : public sc_core::sc_module                   ///< module base clase
{
  SC_HAS_PROCESS(at_aes_memory);
  //=====================================================================
  //  Type definitions
  //=====================================================================
  public:
  typedef tlm::tlm_generic_payload                        transaction_type;
  typedef tlm::tlm_dmi_mode                               dmi_mode_type;
  typedef tlm::tlm_phase                                  phase_type;
  typedef tlm::tlm_sync_enum                              sync_enum_type;
  typedef SimpleATTargetSocket<32,tlm::tlm_generic_payload_types>
      target_socket_type;
  //=====================================================================
  //  Constructors and destructor
  //=====================================================================
  public:
```

```
  at_aes_memory                                     ///< constructor
  (   sc_core::sc_module_name  module_name          ///< module name
  );
  ~at_aes_memory(void);                             ///< destructor
  private:
  at_aes_memory (void);                    ///< disabled default constructor
  //==============================================================================
  //  Target non-blocking forward interface
  //  (m_lookup_socket)
  //==============================================================================
  public:
  /// Target non-blocking transport forward interface
  sync_enum_type lookupNBTransport(                  ///< nb_transport
      transaction_type&         trans,               ///< transaction
      phase_type&               phase,               ///< transaction phase
      sc_core::sc_time&         time);               ///< elapsed time
  //==============================================================================
  //  Internal module functions
  //==============================================================================
  private:
  /// Thread fetching new request, computing and scheduling the response
  void WorkerMethod(void);
  /// Method executing callback to the Initiator to start response phase
  void BeginResponseMethod(void);
  /// Method removing the finished transaction and starting a new response
        phase for the next transaction
  void EndResponseMethod(void);
  //==============================================================================
  //  Member variables
  //==============================================================================
  public:
  target_socket_type     m_lookup_socket;           ///< target socket
  private:
  //std::queue<transaction_type*> m_RequestQueue;
  //std::queue<transaction_type*> m_ResponseQueue;
  std::queue<transaction_type*> m_WorkQueue;
  sc_core::sc_time    m_ResponseDelay;
  sc_core::sc_event   m_StartWorkEvent;
  //sc_core::sc_event   m_EndRequestEvent;
  sc_core::sc_event   m_BeginResponseEvent;
  sc_core::sc_event   m_EndResponseEvent;
}; // end class at_aes_memory
```

memory model as non-pipelined hardware (at_aes_memory.cpp) - complete

```
#include "tlm.h"                                   ///< TLM headers
#include "at_aes_memory.h"                         ///< our class header
using namespace  std;
//==========================================================================
// Constructors and destructor
//==========================================================================
at_aes_memory::at_aes_memory (                     ///< constructor
        sc_core::sc_module_name  module_name       ///< module name
    ) :                                            ///< initializations
      sc_module       (module_name)
    , m_lookup_socket ("m_lookup_socket")
    , m_ResponseDelay (sc_core::SC_ZERO_TIME)
   {
    //======================================================================
    // Register target forward interface functions to replace default
    // Transport default:     error
    // Debug default:         no support (return false)
    // Direct Memory default: no support (return false)
    //======================================================================
    REGISTER_NBTRANSPORT(m_lookup_socket, lookupNBTransport);
    SC_METHOD(WorkerMethod);
    sensitive << m_StartWorkEvent;
    dont_initialize();
    SC_METHOD(BeginResponseMethod)
    sensitive << m_BeginResponseEvent;
    dont_initialize();
    SC_METHOD(EndResponseMethod)
    sensitive << m_EndResponseEvent;
    dont_initialize();
   }

at_aes_memory::~at_aes_memory(void)                    ///< destructor
{
}

//==========================================================================
//  Target non-blocking forward interface
//  (m_lookup_socket)
//==========================================================================
/// Target non-blocking transport forward interface
/// performing the communication, preparing and initiating computation
```

