# Towards the Efficient Creation of Accurate and High-Performance Virtual Prototypes

Thesis approved by the
Department of Computer Science
Technical University of Kaiserslautern
for the award of the doctoral degree:
Doctor of Engineering (Dr.-Ing.)

Dean:
Prof. Dr. Klaus Schneider

under supervision of

Prof. Dr. Christoph Grimm
Institute: Design of Cyber-Physical Systems
Technical University of Kaiserslautern

and

Prof. Dr. Oliver Bringmann
Institute: Embedded Systems
Eberhard Karls University of Tübingen

by

Simon Hufnagel
Eugen-Bolz-Straße 7, 71636 Ludwigsburg

Date of Defense 17 July 2014

**D 386**

# Abstract

As the complexity of embedded systems continuously rises, their development becomes more and more challenging. One technique to cope with this complexity is the employment of virtual prototypes. The virtual prototypes are intended to represent the embedded system's properties on different levels of detail like register transfer level or transaction level. Virtual prototypes can be used for different tasks throughout the development process. They can act as executable specification, can be used for architecture exploration, can ease system integration, and allow for pre- and post-silicon software development and verification. The optimization objectives for virtual prototypes and their creation process are manifold. Finding an appropriate trade-off between the simulation accuracy, the simulation performance, and the implementation effort is a major challenge, as these requirements are contradictory.

In this work, two new and complementary techniques for the efficient creation of accurate and high-performance SystemC based virtual prototypes are proposed: Advanced Temporal Decoupling (ATD) and Transparent Transaction Level Modeling (TTLM). The suitability for industrial environments is assured by the employment of common standards like SystemC TLM-2.0 and IP-XACT.

Advanced Temporal Decoupling allows for cycle accurate simulation results in the context of SystemC TLM-2.0 temporal decoupling. In ATD, accesses to shared resource are managed by Temporal Decoupled Semaphores (TDSems) which are integrated into the modeled shared resources. The set of TDSems assures the correct execution order of shared resource accesses and incorporates timing effects resulting from shared resource access execution and resource conflicts. ATD facilitates modeling of a wide range of resource and resource access properties like preemptable and non-preemptable accesses, synchronous and asynchronous accesses, multiport resources, dynamic access priorities, interacting and cascaded resources, and user specified schedulers prioritizing simultaneous resource accesses.

Transparent Transaction Level Modeling focuses on the efficient creation of virtual prototypes by reducing the implementation effort and consists of a library and a code generator. The TTLM library adds a layer of convenience functions to ATD comprising various application programming interfaces for inter module communication, virtual prototype configuration and run time information extraction. The TTLM generator is used to automatically generate the structural code of the virtual prototype from the formal hardware specification language IP-XACT.

The applicability and benefits of the presented techniques are demonstrated using an image processing centric automotive application. Compared to an existing cycle accurate SystemC model, the implementation effort can be reduced by approximately 50% using TTLM. Applying ATD, the simulation performance can be increased by a factor of up to five while retaining cycle accuracy.

# Acknowledgments

This work would have not been possible without the support and encouragement of many different people. At first, I would like to thank Prof. Dr. phil. nat. Christoph Grimm and Prof. Dr. Oliver Bringmann for the supervision of this work and their valuable feedback during various scientific discussions. Similarly, I would like to thank my friends and colleagues at the Robert Bosch GmbH for their advice and many inspiring dialogues, among them Dr. Nico Bannow, Dr. Hendrik Post, Dr. Christian Kerstan, and Dr. Tobias Kirchner. Special thanks go to my PhD colleagues Michael Frischke, Simon Roth, and René Guillaume for the creative and motivating working climate during the preparation of this work. I wish them all the best for their respective research works. Last but not least I would like to thank my family and my girlfriend Regina for their love and encouragement.

# Contents

# 1. Introduction

For decades, the market demand for new applications in the electronic sector has been significant. In times of social media, smart phones, and autonomous driving cars [Mar10] a further growing market demand on electronic devices seems to be probable.

The trend for miniaturization in semiconductor manufacturing led to nanometer scale structure sizes. In 1965, Gordon Moore [Moo65] observed, that the number of transistors contained in integrated circuits doubles every eighteen months. Even though the doubling period tends to be extended to a three year period [ITR11b], the exponential growth of the number of transistors per integrated circuit is still valid.

These advances in semiconductor technology enable single chip implementations of complex designs like wireless communication chips and Multi Processor System-on-Chips (MPSoCs). Due to the rising complexity of such systems, their development becomes more and more challenging. As the system development productivity is not able to keep pace with the semiconductor production capabilities, the so called "hardware design gap" arises [ITR11a].

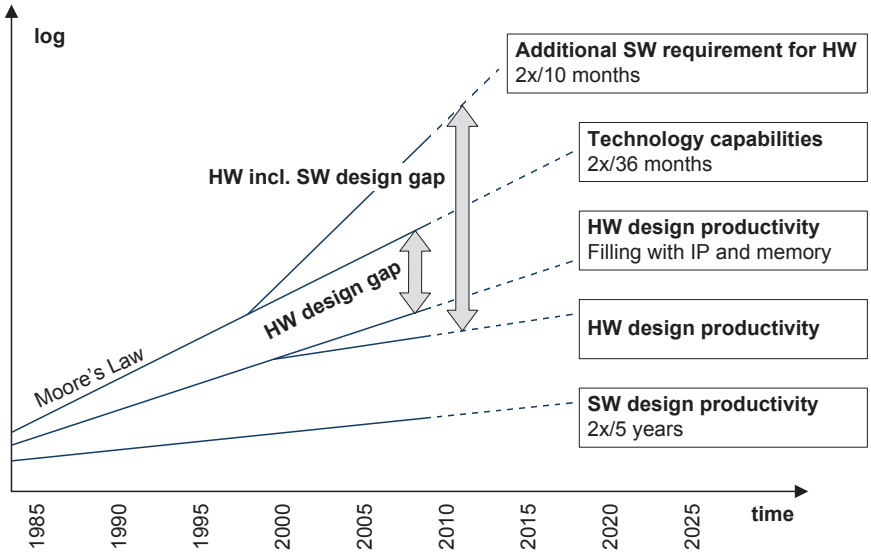



**Figure 1.1.:** Design Gap in Hardware and Software Design [ITR11a]: system development productivity does not keep pace with advances in semiconductor production technology due to the lack of efficient design methodologies.

Besides the discrepancy between the technology capabilities and the hardware design productivity, Figure 1.1 depicts another design gap which is called the "hardware including software design gap" [ITR11a]. As the amount of software needed to efficiently employ the available hardware

capabilities increases, the overall design gap grows further. As nowadays a large part of the software development process relies on the availability of the underlying hardware, the increasing amount of software prolongs the time-to-market which is a crucial factor. Figure 1.2 shows a conventional design flow consisting of a sequential set of design phases each depending on the outcome of the previous one. This is especially the case for the majority of the software design phase which depends on the preceding completion of the hardware design.


**Figure 1.2.:** Conventional system design flow (according to [CAT09]). The design flow consists of a sequence of design phases each depending on the outcome of the previous design phase.

## 1.1. Virtual Prototyping

A technique to cope with the design gaps is the utilization of Virtual Prototypes (VPs). A virtual prototype is a computer-simulation model of an electronic product, component, or system [Gro01] and represents hardware and software properties. These properties may comprise the functionality, the architecture, the timing aspects, and the power consumption. Depending on the intended use of the VP, the amount of available details for each property varies. These levels of available details are denoted as abstraction levels [GAGS09]. The higher the abstraction level, the lower the model complexity as irrelevant details are omitted. Besides reducing the implementation effort, the abstraction in general has a positive impact on the simulation performance of the model [CAT09].

Compared with physical prototypes, virtual prototypes provide a set of benefits. The implementation cost and effort of abstract virtual prototypes is lower and thus virtual prototypes are available early within the design process [Gro01]. As virtual prototypes are implemented as a computer based simulation, they provide a high observability, controllability, flexibility, repeatability, and comprehensive automation capabilities. Additionally, virtual prototypes can be easily duplicated and distributed.

Virtual prototypes can be used throughout the entire design flow. Especially during early design phases, the high abstraction levels of virtual prototypes leave a wide design space [JS05] which in turn is beneficial for tasks like early performance estimation and architecture exploration. After the system design has reached a stable state, the virtual prototype can facilitate the hardware and software development. In contrast to the sequential nature of the conventional design flow, the usage of virtual prototypes enables the parallelization of hardware and software development as shown in Figure 1.3.

During the hardware design phase the virtual prototype can be used as an executable specification accompanying the textual hardware specification derived from the system design process. Instead of describing the hardware behavior using text documents, the executable specification "behaves" like the intended hardware and thus can serve as a demonstrator and a basis for discussions during the refinement process. Additionally, the executable specification can be reused as a golden reference during the hardware verification phase.

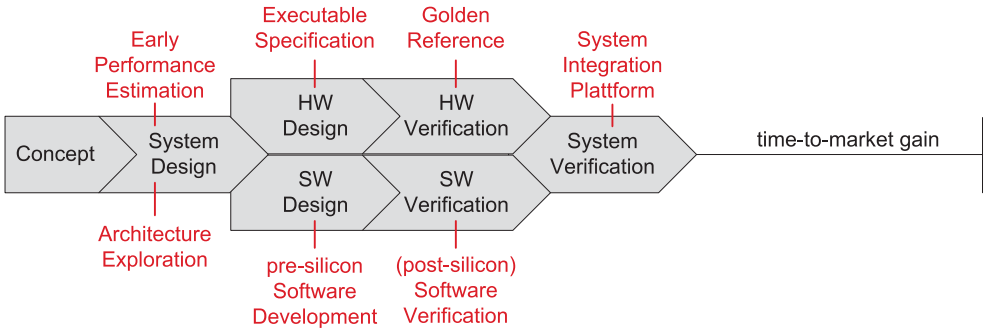



**Figure 1.3.:** Virtual prototype enabled system design flows allow for parallel hardware and software development leading to a significant improvement concerning time-to-market (according to [CAT09]). The usage scenarios of the virtual prototypes are annotated to each design phase.

At the same time, the virtual prototype can be used as an execution platform for pre-silicon software development. The software execution can either be done using a processor model, an Instruction Set Simulator (ISS) or by directly integrating the software source code into the virtual prototype implementation. In the first and the second case, the software is cross-compiled for the target system and executed on the ISS or on the processor model which represent the micro-architectural properties of the target CPU on a higher level of detail. In the third case, the micro-architectural properties are omitted and the software is compiled for the host CPU architecture, hence this approach is denoted as host compiled software execution. Besides the software development, the virtual prototype can be used for software verification. Even in case the physical hardware is already available, verification can benefit from the lower bring-up times and higher availability of the virtual prototype compared to the physical hardware prototype.

During the system integration and verification phase, the virtual prototype can be used as a system integration platform which is progressively substituted by the physical hardware implementation. This step typically necessitates mixed abstraction level simulation or co-emulation techniques [Kir11].

The optimization objectives for virtual prototypes and their creation process are manifold and partially contradictory. Therefore, the creation of a virtual prototype involves finding a trade-off between the achievable simulation timing accuracy, the simulation performance and the required implementation effort. In this work, a novel simulation technique which allows for improved simulation performance while retaining high simulation timing accuracy and a design flow which allows for the reduction of the implementation effort are proposed.

## 1.2. Outline

In Chapter 2 an overview of the state of the art is presented starting with a short introduction of the abstraction levels, design aspects, and hardware structure specification languages relevant for electronic system design. Next, the concept of the discrete event simulation technique is described, which is a wide-spread model of computation for virtual prototypes of digital hardware.

Subsequently, the Transaction Level Modeling (TLM) [IEE11] technique is presented, including an overview of the TLM variants and implementation techniques. The chapter also outlines the contradictory optimization objectives which are to be considered during the creation of Transaction Level Models: simulation timing accuracy, simulation performance, and modeling efficiency.

The Advanced Temporal Decoupling (ATD) approach that is a major novelty proposed in this thesis is presented in Chapter 3. ATD mitigates the conflict between the simulation timing accuracy of shared resource accesses and the simulation performance. This is accomplished by replacing the fixed data granularity used in common Transaction Level Models by a dynamically varying data granularity. The dynamic variation is achieved by exploiting the look-ahead feature arising from the temporal decoupling mechanism [IEE11] to calculate a time budget available for the execution of each transaction. The calculated time budget is as low as needed to allow for cycle accurate mutual exclusive shared resource accesses and as large as possible to achieve high simulation performance.

The Transparent Transaction Level Modeling (TTLM) methodology that is the second major novelty proposed in this thesis is presented in Chapter 4. TTLM allows for the reduction of the implementation effort and expert knowledge required for the creation of SystemC TLM-2.0 compliant virtual prototypes. The TTLM Library provides easy to use application programming interfaces for purposes like inter-module communication, runtime configuration, runtime information extraction, and the incorporation of the Advanced Temporal Decoupling simulation mechanism. The TTLM Generator can be used to automatically create the structural code of the virtual prototype from IP-XACT and therefore allows for the integration of the virtual prototype creation with established digital hardware and hardware dependent software design flows.

In Chapter 5 the simulation performance improvement achievable by ATD and the modeling effort reduction facilitated by the TTLM concept are analyzed using a virtual prototype of the automotive Night Vision system [Rob05]. The ATD and TTLM based virtual prototype is compared to a preexisting cycle accurate virtual prototype [Ban09].

Chapter 6 concludes this work and provides an outlook of possible extensions of the ATD and TTLM concepts.

# 2. State of the Art and Related Work

## 2.1. Electronic System Design Aspects and Abstraction Levels

The top-down design flow of electronic systems involves the consideration of different design aspects like the system's behavior, the architecture which decomposes the system into a set of components and the physical placement and dimensions of those components. The behavioral aspect describes the functionality of the system in terms of what the system does and it is the central aspect at the beginning of the top-down system development process. In the course of the design process, the system's architecture evolves taking various facets like suitability for the intended functionality and costs into account. Section 2.2 gives an overview of specification languages appropriate for the description of the hardware architecture. Especially for high level virtual prototyping which focuses on behavioral and architectural specification, the physical aspects are secondary. Figure 2.1 shows the Y-Chart which illustrates the design aspects and abstraction levels [DG90].

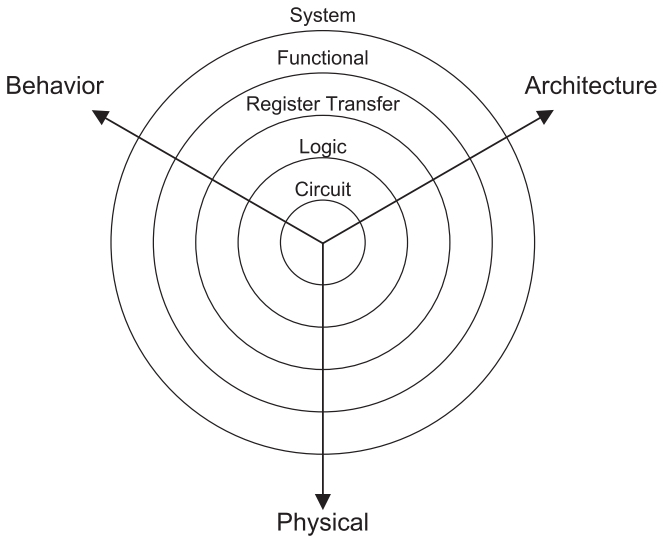



**Figure 2.1.:** The Y-Chart illustrates the three system design aspects behavior, architecture, and physical in conjunction with a set of abstraction levels [DG90]. Virtual prototypes are typically used on functional level, register transfer level, and on transaction level, which is an intermediate level between functional and register transfer level [MCG05].

The system can be represented at different abstraction levels ranging from system level to circuit level. The stepwise transition from higher abstraction levels to lower abstraction levels is denoted as refinement. To support the refinement process, virtual prototypes can be used and implemented at different abstraction levels.

Especially during the concept phase, the system is specified at the *functional level*. At this level, only the system's intended behavior and the incorporated algorithms and functionalities are present which can be modeled e.g. as a set of communicating sequential processes [BHR84]. The communication between these processes takes place using shared variables and function calls. Hardware aspects like a concrete architecture, the resulting timing, and power consumption are omitted.

The *transaction level* is an intermediate level between the functional level and the register transfer level (RTL) [MCG05]. This level extends the functional level by basic hardware structure information in the form of a set of interconnected modules each containing processes representing the modules behavior. Transaction level models provide a promising trade-off between implementation effort, simulation accuracy, and simulation performance suitable to fulfill the needs of virtual prototyping [CAT09]. A commonly used design language for transaction level models is SystemC TLM-2.0 [IEE11]. More details on transaction level modeling variants and techniques are presented in Section 2.4.

At the *register transfer level (RTL)*, the system is described using combinational logic changing the system's state over time. The communication is implemented using dedicated signals. The current state of the system is stored in registers and might change at clock cycle boundaries. Due to the comprehensive hardware awareness of this abstraction level, RTL descriptions are used to synthesize the final hardware. Commonly used design languages for register transfer level models are VHDL [IEE09], Verilog [IEE06], and SystemVerilog [IEE12].

## 2.2. Hardware Architecture Specification

Architectural information arises from the partitioning of the system and includes component structure and interface specifications. It can be incorporated at various steps throughout the design process such as the integration of Intellectual Properties (IPs) during platform assembly, the creation of hardware components, and the development of low level software. To enable the (partial) automation of the mentioned tasks, the architectural information has to be captured using machine readable formats [BM10]. The automation provides benefits such as a reduced implementation effort and a reduced implementation error rate.

Models like SystemRDL [Acc09], SpectaReg [PDT13], and IDesignSpec [Agn13] focus on the definition of register structures of memory mapped peripherals. The register structure is typically defined in a hierarchical way. The entire address space provided by the peripheral is divided into multiple register files. Each register file is composed of a set of registers. These registers in turn consist of fields logically grouping related bits.

In addition to the register specification capabilities of the mentioned languages, IP-XACT [IEE10] provides means to specify supplementary component structure aspects like bus interfaces and signal ports. Furthermore, IP-XACT facilitates the composition of components and hierarchical system descriptions by providing netlisting features like component instantiations and interconnections. In addition, IP-XACT supports the abstraction level concept by providing mechanisms

to specify component interconnections at different levels of abstraction e.g. at "wire" and "transactional" level.

As IP-XACT is maintained as an IEEE standard [IEE10], allows for a comprehensive machine readable specification of hardware architecture aspects, and provides extendability via vendor extensions, the format is the basis for many different design step automation approaches.

## 2.3. Discrete Event Simulation in SystemC

The Discrete Event Simulation (DES) technique is widely used for the simulation of digital hardware systems and employed by languages like VHDL [IEE09], Verilog [IEE06] and SystemC [IEE11]. In DES, there are two equivalent approaches to model the behavior of a system denoted as event-scheduling approach and process-interaction approach respectively [Fis01]. In SystemC, the event-scheduling approach can be implemented using `SC_METHOD`s triggered by events whereas the process-interaction approach can be implemented using `SC_THREAD`s. When using the process-interaction approach, each independent course of action is represented by a distinct thread. Each thread consists of an ordered set of statements which are executed in a sequential way. Sequential execution implies that in general at any given wall clock time, the thread is executing exactly *one* statement of the set of its statements. However, as the simulation time is only advanced by calling the `wait(<time>)` function, all statements belonging to one block surrounded by `wait(<time>)` calls occur simultaneously in simulation time.

Hardware-inherent parallelism is represented incorporating multiple threads which might be logically grouped in a set of modules. Interaction between threads can take place in various ways including explicit synchronization and data exchange. Explicit synchronization can be achieved using event notifications and reactions on notified events. Data exchange between threads residing in the same module can be done using signals or local (volatile) variables in combination with explicit synchronization mechanisms. To exchange data between threads residing in different modules, designated communication channels can be incorporated.

In Figure 2.2 a simplified state chart of the SystemC kernel implementing DES is given. The simulation progress consists of multiple timed notification cycles each being one turn in the timed notification loop [IEE11]. A timed notification cycle starts with the simulation of the entire activity at the current global simulation time. This entails the sequential processing of all triggered methods and all runable threads. A thread becomes runable either after a previously defined waiting time has elapsed or after a discrete event awaited by the thread gets notified. During the execution of a triggered method or a runable thread, other events might be notified to occur at simulation times equal or greater to the current global simulation time. To preserve the causality in case of events that are notified to occur at the current simulation time in a simulator implementation independent way, the delta notification loop mechanism is used [IEE11].

One turn of the delta notification loop is denoted as a delta cycle and is composed of three phases named evaluate phase, update phase, and notify phase respectively. The evaluate phase comprises the processing of all events that are known to occur at the current simulation time before the delta cycle was started. The execution order of these events depends on the implementation

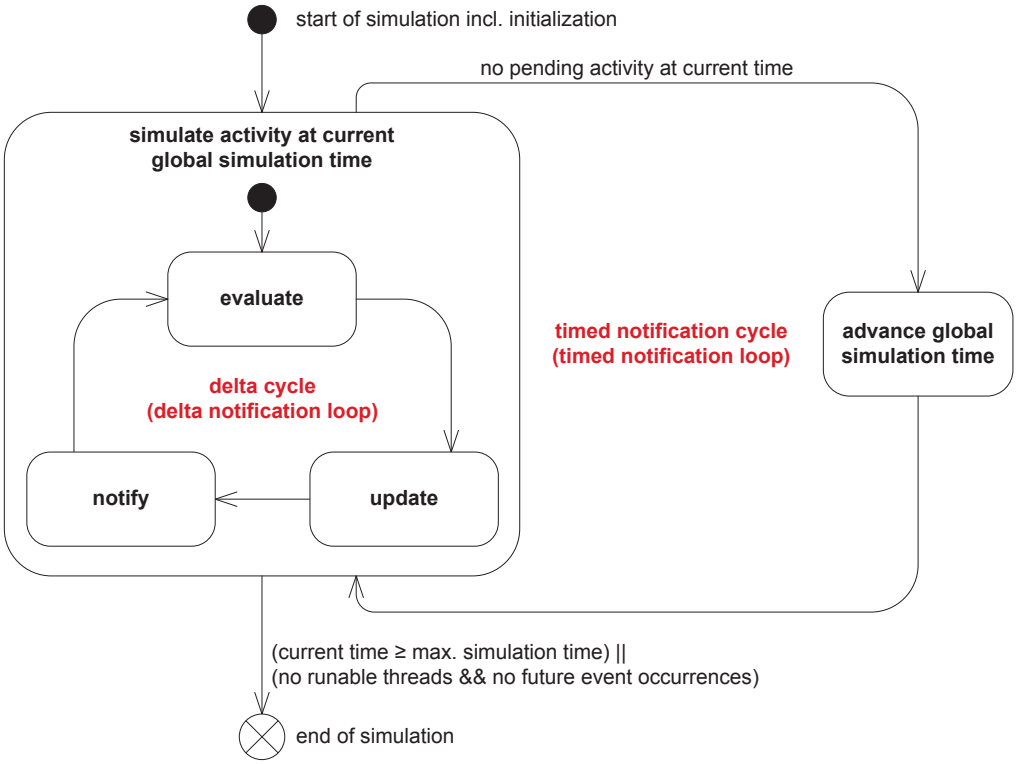



**Figure 2.2.:** Discrete Event Simulation modalities. In DES, computation takes place at distinct points in global simulation time determined by the occurrence of at least one event. For each timed notification cycle all occurring events are processed during an arbitrary amount of delta cycles. After the simulation state is settled meaning that there is no pending activity at current time, the global simulation time is advanced to the time of the next simulation event occurrence and the next timed notification cycle is started.

of the simulator and must not influence the simulation result. To accomplish this claim for the simulation of RTL models, the effect of signal assignments taking place during the evaluate phase is deferred to the following update phase to provide consistent data for all threads reading from the same signals during the current evaluate phase. During the update phase, the signal values are updated and signal change events are triggered. During the notification phase, events that are to occur in the following delta cycle are triggered causing all sensitive methods and threads waiting for these events to become runable.

The notification phase concludes the delta cycle. If there are processes that have become runable during the notification phase, the next delta cycle is started. Otherwise, the current timed notification cycle ends by advancing the global simulation time to the nearest time of any pending timed event occurrence. Additionally, threads that are waiting for the occurrence of at least one of these events are made runable.

Figure 2.3 illustrates the stepwise progress of the global simulation time in DES consisting of an arbitrary number of delta cycles per timed notification cycle. In contrast to other models of computation the simulation time step width in DESs is not necessarily constant.

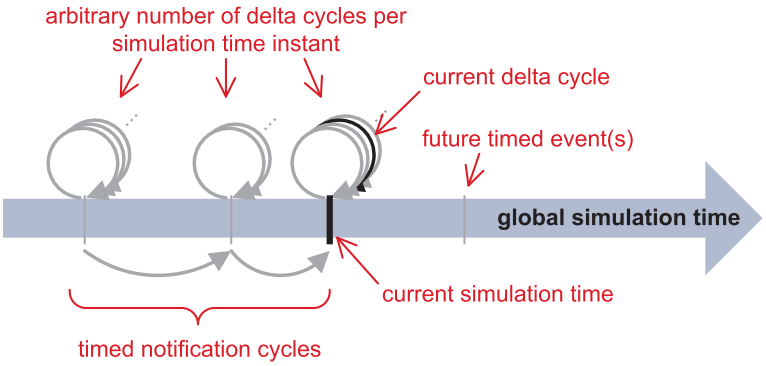



**Figure 2.3.:** Simulation time progress in DES. After all delta cycles belonging to one timed notification cycle have been processed, the global simulation time is advanced to the nearest time of any pending timed event occurrence.

## 2.4. Transaction Level Modeling (TLM)

At transaction level, the computational tasks and communication sequences are separated [CG03, Mey05]. According to the process-interaction approach of DES, the computational tasks can be implemented using a set of processes or threads. This allows to represent hardware and software components of a system in a common manner leaving a wider design space by deferring the determination of the implementation technology [JS05]. In contrast to signal based communication used at register transfer level, the inter-thread communication at transaction level takes place using transactions traversing abstract communication fabrics. A transaction represents a logically delimited interaction occurrence between a sending unit called initiator and a receiving unit denoted as target.

In the following, transaction level model variants, implementation techniques, and optimization objectives are presented.

### 2.4.1. Variants

Transaction level models can be categorized in various ways [MCG05]. The timing aspects of computational and communication tasks can be represented on a broad accuracy spectrum ranging from fully untimed omitting any notion of time to clock cycle accurate. The data granularity varies from application packets to bus words.