```
at_aes_memory::sync_enum_type                    ///< result
at_aes_memory::lookupNBTransport(                ///< nb_transport
  transaction_type&          trans,              ///< transaction
  phase_type&                phase,              ///< transaction phase
  sc_core::sc_time&          time                ///< elapsed time
  )
{
  switch (phase)
  {
    case tlm::BEGIN_REQ:
      assert(time == sc_core::SC_ZERO_TIME); // No TD in AT
      // Pipelined HW: Notify end of request phase after accept delay,
      // if no other request end is currently scheduled
      //if (m_RequestQueue.empty())
      //  m_EndRequestEvent.notify(global_delay_sbox_accept);
      //m_RequestQueue.push(&trans);

      // Notify start of work (carried out in zero time),
      // if no other work is currently scheduled
      // (without TD, a simple queue is enough to keep execution order)
      if (m_WorkQueue.empty())
        m_StartWorkEvent.notify(sc_core::SC_ZERO_TIME);
      m_WorkQueue.push(&trans);

      // Pipelined HW: Return END_REQ with associated delay
      //time += global_delay_sbox_accept;
      //phase = tlm::END_REQ;
      //return tlm::TLM_UPDATED;

      // SW or non-pipelined HW: Return END_REQ implicitly by BEGIN_RESP
      return tlm::TLM_ACCEPTED;
      break;
    case tlm::END_RESP:
      m_EndResponseEvent.notify(time);
      return tlm::TLM_COMPLETED;
      break;
    case tlm::END_REQ:   // initiator should never call with these phases
    case tlm::BEGIN_RESP://
    default:
      assert(0); exit(1);
  };
}
```

```
//============================================================================
//  Internal module functions
//============================================================================
/// Thread fetching new request, computing and scheduling the response
void at_aes_memory::WorkerMethod(void)
{
  assert(!m_WorkQueue.empty());
  transaction_type* trans_p = m_WorkQueue.front(); // Process oldest trans
  assert(trans_p);

  // Start processing transaction
  m_ResponseDelay = sc_core::SC_ZERO_TIME;

  sc_dt::uint64    address  = trans_p->get_address(); ///< memory address
  tlm::tlm_command command  = trans_p->get_command(); ///< memory command
  uint8_t* data_p = reinterpret_cast<uint8_t*>(trans_p->get_data_ptr());

  // assure that call is meant for this module
  assert(address >> 2 == MODULE_MEMORY_ADDRESS >> 2);
  switch (command)
  {
    default:
    {
      break; // error
    }
    case tlm::TLM_WRITE_COMMAND:
    {
      break; // error
    }
    case tlm::TLM_READ_COMMAND: ///< read request - return the result
    {
      unsigned int sbox_id = address & 0x3;
      switch (sbox_id)
      {
        case(1):
          data_p[0] = S1[data_p[0]];
          data_p[1] = S1[data_p[1]];
          data_p[2] = S1[data_p[2]];
          data_p[3] = S1[data_p[3]];
          m_ResponseDelay += global_delay_sbox_read +
              global_delay_sbox_lookup;
          break;
        case(2):
          data_p[0] = S2[data_p[0]];
          data_p[1] = S2[data_p[1]];
          data_p[2] = S2[data_p[2]];
          data_p[3] = S2[data_p[3]];
          m_ResponseDelay += global_delay_sbox_read +
              global_delay_sbox_lookup;
          break;
        case(3):
          data_p[0] = S3[data_p[0]];
          data_p[1] = S3[data_p[1]];
          data_p[2] = S3[data_p[2]];
          data_p[3] = S3[data_p[3]];
          m_ResponseDelay += global_delay_sbox_read +
              global_delay_sbox_lookup;
          break;
        case(0):
          assert(data_p[0] < 10);
          data_p[0] = rcon[data_p[0]];
          m_ResponseDelay += global_delay_rcon_read +
              global_delay_rcon_lookup;
          break;
      }
      break;
    }
  }
  // SW or non-pipelined HW: trigger response after work is done
  m_BeginResponseEvent.notify(m_ResponseDelay);

  // Pipelined HW: Remove transaction from queue and rocess the next
      transaction, if available
  //m_WorkQueue.pop();
  //if (!m_WorkQueue.empty())
  //   m_StartWorkEvent.notify(sc_core::SC_ZERO_TIME);