Application packets are typically used in functional models where both data size and semantic depend on the requirements of the high level algorithm. A popular example for application packets are video frames in video processing applications. If the level of detail achieved by the utilization of application packets is insufficient, application packets can be divided into smaller pieces like bus packets. Bus packets represent data portions transferred en bloc over a bus model. A typical example for bus packets are burst transfers. The size of a bus packet is jointly determined by the implementation details of the computing algorithms and the capabilities of the modeled hardware architecture (e.g. bus). The highest data granularity is achieved using one bus word per

transaction. Bus words are data portions atomically transferred over the communication fabric. The size of one bus word is typically defined by the data bus width.

Based on these data granularity and timing accuracy gradings, a set of TLM variants can be defined as shown in Figure 2.4 [MCG05]. This set comprises the following TLM variants in increasing timing accuracy order.

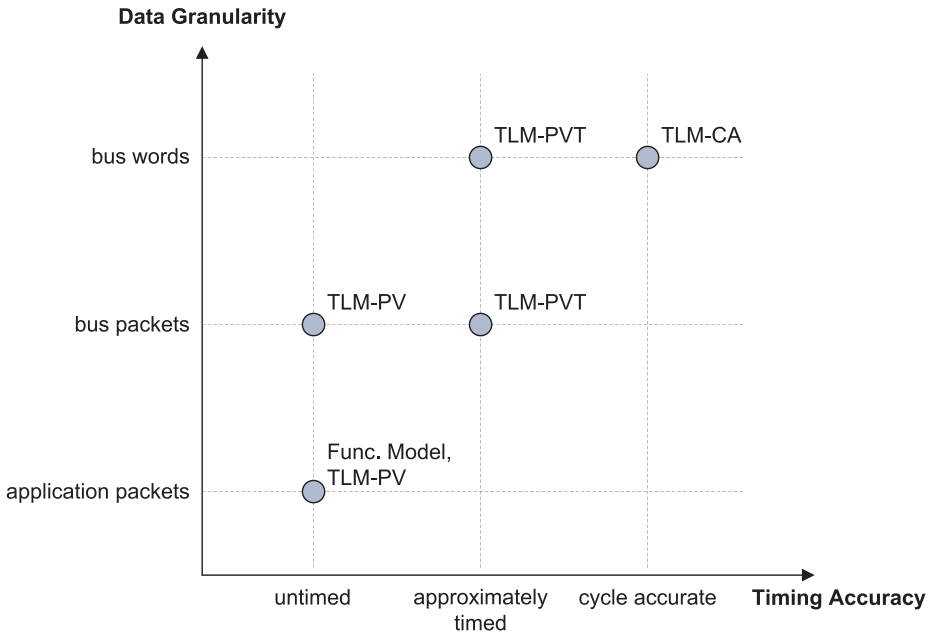



**Figure 2.4.:** Transaction Level Modeling Variants Overview [MCG05]: TLM variants arise from the comparison of transaction data granularity and achievable timing accuracy.

- TLM - Programmer's View [MCG05] (TLM-PV) models lack any notion of time but preserve functional and causal dependencies between events occurring during simulation. Together with the fact, that basic hardware architecture information is present, TLM-PV models are suitable for high level algorithm and software development. Due to the low data granularity of application and bus packets, the simulation speed of this kind of model is typically high.
- TLM - Programmer's View with Timing [MCG05] (TLM-PVT) models extend the preceding modeling variant by a basic amount of timing information representing the time consumption of behavioral and / or communicational procedures. Hence, this modeling variant is suitable for early performance estimation tasks and lower level software development. The data granularity typically ranges from bus packet to bus word size. The simulation speed varies depending on the actual data granularity.
- TLM - Cycle Accurate [MCG05] (TLM-CA) models provide cycle accurate timing information taking resource contention effects into account. In general, this requires the fragmentation of the transferred data into bus word sized transactions resulting in reduced simulation performance. Because of the exhaustive timing accuracy of cycle accurate models, these models can be used for precise performance estimation, driver software development for timing critical peripherals and the validation of register transfer level models.

### 2.4.2. Implementation Techniques

Independently of the chosen TLM variant and the appropriate data granularity, the transaction processing is implemented using one of the following techniques defined by the SystemC TLM-2.0 standard [IEE11]. Basically, these techniques differ in the number of available transaction processing phases resulting in a varying amount of function calls used to conduct one arbitrarily sized transaction.

#### 2.4.2.1. Approximately-timed Style

Using the approximately-timed style, each transaction passes a set of phases. The set of available phases depends on the employed communication protocol. The SystemC TLM-2.0 base protocol defines four phases namely begin_request, end_request, begin_response, and end_response. The lifecycle of each transaction can be implemented by one to four function calls. The approximately-timed implementation shown in Figure 2.5 uses two function calls to process a single transaction.

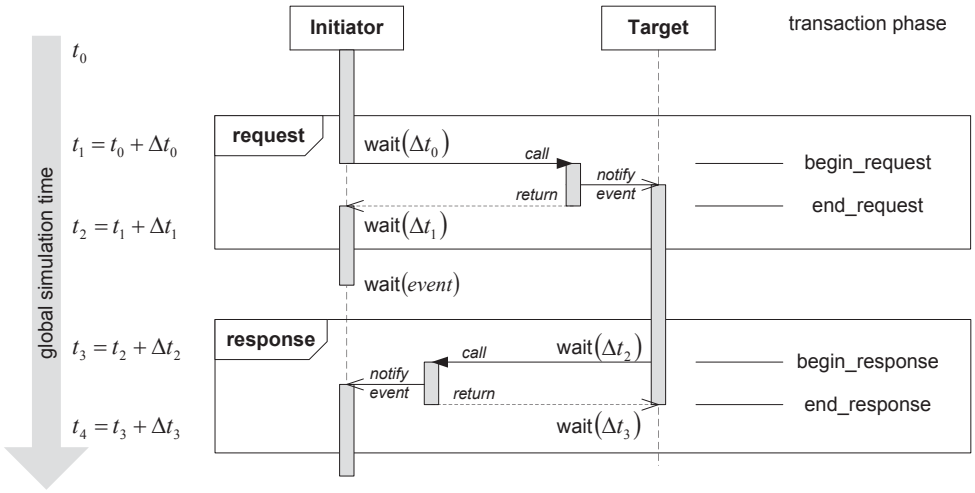



**Figure 2.5.:** Approximately-timed TLM style [IEE11]. When using the return path, one transaction is typically conducted using one function call from initiator to the target (request) and a reverse function call (response). Typically, each of the function calls comprises an event notification triggering the destination thread.

The first function call from the initiator to the target represents the interaction request where the invocation of the function states the begin_request. The phase transition to end_request takes place using the return path. The request typically leads to an event notification triggering the process contained in the target module to process the transaction. After the request has been processed by the target, the response is transferred using the second function call from the target to the initiator. Similar to the request, the response starts with the function invocation (begin_response) and ends with the return statement (end_response) and typically leads to another event notification triggering the initiator's thread. It should be noticed that not all phase transitions are mandatory allowing for early completion of transactions.

Typically each phase transition is accompanied by a call to the wait(time) function which mimics the time consumption that has occurred during the preceding phase. Each wait(time) call increases the global simulation time resulting in four timing points per transaction.

#### 2.4.2.2. Loosely-timed Style

If two timing points per transaction are sufficient in terms of the desired simulation timing accuracy, the loosely-timed style can be used. Figure 2.6 shows the message sequence chart of the loosely-timed transaction implementation.

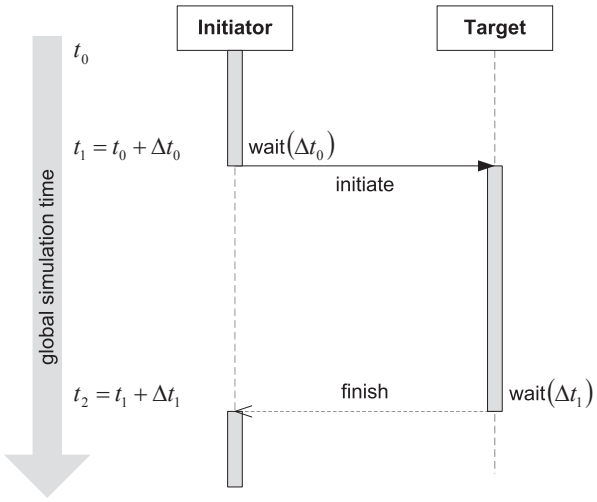



**Figure 2.6.:** Loosely-timed TLM style [IEE11]. Each transaction is conducted using one function call from the initiator to the target resulting in two timing points per transaction.

Each transaction is represented by exactly one function call. The transaction is started by the initiator on invocation of the function and is ended by the return of the function. Similar to the approximately-timed style, the transaction's start and end points are accompanied by a call to the wait(time) function. The first wait(time) call represents the initiator's time consumption in advance of the transaction. The second wait(time) call indicates the duration of the transaction.

In approximately timed or loosely timed models the simulation time consumption of each thread can be classified into four simulation time consumption states as shown in Figure 2.7.

Each thread starts in the **self contained execution** state. In this state, the thread executes its functionality which can not be influenced by other parts of the system. In case the further progress of the thread depends on the occurrence of an event, the state **waiting for event occurrence** is entered. After the corresponding event has been notified, the thread returns to the **self contained execution** state.

In case the thread attempts to access any shared resource, the next state depends on the availability of the accessed resource. If the resource is currently unavailable, the **blocked due to resource contention** state is entered and any further processing of the thread is deferred until the resource becomes available. Otherwise, the **interacting with shared resource** state is entered and the communication can take place. In case another thread having a higher priority attempts to access

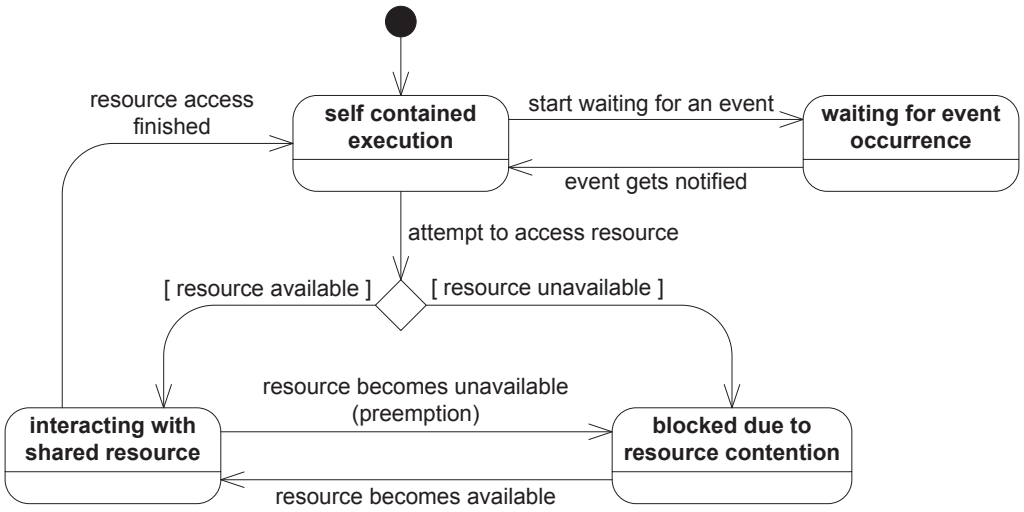



**Figure 2.7.:** Simulation time consumption state chart for approximately timed or loosely timed threads.

the same resource, the current thread might get preempted depending on the capabilities of the shared resource. In this case, the **blocked due to resource contention** state is revisited until the resource contention has disappeared. After the interaction is finished, the thread returns to the **self contained execution** state.

#### 2.4.2.3. Loosely-timed Style with Temporal Decoupling

Calls to the wait(time) function provoke a context switch between simulation threads [IEE11] and thus they are costly in terms of simulation performance. Therefore, the goal of the temporal decoupling mechanism is to reduce the number of wait(time) calls to speed up the simulation. This is achieved by aggregating the simulation time consumption of each simulation thread using a local simulation time offset. Figure 2.8 illustrates the temporal decoupling mechanism spanning two transactions.

As the temporal decoupling mechanism extends the loosely-timed style, each transaction is represented by exactly one function call from the initiator to the target. In contrast to the conventional loosely-timed style, the transaction's start and end point are not accompanied by wait(time) calls. Instead, the local simulation time offset of the transaction is annotated to the function call and consumed all at once at some later point in simulation. Effectively, this means that the simulation threads are allowed to run ahead in simulation time resulting in a temporal decoupling of these threads. This effect is often denoted as local time warp [IEE11].

Due to the absence of context switches and the cooperative multitasking scheme of SystemC, the execution of concurrent threads is deferred. This might result in simulation errors like out-of-order execution of transactions or disregarded mutual exclusive access to shared resources. Thus, explicit care has to be taken by the designer to avoid data dependency violations by manually employing synchronization mechanisms.

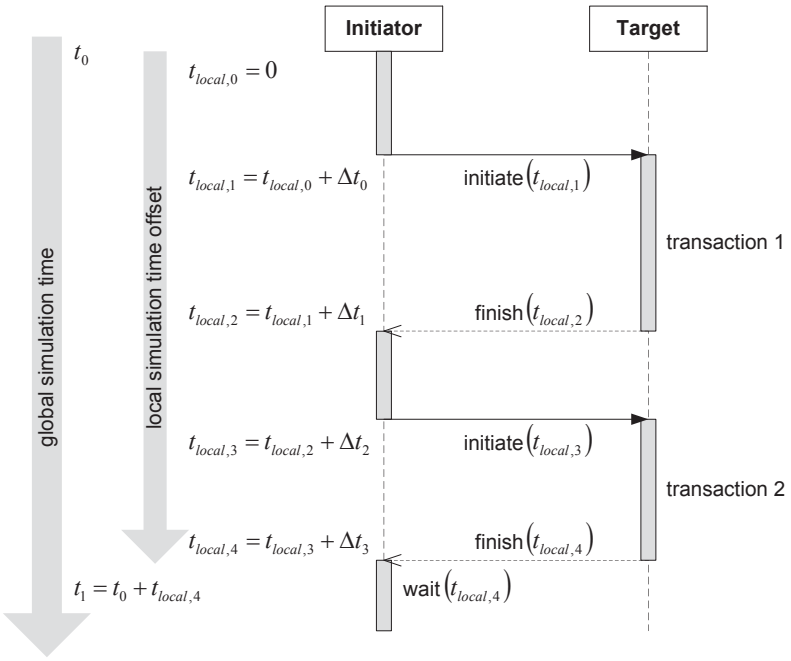



**Figure 2.8.:** Temporal Decoupling Concept [IEE11]. To reduce the number of context switches provoked by calling the wait(time) function, the time consumption of multiple transactions is aggregated using a local simulation time offset and consumed at a later point in simulation. Thereby, the simulation threads are allowed to run ahead in simulation time resulting in a temporal decoupling of the involved simulation threads.

### 2.4.3. Optimization Objectives of TLM implementations

For a valuable incorporation of transaction level models in virtual prototyping of digital hardware, the transaction level models have to meet a set of requirements [MCG05].

- Timing Accuracy: to gather reliable timing estimations from virtual prototypes, a high timing accuracy for transaction level models is required. Especially, for the development of real-time applications, the achieved timing accuracy of the utilized models is crucial.
- Simulation Performance: particularly for pre-silicon software development where a considerable amount of software has to be executed upon the virtual prototype, a high simulation performance is desirable to reduce turnaround times and allow for efficient software debugging.
- Modeling Efficiency: as the time-to-market gain achieved by the incorporation of virtual prototypes directly depends on the time of their availability, the creation of virtual prototypes and therein contained transaction level models has to be efficient.

As these optimization objectives are partially contradictory, the creation of transaction level models incorporates finding a suitable trade-off. Figure 2.9 illustrates this trade-off by depicting the degree of requirement fulfillment for different transaction level modeling styles. This set comprises the cycle accurate (TLM-CA), approximately-timed (TLM-AT), loosely-timed (TLM-LT) and loosely-timed style with temporal decoupling (TLM-LT + TD). The actual values for these

metrics depend on the characteristic of the modeled system and various other factors. Thus, only trends are given.

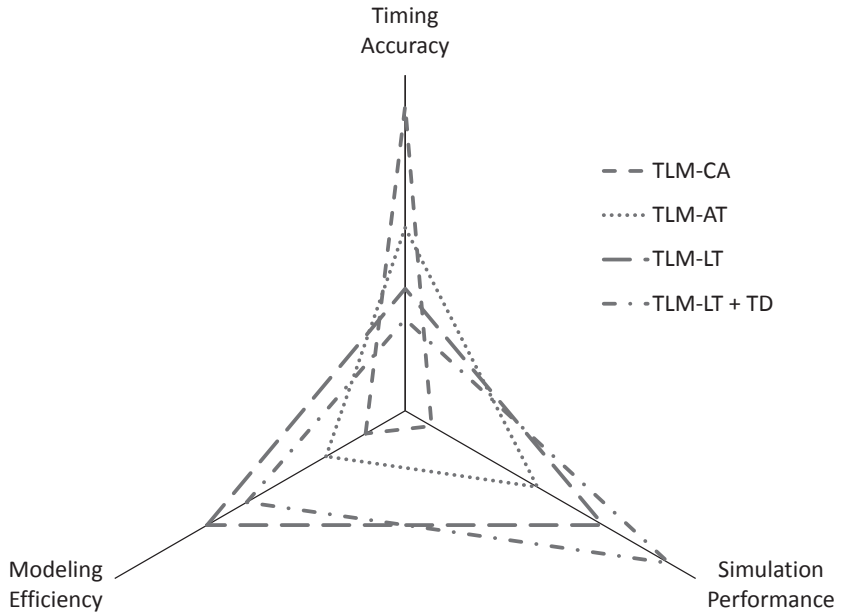



**Figure 2.9.:** The TLM optimization objectives Modeling Efficiency, Timing Accuracy, and Simulation Performance are achieved in varying extent depending on the applied modeling style. Cycle accurate models (TLM-CA) provide extensive timing accuracy at a low simulation performance and modeling efficiency. Approximately-timed models (TLM-AT) provide higher simulation performance and modeling efficiency but lower timing accuracy. Loosely-timed models (TLM-LT) provide the highest modeling efficiency, high simulation performance but further reduced timing accuracy. Loosely-timed models with temporal decoupling (TLM-LT + TD) provide the best simulation performance and a high modeling efficiency at the cost of poor timing accuracy.

The timing accuracy is affected by the amount of timing details modeled by the designer. Furthermore, the timing accuracy is limited by the simulation modalities of SystemC TLM-2.0. Especially the data granularity used in non-cycle accurate transaction level modeling styles restricts the timing resolution and the resulting timing accuracy [MCG05, SD07]. Transactions exceeding the atomic time unit of the modeled hardware are denoted as non-atomic transactions and are mainly used in approximately- and loosely-timed transaction level models. The reason for the timing inaccuracy caused by the simulation modalities is, that the preemption of non-atomic transactions in case of resource contention is not supported inherently. This implies that potential preemptions occurring in the real system are disregarded in the simulation model. If high timing accuracy is required, a common approach is to increase the data granularity by precautionary partition non-atomic transactions into multiple atomic transactions. The atomic transactions are transferred successively allowing for interruptions at the borders of atomic transactions. Effectively, this results in a cycle accurate modeling style.

The simulation performance of discrete event simulations mainly depends on the amount of events occurring per simulated time [Ulr68]. As the size and the operational state of the modeled system

typically influence the number of events per simulated time, the simulation performance is affected by the characteristics of the modeled system. In addition, the simulation performance is influenced by the modeling style and the resulting ratio of simulated functionality per simulation event. Especially for transaction level models this ratio results from the chosen data granularity. This means, if the data granularity is reduced, the transaction size and the simulation performance are rising. The simulation performance of non-cycle accurate transaction level models having low data granularity exceeds the simulation performance of cycle accurate transaction level models by some orders of magnitude [SD09, GAGS09].

### 2.4.4. Increasing the Modeling Efficiency

The modeling efficiency is influenced by various factors like the availability of modeling libraries, modeling tools and the skills and experience of the designer. Modeling libraries contain ready-to-use components like processor, bus, and peripheral models to compose virtual prototypes and / or elements simplifying the creation of custom components.

#### 2.4.4.1. Modeling Libraries

In [Ban09] a SystemC based library focusing on communication centric tasks is presented. The most important part of the library is the module adapter class. Module adapters are used to automatically transfer application packets by partitioning them into atomic transactions which are successively transferred. Thereby mechanisms like out-of-order and read / read-response accesses are supported. Similarly, the TLM+ library provides means to easily transfer "complex information structures" [EES+10] between components of the virtual prototype. In contrast to the module adapter library, the TLM+ library omits the partitioning of the "complex information structures" and transfers the information using a single function call. This allows for "faster simulation runs" [EES+10] at the cost of simulation accuracy. The concept of convenient library elements providing communication capabilities is pursued by the TLM-module-adapters presented in [Ker09]. The TLM-module-adapters encapsulate SystemC TLM-2.0 specific communication code by providing convenient read and write methods. Especially for the creation of target modules, the SystemC Modeling Library 2 [Syn11] (SCML2) [Syn11] provides valuable means for modeling memory mapped peripherals and their register structure. The GreenSocket[Gre12] library is compatible to the SystemC TLM-2.0 sockets and facilitates the usage of extensions allowing to augment the payload with additional user defined information.

Besides the previously mentioned modeling libraries aiming at the simplification of the creation process of custom components, there are component libraries for various purposes and domains. A modeling library containing ready-to-use ARM processor models is available at [ARM13]. In [TUV13], a library containing analogue and mixed-signal building-blocks is presented. Especially for the development of software for embedded systems, an open library "of processor and behavioral models" as well as "APIs for building [..] processors, peripherals and platforms" is provided at [Imp14]. Another library of commercially available components can be found at [TLM14].

#### 2.4.4.2. Modeling Tools and Design Automation

To further leverage the benefits of modeling libraries, tools can be employed to automate design tasks. Especially for hardware architecture centric tasks like the generation of structural component code for different design artifacts and target languages, IP-XACT is frequently incorporated as the central data source.

In [PWB11], the authors present a Perl based framework for the generation of VHDL entity declarations from IP-XACT component descriptions. The generation of transaction level component models is shown in [Cad11]. In [vHdKV11] the creation of transaction level models is extended by the automatic creation of verification software, used for reset tests, accesses tests, and address tests for the generated registers.

Another use case of IP-XACT in system design is the support of IP integration. In [KvdWdK+08] an overview of IP-XACT based industrial IP integration flows is given. In [XDM10] and [LBC+10] two IP integration flows are presented which allow for the automated assembly of IP components to virtual prototypes. The authors of [ZBBR09] present an IP integration methodology based on the generation of tailor-made adapters which are used to map different bus protocols to a generic bus protocol. The authors of [AEG+10] present an integration methodology based on a library of parameterizable IP cores suitable for the creation of communication circuits.

The previously mentioned works utilize the information stored in IP-XACT to partially automate design steps like the generation of component code. The other direction is followed by the authors of [LTdS11]. They present an approach to populate data from virtual prototypes to IP-XACT by extracting structural information from SystemC models which therefore have to be assembled of specific coding constructs recognizable by the employed code parser.

Besides the mentioned IP-XACT based tools, there are various commercial tools available enabling the creation of virtual prototypes partially based on proprietary mechanisms (e.g. [Cad13, Men13, Syn13]).

### 2.4.5. Handling the Trade-Off between Accuracy and Simulation Performance

#### 2.4.5.1. Simulation Performance Improvement Techniques

**Parallel Execution of SystemC Models**

The SystemC reference implementation only allows for a sequential execution of simulation threads resulting in a dissipation of processing resources on multicore host CPUs. There are various approaches aiming to overcome the shortcoming of idle host processing capabilities [GMPCM09]. One technique to enable parallel execution of SystemC models is to use one dedicated SystemC kernel instance per host CPU core each simulating a part of the model [SKR09]. This approach requires a preceding model partitioning step which is either done manually [PVSL11] or in an automated way [HBHT08]. Communication spanning multiple partitions has to be implemented using common inter process communication mechanisms like shared memory or sockets. As these inter process communication mechanisms imply a significant processing overhead, beneficial model

partitioning resulting in minimum communication occurrences across partition boarders is crucial. Instead of using multiple SystemC kernel instances, a single parallelized SystemC kernel instance utilizing multiple host CPUs can be used [MMGP10, Jon11, SLPH10, CCZ06].

In addition, there are several approaches utilizing Graphic Processing Units (GPUs) instead of CPUs for the parallel execution of SystemC models. In [NPJS10] and [VCBF12] code transformation based techniques are presented which allow for the execution of RTL code or dataflow centric SystemC code on General Purpose Graphic Processing Units (GPGPUs) using the Compute Unified Device Architecture (CUDA) [NVI13].

#### Trace driven Simulation Acceleration

Besides performance improvements achieved by the parallel model execution, there are approaches using traces to speed up the simulation. A trace is a "record of the execution of a computer program, showing the sequence of instructions executed" [IEE90]. The traces are used to replace frequently simulated portions of complex system behavior by abstract representations. The traces can either be extracted from high level system descriptions using symbolic search techniques [GHT12] or recorded from lower level cycle accurate models which have been executed previously [PMG+10].

### 2.4.5.2. Timing Accuracy Improvement Techniques

For the sake of high simulation performance, transaction level models are often implemented with low data granularity using non-atomic transactions resulting in timing errors due to disregarded resource conflicts and neglected transaction preemptions. There are various techniques to reduce the extent of these timing errors.

#### Accuracy Alteration

Especially for employment in temporal decoupled transaction level models, the SystemC TLM-2.0 standard [IEE11] provides the global quantum mechanism which limits the offset between the global simulation time and the thread's local simulation time. Low global quantum values result in an improved timing accuracy at the cost of simulation performance, high global quantum values speed up the simulation at the cost of timing accuracy. To take advantage of this concept, the designer can incorporate the quantum keeper utility class [IEE11]. However, even for very low quantum values, the global quantum mechanism can not guarantee the absence of simulation timing errors caused by temporal decoupling.

Besides this mechanism provided by the SystemC standard, there are some alternative approaches allowing the alteration of the simulation timing accuracy. In [RK08] an adaptive accuracy switching technique for arbitrating bus models is presented. Transactions traversing the bus model are withheld until the estimated end time of the transaction is reached. To estimate the transaction's end time, it is assumed, that no preemption occurs. When the global simulation time reaches the estimated end time and no preemption by a higher priority transaction has been requested,

the current transaction is atomically conducted. Otherwise, the transaction is split up allowing for cycle accurate bus transaction implementation. Due to the proposed preemption detection mechanism, transactions arrive late at the target.