  // Pipelined HW: Notify start of response phase after response delay, if
       no other transaction is yet scheduled for response
  //if (m_ResponseQueue.empty())
  //   m_BeginResponseEvent.notify(m_ResponseDelay);
  //m_ResponseQueue.push(trans_p);
}
```

```
/// Method executing callback to the Initiator to start the response phase
void at_aes_memory::BeginResponseMethod(void)
{
  //assert(!m_ResponseQueue.empty()); // Pipelined HW: assert resp. queue
  assert(!m_WorkQueue.empty()); // SW, non-pipelined HW: assert work queue

  // Start response phase of oldest transaction
  phase_type phase = tlm::BEGIN_RESP;
  sc_core::sc_time t = sc_core::SC_ZERO_TIME;
  //transaction_type* trans_p = m_ResponseQueue.front();
  transaction_type* trans_p = m_WorkQueue.front();
  assert(trans_p);

  // Set response data
  trans_p->set_response_status(tlm::TLM_OK_RESPONSE);
  switch (m_lookup_socket->nb_transport(*trans_p, phase, t))
  {
    case tlm::TLM_COMPLETED:
      m_EndResponseEvent.notify(t);  // Response phase ends after t
      break;
    case tlm::TLM_ACCEPTED:
    case tlm::TLM_UPDATED:  // Initiator will call nb_transport to
         indicate end of response phase
      break;
    case tlm::TLM_REJECTED:
    default:
      assert(0); exit(1);
  };
}
/// Method removing the finished transaction and starting a new response
    phase for the next transaction
void at_aes_memory::EndResponseMethod(void)
{
  //assert(!m_ResponseQueue.empty()); // Pipelined HW: assert resp. queue
  assert(!m_WorkQueue.empty()); // SW, non-pipelined HW: assert work queue
  //m_ResponseQueue.pop();
  m_WorkQueue.pop();
  // Pipelined HW: Notify begin of response phase for next transaction
      after new response delay, if a new transaction is available
  //if (!m_ResponseQueue.empty())
  //  m_BeginResponseEvent.notify(m_ResponseDelay);
  // SW or non-pipelined HW: Notify work begin for next transaction if a
```

```
      new transaction is available
  if (!m_WorkQueue.empty())
    m_StartWorkEvent.notify(sc_core::SC_ZERO_TIME);
}
```

C.5. Switching between HW/SW behavior

Choice between concurrency behavior in at_aes_addkey - condensed