In [BSS07], a methodology for run-time accuracy switching is presented. Different techniques for channels and modules are combined. Multiple channel models having different abstraction levels are combined within a multilevel channel providing a unified outer interface. At runtime, the appropriate channel model can be selected to either provide high simulation performance or high timing accuracy. The accuracy switching of modules is done by temporarily disconnecting the computation module from the clock distribution system and other global synchronization signals. Thereby, synchronization is reduced to an user defined extent accepting a loss in timing accuracy.

#### Analytical and Statistical Resource Contention Analysis

These approaches have in common, that the probability of resource conflict occurrences and their influence on the simulated time are estimated at model creation time. Analytic approaches typically use formal component models which either depend on the determinism of the application and the platform [CKT03] or aim at the calculation of worst case execution times [SHRE09] being an over estimation of the actual resource contention effects.

In statistical approaches resource characteristics, resource utilization, and resource contention effects are approximated incorporating statistical methods [Bob07, BPT07, ANMD07]. Alternatively, the data can be extracted from previous simulation runs [LRD01]. During the simulation, the gathered information is employed to estimate the timing effects of resource contentions.

#### Postponed Transaction Timing Correction in case of Resource Conflicts

To reduce the timing errors resulting from disregarding preemptions of non-atomic transactions, the "postponed transaction timing correction in case of resource conflicts" mechanism can be used. Using this approach, non-atomic transactions are processed in a non-preemptable way, temporarily assuming that no preemption takes place. In addition, these approaches comprise mechanisms to automatically detect resource conflicts afterwards. In case a contention is detected, the simulation time of the affected initiator thread is updated postponed to the processing of the corresponding transaction [SD07].

Using this technique, the timing errors resulting from resource conflicts can be reduced while avoiding the simulation performance degrading precautionary partitioning of non-atomic transactions into atomic ones. But, as the non-atomic transactions are processed in a non-preemptive way, this approach may lead to data dependency violations between transaction fragments of interleaving transactions.

In [SBR11] the so-called Quantum Giver applies the postponed transaction timing correction approach to temporal decoupled models introducing several distinct and alternating simulation phases denoted as simulation phase, synchronization phase, and scheduling phase respectively. During the simulation phase, all runable initiators are executed and "all transactions

issued by an initiator are completed immediately, meaning without synchronizing with other processes" [SBR11]. During the synchronization and scheduling phase, the initiator's local quantum values available during the following simulation phase are calculated taking resource contention occurrences into account. Additionally, resume events used to wake up the corresponding initiator threads are scheduled.

The processing of a transaction by one resource might lead to the occupation of other resources as it is the case for bus transactions which are routed to other resources. These dependencies between resources are essential for simulation accuracy and must not be disregarded. In [EES+10] this issue is addressed using an external statically defined data dependency table specifying which additional resources are required to complete a transaction processed by another resource. For this task, a Domain Specific Language (DSL) is used.

In [LCH08] the multiple alternating simulation phase concept is used to enable temporal decoupling and postponed timing correction for the simulation of cycle accurate transaction level models. During the virtual synchronization simulation phase, the functionality of all initiator modules is executed. Each master module has a local clock cycle counter and is allowed to run ahead of the global simulation time. The initiator's execution is preempted if a data dependency is detected to allow the rest of the system to catch up in simulation time. Data dependency detection is done querying a user provided data dependency table, which specifies memory regions used by multiple initiators. During the cycle count reconstruction phase, the local clock cycle counters of initiators affected by resource conflicts are updated.

The postponed timing correction mechanism is not limited to hardware modeling but can also be applied to Real Rime Operating System (RTOS) and software task models where CPUs are treated as shared resources [ZMG09, ZM08]. In this work, the software tasks and interrupt service routines are divided into segments. The functionality of each segment is executed atomically and the CPU time consumption of the segment is annotated using a `CONSUME_CPU_TIME(<time>)` function. This function models the CPU time consumption of the already executed software segment using the notification of a timed event at the estimated end time of the software segment. In case the software segment is interrupted by a higher priority task, the notification of the timed event is delayed accordingly. A similar approach is presented in [RG12].

## 2.5. Challenges and Proposed Solution

Especially for virtual prototype use cases like the development of safety critical embedded systems, driver software development for timing critical peripherals, and verification of RTL code, a realistic timing representation is essential and timing errors arising from simulation modalities have to be avoided. The employment of cycle accurate transaction level models is not suitable as the partitioning of non-atomic transactions into atomic transactions and their successive execution implies a negative impact on the simulation performance. Figure 2.10 (a) illustrates the cycle accurate implementation of a simple example system. The example system consists of the low priority thread $Th_1$ and the high priority thread $Th_2$ each attempting to transfer interleaved three word sized application packets to a single shared resource. According to the cycle accurate

implementation paradigm, the application packets are divided into three successively transferred bus words.

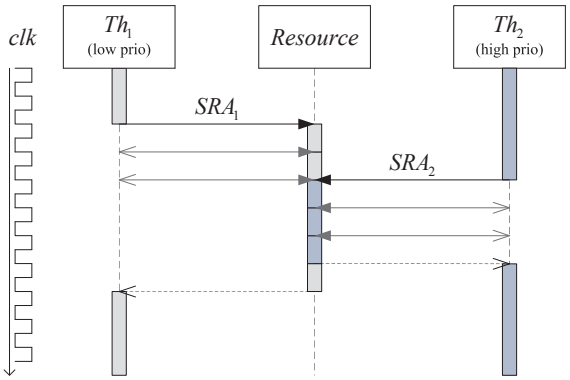
The previously presented simulation performance improvement techniques apparently allow to speed up the simulation of cycle accurate transaction level models, but entail some shortcomings. The parallel execution of cycle accurate models implicates the following drawbacks:

- Due to the computational overhead required for synchronization between the host CPUs, the achieved simulation performance improvement factor lags behind the number of additional CPUs used for the parallel simulation [HBHT08].
- Many techniques used to cope with this scalability problem have a negative impact on the simulation timing accuracy [Jon11, PVSL11].
- The simulation performance improvement factor heavily depends on the characteristics of the modeled system like the ratio between computational and communicational tasks. In case of a low ratio between computational and communicational tasks even a simulation slowdown compared to the execution on a single core is possible [SKR09].
- An additional model partitioning step resulting in equalized computational tasks per host CPU and minimized communicational tasks spanning partition borders is required. This increases the required modeling effort [PVSL11].

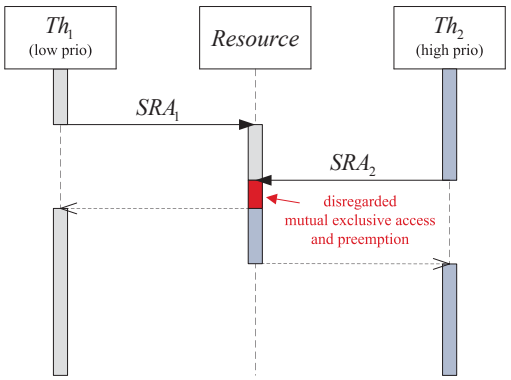
The trace driven simulation acceleration requires additional models implemented at higher or lower abstraction levels resulting in an increased modeling effort [PMG+10].

The employment of non-cycle accurate transaction level models as shown in Figure 2.10 (b) to overcome the low simulation performance of cycle accurate transaction level models implies simulation timing errors. The simulation timing errors arise from disregarding mutual exclusive access to shared resources and disregarding transaction preemption. The previously presented timing accuracy improvement techniques promise simulation timing error reduction while retaining high simulation performance but entail different shortcomings depending on the actual concept. The global quantum mechanism [IEE11] can not guarantee cycle accuracy even for low quantum values. The other approaches either implicate simulation errors like late arriving transactions [RK08] or require distinct models for each accuracy grade implying additional implementation effort [BSS07]. Analytical and statistical resource contention analysis approaches require additional effort for resource contention estimation at model creation time and often result in over estimations [SHRE09] or do not provide cycle accurate timing results due to statistical approximations [Bob07, BPT07, ANMD07].

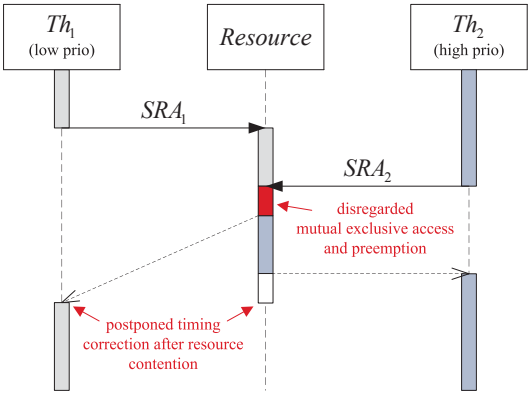
The postponed timing correction in case of resource contention mechanism as shown in Figure 2.10 (c) reduces the timing errors arising from formerly disregarded resource contention effects but does not eliminate them. In addition, this mechanism results in a mismatch between the actual transaction execution and the pretended timing and hence lacks support for data dependencies between interleaved transactions [LMGS12, SD07]. Furthermore, the support for inter-resource dependencies like resources accessing other resources during transaction processing is limited and resource access timing calculations are occasionally restricted to specifying fixed data rates [EES+10].




(a) cycle accurate implementation: application packets are divided into multiple bus words which are successively processed resulting in a simulation slowdown.




(b) non-cycle accurate implementation: non-atomic transactions are processed atomically disregarding mutual exclusive resource access and transaction preemption.




(c) non-cycle accurate with postponed timing correction implementation: erroneous timing arising from non-cycle accurate implementation is revised resulting in a mismatch between transaction execution and pretended timing.

**Figure 2.10.:** Applying TLM implementation alternatives to a simple example system. The simple example system consists of two threads ($Th_1$ and $Th_2$) performing interleaved three word sized non-atomic transactions on a single shared resource.

Therefore, a simulation technique which is devoid of timing errors resulting from the simulation modalities but still provides a high simulation performance is required. In Chapter 3 a SystemC based simulation and modeling technique denoted as Advanced Temporal Decoupling (ATD) is proposed. Advanced Temporal Decoupling provides a set of benefits compared to other established simulation techniques:

- ATD allows for the cycle accurate simulation of mutual exclusive shared resource accesses in the context of temporal decoupling. The simulation timing accuracy achievable by ATD exceeds the simulation timing accuracy resulting from the "postponed timing correction in case of resource conflicts" approach [SD07, EES+10, SBR11].
- ATD provides comprehensive resource and resource access property support. This includes preemption of resource accesses, synchronous and asynchronous resource accesses, and inter-

resource dependencies.

- ATD uses optimal size for processed shared resource access fragments. Thereby, the fragment size is as low as needed to achieve cycle accurate simulation results and as large as possible to achieve high simulation performance. Compared to the fixed and atomic transaction size used in conventional cycle accurate TLM models, this allows for improved simulation performance.

Besides extensive timing accuracy and a high simulation speed, a design flow allowing for the creation of virtual prototypes with low modeling effort is an important requirement in industrial environments. The ease of modeling is influenced by the overall amount of code needed for the creation of a virtual prototype and the availability of appropriate modeling libraries and tools. To enable high versatility and vendor independence, the design flow has to be based on industry standards.

In Chapter 4 a modeling design flow denoted as Transparent Transaction Level Modeling (TTLM) is presented. This design flow consists of a modeling library providing a set of convenience functions and an accompanying code generator. The transparent transaction level modeling library provides means for encapsulating SystemC, TLM-2.0, and ATD specific communication code and provides a set of programming interfaces for various purposes. This allows the designer to focus on the implementation of the actual functionality of the system without requiring SystemC expert knowledge. The transparent transaction level generator is capable of generating the structure of virtual prototypes including the register architecture and the netlist from an IP-XACT specification. Thereby, the generator considers the concepts of ATD and provides variant handling features.

The combination of Advanced Temporal Decoupling and Transparent Transaction Level Modeling forms a framework for the efficient creation of accurate and high-performance virtual prototypes.

# 3. Advanced Temporal Decoupling (ATD)

## 3.1. Fundamental Concept

In the following, the fundamental concept of Advanced Temporal Decoupling (ATD) [HPBG13] will be presented. As already shown, the utilization of conventional non-cycle accurate transaction level models might lead to simulation errors like disregarded mutual exclusive resource accesses and disregarded transaction preemption resulting in erroneous transaction timing. One of the main reasons for these timing errors is the fact, that the data granularity of the transactions has to be determined by the designer at model construction time and hence is fixed at model execution time. Along with the non-interruptibility of transactions due to the cooperative multitasking scheme of the SystemC kernel, this leads to the mentioned timing errors.

In contrast, ATD exploits the look-ahead feature inherent to the temporal decoupling mechanism of SystemC TLM-2.0 to achieve optimal data granularity by providing means to vary the data granularity at runtime. Thereby, optimal data granularity means, that the size of the processed transactions is as small as needed for cycle accuracy and as large as possible for high simulation performance. This is achieved using the simulation modalities shown in Figure 3.1. The ATD simulation modalities are based on and are backward compatible to SystemC simulation modalities illustrated in Figure 2.2. This allows the cooperative usage of ATD aware modules in combination with plain SystemC modules within one simulation.

The ATD simulation modalities consist of two alternating simulation phases denoted as temporal decoupled thread execution phase and transaction processing phase, respectively. The temporal decoupled thread execution phase corresponds to the entirety of delta cycles belonging to one timed notification cycle of the SystemC simulation modalities. During this phase the global simulation time remains constant. In accordance to conventional temporal decoupling, all simulation threads are allowed to run ahead increasing their respective local simulation time and issuing application packet sized transactions which are to be executed by shared resources. Each thread is suspended when reaching a statement comprising inbound data dependency. The transactions issued by the simulation threads are not executed immediately, but are aggregated by Temporal Decoupled Semaphores (TDSems) instead. Temporal decoupled semaphores are integrated into shared resources and assure the correct transaction execution order, mutual exclusive resource accesses, and handle timing effects arising from resource contention and transaction preemption. The aggregation of these transactions does not entail any SystemC event notification. Therefore, the issued transactions are not reflected within the global SystemC event queue.

After all runable threads have been executed during the temporal decoupled thread execution phase, there is no more pending activity at the current global simulation time and the transaction

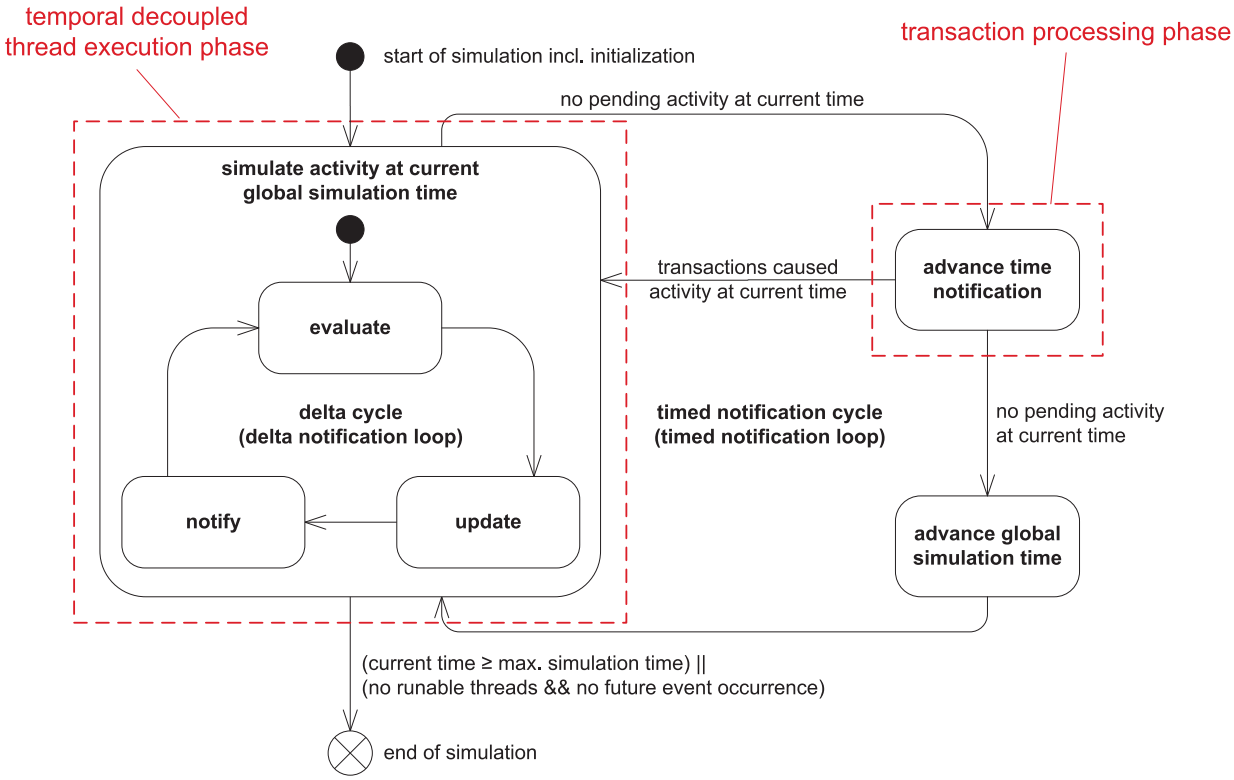



**Figure 3.1.:** ATD simulation modalities: the ATD simulation modalities extend the SystemC simulation modalities by an additional state denoted as advance time notification which is entered prior to advancing the global simulation time and corresponds to the transaction processing phase.

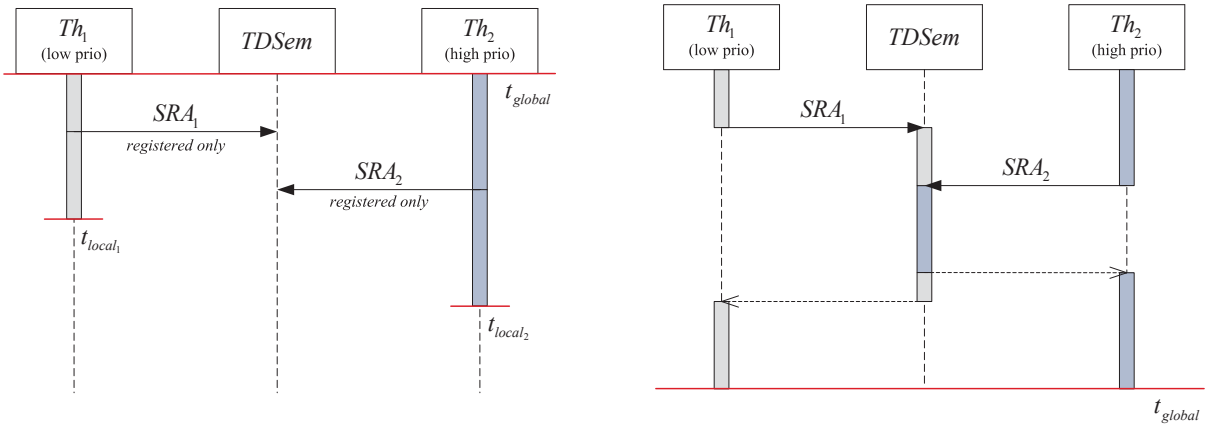
processing phase is started using the advance time notification[1] mechanism. The advance time notification state is added to the SystemC simulation modalities and is entered at the end of the last delta cycle of a timed notification cycle just before the global simulation time advances. During the transaction processing phase, the pending shared resource accesses are processed in the correct temporal order starting with the shared resource access having the lowest start time. As the start times of all pending transactions are known, the amount of time available for the processing of each transaction can be computed and considered during the transaction processing to allow for the preemption of transactions. The amount of simulation time available for the processing of each transaction is denoted as time budget. Due to the cooperative multitasking scheme of SystemC, the preemption of the shared resource access execution after the time budget has elapsed can not be enforced. Instead, the knowledge about the time budget enables the designer to implement cycle accurate transaction preemption.

After all transactions have been processed, the global simulation time is advanced to the time of the next SystemC event and the simulation continues with the next temporal decoupled thread execution phase. In case delta events arose during the transaction processing phase causing activity at the current time, the temporal decoupled thread execution phase is reentered without advancing the global simulation time to allow the newly created activity at current time to be processed.

[1] The implementation of the advance time notification mechanism is shown in Section A.1

Using this simulation mechanism, the virtual prototype is conceptually divided into an independent and a reactive simulation part. The independent simulation part comprises all simulation threads each running ahead in simulation time leaving a sequence of interaction attempts. The reactive simulation part comprises transaction processing functions used to process the pending shared resource accesses with optimal data granularity by exploiting the timing information gathered from the interaction attempt sequences.

In the following, the ATD simulation modalities are applied to the simple example system introduced in Section 2.5. The simple example system consists of two threads performing interleaved three word sized non-atomic transactions on a single shared resource. Figure 3.2 (a) shows the simulation state of the simple example system at the end of the first temporal decoupled thread execution phase. The threads $Th_1$ and $Th_2$ have been executed up to their local simulation times $t_{local_1}$ and $t_{local_2}$, respectively. Both local simulation times are ahead of the global simulation time $t_{global}$. The shared resource accesses denoted as $SRA_1$ and $SRA_2$ have been registered at the $TDSem$ taking their respective start times into account. At this stage, a time span ranging from $t_{global}$ to the lowest local simulation time of any thread ($t_{local_1}$ in Figure 3.2 (a)) has emerged, where the behavior of all threads composing the independent simulation part is known. Thus, the state of the system at the end of this time span only depends on the timing and results of the currently unprocessed shared resource accesses starting within this time span.




**(a)** After the first temporal decoupled thread execution phase: the threads have been running ahead of the global simulation time $t_{global}$ registering shared resource accesses ($SRA_1$ and $SRA_2$) at the temporal decoupled semaphore $TDSem$.

**(b)** After the first transaction processing phase: as the start times of all pending shared resource accesses have become known during the temporal decoupled thread execution phase, the SRAs can be processed with optimal data granularity.

**Figure 3.2.:** Applying Advanced Temporal Decoupling to the simple example system.

Figure 3.2 (b) shows the result of the ATD-based processing of the simple example which is identical to the result of the cycle accurate implementation shown in Figure 2.10 (a). The execution of the pending shared resource accesses starts with the partial execution of the three word sized $SRA_1$. As the start time of $SRA_2$ is already known to be two words after the start time of the first shared resource access, the time budget for the execution of $SRA_1$ equals the amount of time needed to process two out of three words. The time budget for the following execution of $SRA_2$ is not limited as there are no higher priority shared resource accesses. Thus, all three words of $SRA_2$

are processed without interruption. After $SRA_2$ is finished, $SRA_1$ is resumed and completes the processing of the pending shared resource accesses. The time consumption of the shared resource access processing is assigned to the thread, which issued the corresponding shared resource access.

In contrast to the cycle accurate implementation, the data granularity of the ATD based implementation is not fixed to bus word sized transactions. Instead, the data granularity is dynamically adapted using the time budget mechanism. Thereby, application packet sized transactions can be divided into multiple, not necessarily equal sized transactions. Therefore, it can be said that the resulting package sizes are as small as needed to achieve cycle accuracy and as large as possible to achieve a high simulation speed. In case there are no interfering SRAs this can be seen as a kind of automatic "fast-forward" mode.

In the following, the different aspects of the Advanced Temporal Decoupling concept will be presented in more detail. At first, in-depth descriptions of the thread model, shared resource accesses, and temporal decoupled semaphores are given. Afterwards, a description of their collaboration during the temporal decoupled thread execution phase and the transaction processing phase is presented.

## 3.2. Thread Model

As stated in Section 2.3, one possibility to model hardware inherent parallelism in a Discrete Event Simulation is the employment of the process-interaction approach [Fis01]. Thereby, a set of threads is used, each executing a sequence of instructions representing the functionality of the corresponding portion of the virtual prototype. The thread execution semantics used in ATD differ from the thread execution semantics used in conventional non temporal decoupled models. However, the ATD implementation ensures the conformance of the ATD based thread execution semantics with the conventional non temporal decoupled thread execution semantics.

### 3.2.1. ATD based Thread Execution Semantics

One of the main goals of temporal decoupling is to improve the simulation performance by reducing the number of context switches compared to conventional non temporal decoupled execution. Therefore, it is desirable to progress the state and the local simulation time of the currently executed thread as far as possible without suspension. This voids the equality of the simulation time between the different simulation processes belonging to the virtual prototype. Simulation processes having lower local simulation times than the currently executed thread are denoted as trailing simulation processes. The inequality of the simulation time renders the internal state of trailing simulation processes undefined if an interaction between a temporal decoupled thread and a trailing simulation process occurs. For instance, if a temporal decoupled thread attempts to access a trailing shared resource, it can not be determined, if the state **interacting with shared resource** can be entered directly or if the state **blocked due to resource contention** has to be entered due to a potential unavailability of the resource.

Therefore, the simulation time consumption state chart used for the conventional non temporal decoupled thread execution presented in Figure 2.7 on page 21 is substituted by the simulation time consumption state chart suitable for ATD based thread execution shown in Figure 3.3. The objective of ATD is to ensure the conformance of these execution models while still allowing for temporal decoupling.