```
//==========================================================================
//   Constructors and destructor
//==========================================================================
at_aes_addkey::at_aes_addkey (                    ///< constructor
        sc_core::sc_module_name  module_name      ///< module name
    ) :                                           ///< initializations
    ...
    , m_module_type (SW)
   {
    ...
    SC_THREAD(Encrypt);
    SC_THREAD(SKeyComThread);
    SC_THREAD(NextStateComThread);
   }
...
//==========================================================================
// User function for coordination of encryption
//==========================================================================
void at_aes_addkey::Encrypt()
{
  ...
  while(1)
  {
    /* Wait for wakeup call from transport procedure */
    wait(m_StartEncryptEvent);
    ...
    for(round = 0; round<10; round++)
    {
      ...
      if (m_module_type == SW)  // Notify communication sequentially
      {
        // Trigger SKey key read
        m_StartSKeyTransEvent.notify(sc_core::SC_ZERO_TIME);
        wait(m_SKeyTransDoneEvent);
        // Trigger NextState write/read
        m_StartNextStateTransEvent.notify(sc_core::SC_ZERO_TIME);
        wait(m_NextStateTransDoneEvent);
      }
      if (m_module_type == HW)  // Notify communication concurrently
      {
        // Trigger SKey key read
        m_StartSKeyTransEvent.notify(sc_core::SC_ZERO_TIME);
        // Trigger NextState write/read
        m_StartNextStateTransEvent.notify(sc_core::SC_ZERO_TIME);
        wait(m_SKeyTransDoneEvent & m_NextStateTransDoneEvent);
      }
      ...
    }
    /* Reset control variables for possible next encryption */
    ...
  }
}
```

Appendix D.

Bibliography

[AHL+06] K. Ambrosch, C. Helpa, J. Lechner, R. Leidenfrost, T. Panhofer, A. Platschek, S. Ramberger, U. Stadler, D. Steiner, H. Trinkl, C. Widtmann, and M. Delvai. Design Variety in Hardware/Software Codesign - Implementations of an AES Encoder. In *Austrochip proceedings, Vienna, Austria*, 2006.

[BD05] David C. Black and Jack Donovan. *SystemC: From the Ground Up*. Springer, Berlin, 2nd printing edition, Oct. 2005.

[Dou08] Doulos Ltd. Getting Started with TLM-2.0. `http://www.doulos.com/knowhow/systemc/tlm2/`, 2008.

[Ele00] Petru Eles. System Synthesis - VHDL Basic Issues and Simulation Semantics. Linköping University, Linköping, Sweden. `http://www.ida.liu.se/~petel/SysSyn/lect2.frm.pdf`, 2000.

[FD01] Viktor Fischer and Milos Drutarovsk. Two Methods of Rijndael Implementation in Reconfigurable Hardware. *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 77–92, 2001.

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387262326.

[Goo05] Tom Gooch. History of UML. `http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm`, Dec. 2005.

[Gro02] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, 2002. ISBN 1402070721.

[IEE01] IEEE Computer Society. *IEEE standard Verilog hardware description language, IEEE Std 1364-2001*, 2001.

[IEE03] IEEE Computer Society. *IEEE standard SystemC language reference manual, IEEE Std 1666-2005*, Mar. 2003.

[KHA05] Tim Kogel, Anssi Haverinen, and James Aldis. *OCP TLM for Architectural Modelling*, 2005.

[MB06] Clive Maxfield and Alvin Brown. DIY Calculator. `www.diycalculator.com/sp-compuniverse.shtml`, 2006.

[MBP07] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification. A Prescription for Electronic System Level Methodology (Systems on Silicon)*. Morgan Kaufmann, Mar. 2007.

[MRR03] M. Müller, W. Rosenstiel, and J. Ruf. *SystemC - Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[MS04] Sumio Morioka and Akashi Satoh. A 10-Gbps full-AES crypto design with a twisted BDD S-box architecture. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(7):686–691, 2004.

[Nat01] National Institute of Standards and Technology (NIST). Announcing the Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication*, (197), Nov. 2001.

[Ope04] Open SystemC Initiative. *SystemC Synthesizable Subset Draft 1.1.18*, Dec. 2004.

[Ope07a] Open SystemC Initiative. *Requirements specification for TLM 2.0, Version 1.1*, Sep. 2007.

[Ope07b] Open SystemC Initiative. *TLM2 User Manual, TLM 2.0 draft 2, Version 1.0.0*, Oct. 2007.

[RDL05] Tero Rissa, Adam Donlin, and Wayne Luk. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In *DATE '05: Proceedings of the* *conference on Design, Automation and Test in Europe*, pages 253–258. IEEE Computer Society, 2005. ISBN 0-7695-2288-2.

[RSQL04] Gael Rouvroy, Francois-Xavier Standaert, Jean-Jacques Quisquater, and Jean-Didier Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications. *ITCC*, 02:583, 2004.

[Sys04] Stuart Swan (Cadence Design Systems). SystemC 2.1 Overview. www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-9-OSCI_2_swan.pdf, Feb. 2004.

[Wik08] Wikipedia. The free encyclopedia. http://www.wikipedia.org, Nov. 2008.

[Wil06] André Willms. *C++ Master Class. Einstieg für Anspruchsvolle*. Addison-Wesley, München, Jan. 2006. ISBN 9783827323682.

[Zab03] Enrique Zabala. Rijndael Cipher. Universidad ORT, Montevideo, Uruguay. http://www.formaestudio.com/rijndaelinspector/, 2003.