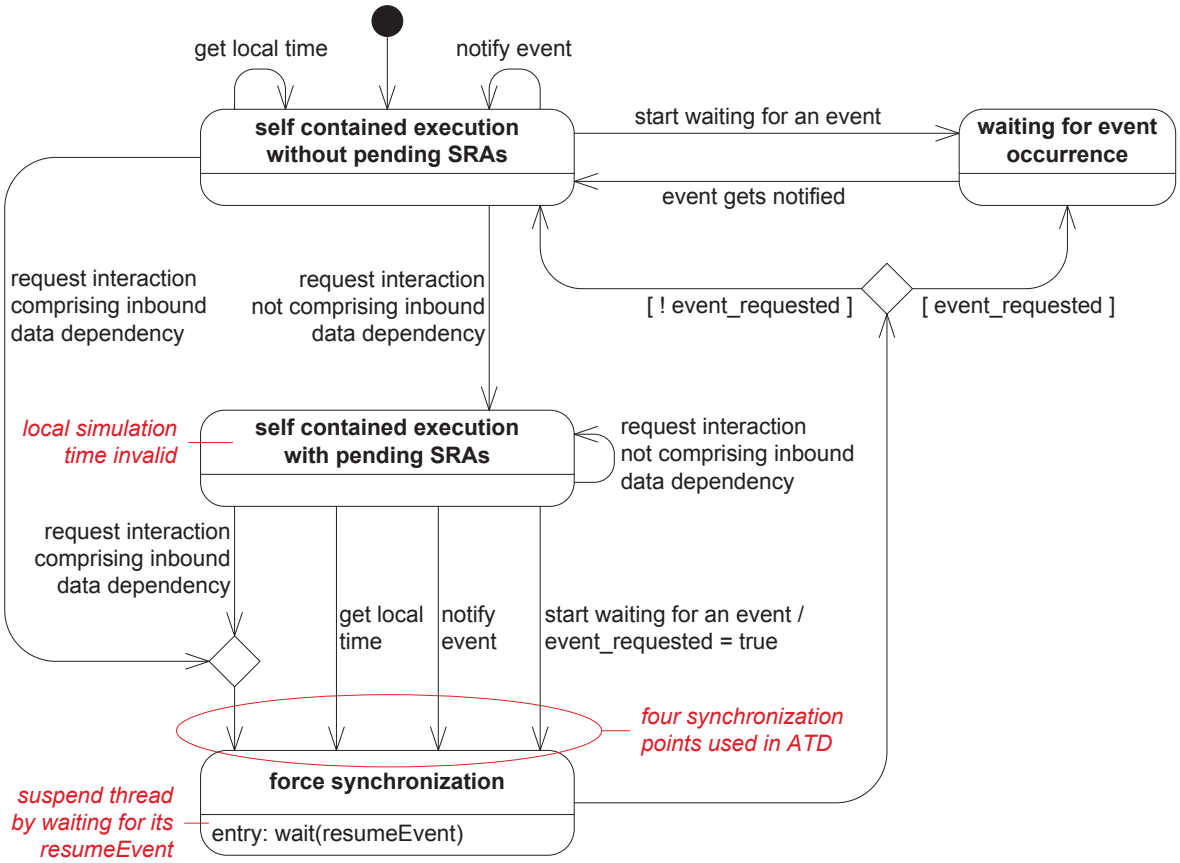



**Figure 3.3.:** Simulation time consumption state chart for ATD based thread execution. As the temporal decoupled thread execution renders the current state of trailing simulation processes undefined, the processing of the interactions has to be deferred until the internal state of the trailing simulation processes is determined during synchronization taking place in the **force synchronization** state.

Compared to the conventional non temporal decoupled case, the initial **self contained execution** state is split into two states denoted as **self contained execution without pending SRAs** and **self contained execution with pending SRAs**, respectively. The initial state of the ATD thread model is the **self contained execution without pending SRAs** state which corresponds to the **self contained execution** state of the conventional non temporal decoupled thread model. The **waiting for event occurrence** state and the transitions from and to the **self contained execution without pending SRAs** state are retained. As interaction attempts with trailing simulation processes have to be deferred due to the currently unknown state of the trailing simulation process, the interaction is only requested. This is done by registering a shared resource access at the temporal decoupled semaphore which is integrated into the trailing

simulation process. If the further behavior of the currently executed thread depends on the result of the requested interaction, an inbound data dependency exists. As SystemC does not provide checkpoint and rollback mechanisms, the temporal decoupling has to be stopped in case such a synchronization point is reached. This is achieved by entering the **force synchronization** state which suspends the temporal decoupled thread to allow the trailing simulation processes to catch up in simulation time and the internal state of the trailing simulation processes to become valid.

If a requested interaction does not comprise inbound data dependency as it might be the case for write accesses to shared resources, the temporal decoupled execution of the thread can be continued without suspension and the state **self contained execution with pending SRAs** is entered. As the duration of the interaction is currently unknown, it must be temporarily disregarded leading to the invalidity of the local simulation time of the current thread. The invalidity of the local simulation time during the **self contained execution with pending SRAs** state implies the need for additional synchronization points for example in case a reading access to the local simulation time occurs as illustrated by the get local time transition in Figure 3.3. Furthermore, synchronization is required in case the thread attempts to notify an event or starts to wait for the occurrence of an event. Table 3.1 summarizes the four synchronization points used in ATD.

| |
|---|
| • request interaction with trailing simulation process comprising inbound data dependency (e.g. read from a shared resource or local volatile variable) |
| • get local simulation time[1] |
| • notify event[1] |
| • start waiting for an event[1] |

[1] only in case there are pending SRAs

**Table 3.1.:** Set of synchronization points used in ATD.

### 3.2.2. Implementation

The thread implementation used in ATD augments the `SC_THREAD` implementation provided by SystemC. Each `SC_THREAD` participating in the ATD based simulation is extended by an object of the `ThreadDescriptor` class provided by ATD. Figure 3.4 shows the simplified class diagram of the `ThreadDescriptor` class.

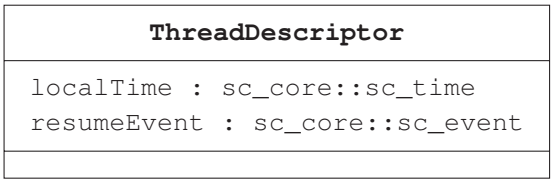



**Figure 3.4.:** Simplified ThreadDescriptor class diagram.

The `localTime` stores the thread's current local simulation time required for the temporal decoupled simulation. The replacement of the global simulation time by multiple local simu-

lation times each associated to one thread requires the adaption of the `wait(<time>)` and `sc_time_stamp()` functions to operate on the local simulation times instead of the global simulation time. The timing API of the Transparent Transaction Level Modeling (TTLM) library presented in Section 4.2.2 provides the `ttlm::sc_ttlm_module` class which is derived from the `sc_module` class and comprises appropriate versions of the mentioned functions.

The `resumeEvent` is used to implement the synchronization after reaching a synchronization point. In case the **force synchronization** state is entered, the thread is suspended waiting for the occurrence of its resume event. As long as the resume event has not been notified, it is not reflected within the SystemC kernel internal event queue and therefore does not influence the value returned by `sc_time_to_pending_activity()`. In this case, the corresponding synchronization point is denoted as a non-influencing synchronization point. The notification of the resume event will take place during the transaction processing phase (refer to Section 3.6) after all interactions requested by this thread have been processed and the occurrence time of the resume event has been calculated. The notification causes the resume event to be part of the SystemC kernel internal event queue which might limit the value returned by `sc_time_to_pending_activity()`. Therefore, the synchronization point is denoted as an influencing synchronization point. The corresponding thread will become runable after the global simulation time has advanced to the calculated time of the resume event occurrence.

In case ATD is used in conjunction with the TTLM library, the synchronization points do not need to be added by the designer as they are integrated into the corresponding library elements.

## 3.3. Shared Resource Access (SRA)

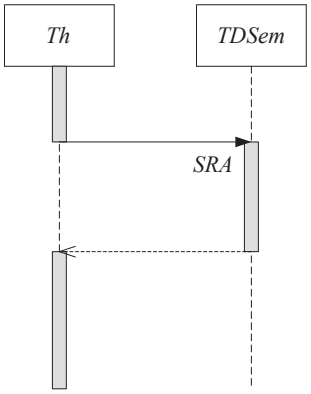
The interaction between an initiating instance like a simulation thread and a shared resource is denoted as SRA. The SRA execution is not accomplished during the execution of the corresponding thread. Instead, the execution of the SRAs is deferred to one of the following transaction processing phases allowing to achieve optimal data granularity.

Despite of this deferment, the results of the ATD based simulation have to correspond to the simulation results that would be achieved when using the conventional non temporal decoupled simulation mechanism. Therefore, a comprehensive set of SRA execution modalities is required.

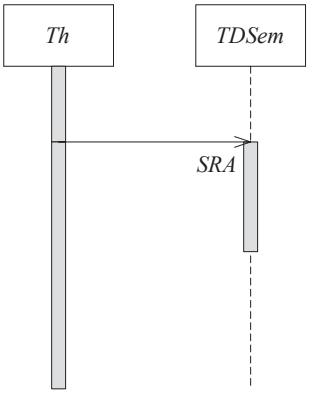
### 3.3.1. Execution Modalities

Figure 3.5 gives an overview of the SRA execution modalities. SRAs can either be processed in a synchronous or asynchronous way and are either preemptable or non-preemptable. Additionally, the processing of an SRA might lead to subordinate interactions with other resources. In this case, the concerned resources are denoted as cascaded resources. Any combination of the SRA execution modalities is supported.

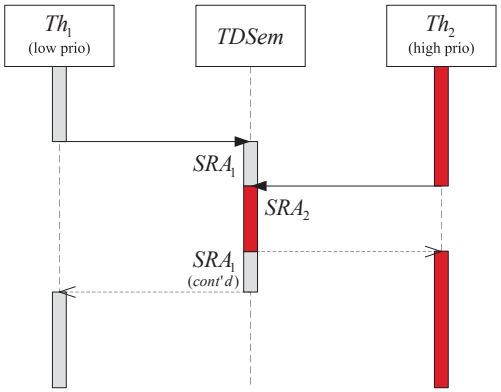
In case of a synchronous access as shown in Figure 3.5 (a), the thread $Th$ is blocked after registering the SRA until the SRA execution has finished and awaits the resource's response. In case of



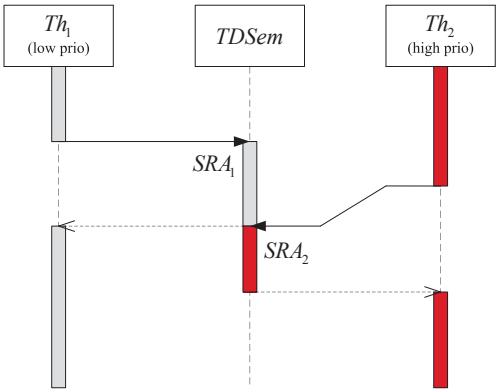

**(a)** *Synchronous execution*: $Th$ is blocked until the SRA execution has finished and awaits the resource's response.



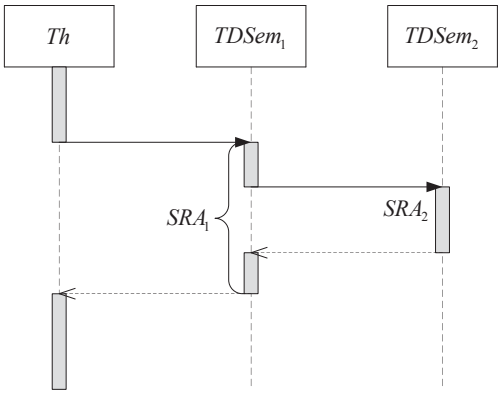

**(b)** *Asynchronous execution*: $Th$ remains unblocked after issuing the SRA.




**(c)** *Preemptable execution*: the preemptable low priority $SRA_1$ issued by $Th_1$ is interrupted by higher priority $SRA_2$ issued by $Th_2$ and resumed at a later point.




**(d)** *Non-preemptable execution*: the non-preemptable low priority $SRA_1$ issued by $Th_1$ defers the execution of the higher priority $SRA_2$ issued by $Th_2$.




**(e)** *Cascaded resources*: the execution of $SRA_1$ leads to the creation of the nested $SRA_2$. There is no limitation on the number of subsequent interactions resulting from the processing of a single SRA and on the nesting depth.

**Figure 3.5.:** Overview of supported shared resource access execution modalities. The actual processing modalities may differ as the suspension of the corresponding threads only depends on the existence of an inbound data dependency. However, ATD re-establishes the correct execution modalities during the transaction processing phase.

an asynchronous access as shown in Figure 3.5 (b), $Th$ does not get blocked after the SRA registration [Man09]. Instead, the execution of the threads functionality is resumed immediately. In other words: asynchronous SRAs cause parallelism as the functionality of the shared resource triggered by the SRA and the behavior of the issuing thread are processed simultaneously. As shown in Section 3.2.1 the ATD based thread execution semantic differs from the conventional non temporal decoupled thread execution semantic. The suspension of a thread due to resource interaction only depends on the existence of an inbound data dependency. This means, if a synchronous SRA does not comprise inbound data dependency, the simulation thread does not get suspended. Instead, the thread (re-)enters the **self contained execution with pending SRAs** state rendering its local simulation time invalid. The synchronous SRA execution semantic of the shared resource access is re-established during the transaction processing phase after the synchronous SRA has been processed.

In case of the preemptable execution as show in Figure 3.5 (c), the low priority $SRA_1$ is preempted by the higher priority $SRA_2$. The number of preemptions per SRA execution is not limited. In case of the non-preemptable execution as shown in Figure 3.5 (d) the low priority $SRA_1$ does not get preempted. Instead, the execution of the higher priority $SRA_2$ is deferred until the execution of $SRA_1$ is finished. As the ATD simulation is based on SystemC which implements a cooperative multitasking execution model [IEE11], preemption can not be enforced by the simulation kernel. For this reason, preemption has to be implemented by the designer employing the time budget mechanism.

As the communication in embedded systems can span multiple resources, ATD supports the concatenation of SRAs by allowing the registration of child SRAs during the transaction processing phase. Figure 3.5 (e) shows a simple example of such cascaded resource accesses. Thread $Th$ accesses $TDSem_1$ by issuing $SRA_1$. The execution of $SRA_1$ leads to the nested $SRA_2$ registered at $TDSem_2$. ATD assures that all timing effects arising from the execution of nested SRAs or resource contentions occurring at any of the cascaded resources are considered. Furthermore, ATD does neither impose any limitation on the number of interactions taking place during the processing of a single SRA nor on the nesting depth of cascaded resource accesses.

### 3.3.2. Shared Resource Access Lifecycle

The SRA execution modalities result in the shared resource access lifecycle shown in Figure 3.6.

The state **SRA registered** is entered after the shared resource interaction has been requested either by a thread during the temporal decoupled thread execution phase or by a superior SRA during the transaction processing phase. The rest of the SRA lifecycle transitions only take place during the transaction processing phase. In case the SRA is selected to be the next one to be executed, the state **SRA executing** is entered and the processing of the transaction takes place. In case the resource supports transaction preemption, the transaction processing involves the consideration of the time budget. If the transaction duration exceeds the time budget, the transaction is partially processed only. After the partial processing has taken place, the state of the SRA changes to **SRA preempted** until the resource contention has disappeared and the SRA is selected to be the next one to be executed again. In case of cascaded resources,

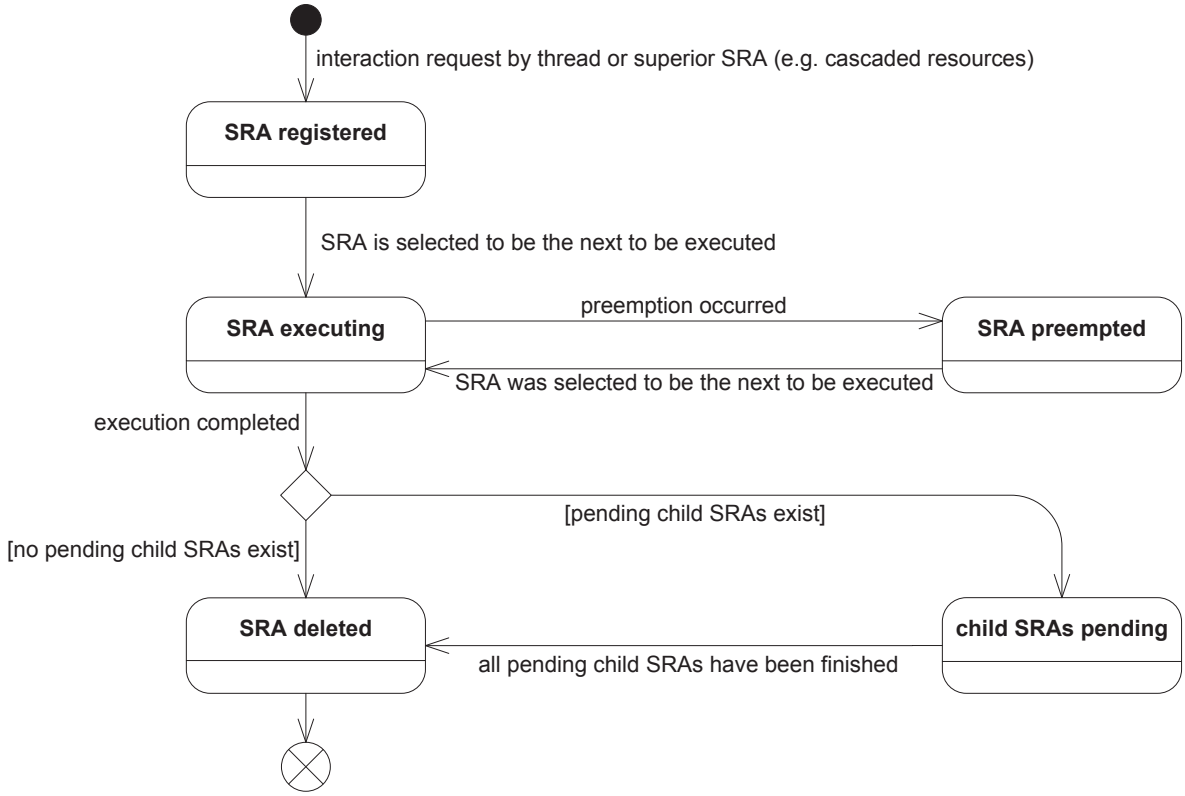



**Figure 3.6.:** SRA lifecycle. The execution of the SRA takes place during the transaction processing phase and might lead to the registration of child SRAs. In case the child SRAs do not comprise inbound data dependency, the parent SRA might be completed before the child SRAs have been processed. If there are pending child SRAs, the parent SRA enters the **child SRAs pending** state.

the SRA execution might lead to the registration of subordinate transactions at other shared resources. Similar to threads being only suspended in case of an inbound data dependency during the temporal decoupled thread execution phase, the execution of SRAs might be completed before all subordinate transactions have been processed. In this case, the SRA enters the **child SRAs pending** state. After the SRA and all possibly existing subordinate SRAs have been completed, the state **SRA deleted** is entered and the SRA is removed.

### 3.3.3. Implementation

SRAs are characterized by a set of attributes as shown in the simplified class diagram given in Figure 3.7. The `startTime` attribute contains the preliminary start time of the SRA which is determined when the registration of the SRA takes place. During the transaction processing phase, the start time attribute might be increased due to the execution of preceding pending SRAs issued by the same SRA source. The start time also might be increased due to the contention of the accessed shared resource caused by other shared resource accesses which is detected during the transaction processing phase. As the start time might only be increased by the mentioned effects, the initial value determined by the SRA source is a lower bound for the actual SRA

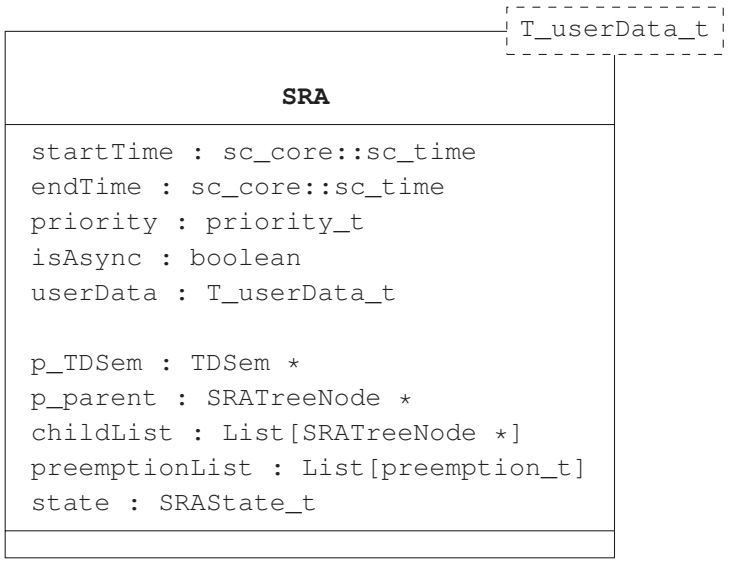



**Figure 3.7.:** Simplified SRA class diagram.

start time. The `endTime` attribute represents the end time of the SRA which is determined during the transaction processing phase. The `priority` can be used to determine the processing order of simultaneous accesses. The scheduling policy used to determine the processing order of simultaneous SRAs pending at the same shared resource can be defined by the designer. For more details refer to Section 3.4. The `isAsync` flag determines whether, the SRA is to be processed in an asynchronous or a synchronous way. The data which is to be processed during SRA execution can be passed to the SRA executor using the `userData` attribute. The data type is determined by the `T_userData_t` template parameter. For SystemC TLM-2.0 based shared resources, the user data typically is of type `tlm::tlm_generic_payload`.

Beside the previous attributes deduced from the SRA registration, there are additional attributes required for the SRA management during the transaction processing phase. The `p_TDSem` attribute points to the TDSem instance integrated into the accessed shared resource. `p_parent` points to the SRA source being the simulation process which registered the SRA. This might either be a thread or a superior SRA executed by a cascaded shared resource. To support the polymorphism of the SRA source, the `ThreadDescriptor` and `SRA` classes have a common base class named `SRATreeNode`. The `childList` attribute contains pointers to all SRAs issued during the execution of the current SRA. In conjunction with `p_parent`, this attribute is used to implement the SRA tree (refer to Section 3.4.2.1). The `preemptionList` contains an ordered set of previously occurred preemptions of the current SRA which can be used for debugging and tracing purposes. Each preemption occurrence is represented by a pair of `preemptionTime` and `resumeTime` attributes. The `state` attribute contains the SRA's current state within its lifecycle as shown in Figure 3.6.

## 3.4. Temporal Decoupled Semaphore (TDSem)

One of the purposes of conventional semaphores is the assurance of mutual exclusive shared resource access for individual threads in parallel applications [Dij65]. Threads attempting to access the shared resource try to get a lock on the semaphore. On success, the thread holding the lock has exclusive access to the shared resource. Other threads attempting to access the shared resource while it is locked are suspended and have to wait until the resource becomes available again. In ATD, shared resource accesses are managed using Temporal Decoupled Semaphores (TDSems) which are integrated into the shared resources. TDSems provide mutual exclusive shared resource access functionality in temporal decoupled SystemC TLM-2.0 models. In contrast to conventional semaphores, TDSems do not block temporal decoupled threads that attempt to get access to the shared resource. Instead of getting a lock on the TDSem, temporal decoupled threads access shared resources by registering future SRAs at the TDSem integrated in the corresponding shared resource.

### 3.4.1. Shared Resource Access Registration

The SRA registration takes place using the `registerAccess_if` interface shown in Figure 3.8.


**Figure 3.8.:** Interface used for SRA registration.

The `delay` parameter specifies the start time of the SRA. In case the SRA is registered by a thread during the temporal decoupled thread execution phase, the start time is relative to the local simulation time of the thread and corresponds to the delay parameter used in conventional SystemC TLM-2.0 models. In case the SRA is registered by a superior SRA during the transaction processing phase, the start time is relative to the start time of the superior SRA. The remaining parameters `priority`, `userData` and `isAsync` are stored in the newly created SRA object and are processed during the transaction processing phase.

The algorithm used for the registration of SRAs is shown in Listing 3.1. The number of pending SRAs registered by one thread without synchronization taking place can be limited by the $SRA_{quantum}$. This is done to reduce the amount of memory required for the simulation as the SRA registration leads to a deep copy of the user data. This deep copy is required to preserve the user data for processing during the transaction processing phase in case the temporal decoupled thread reuses the user data container to register further SRAs. In contrast to the global time quantum used in TLM-LT simulations [IEE11], the value of $SRA_{quantum}$ does not influence the

```
1 create SRA object
2 add SRA to corresponding SRA tree //refer to Section 3.4.2.1
3 add SRA to corresponding SRA matrix cell //refer to Section 3.4.2.2
4 if (number of pending SRAs registered by thread Th_i) ≥ SRA_quantum
5   Th_i->sync()
```

**Listing 3.1:** SRA registration during SRA creation phase.

simulation accuracy. Instead, the $SRA_{quantum}$ provides a trade-off between the number of context switches and the memory footprint of the simulation.

## 3.4.2. Shared Resource Access Management

### 3.4.2.1. Shared Resource Access Tree

SRA trees are used to efficiently manage cascaded resource accesses by reflecting parent-child relationships between SRAs and threads. Figure 3.9 shows an exemplary SRA tree.

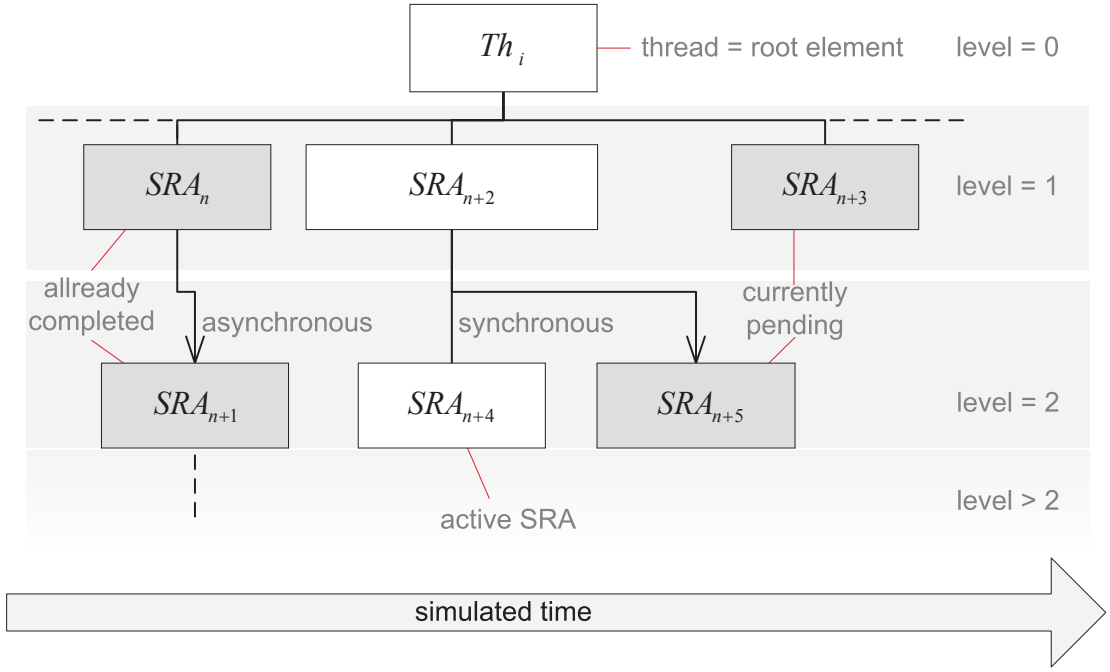



**Figure 3.9.:** Exemplary SRA tree used for inter-SRA dependency management. Asynchronous SRAs are indicated by an arrow pointing to the asynchronous SRA, synchronous SRAs don't have an arrow.

There is one SRA tree ${}^{Th_i}\mathbf{T}$ per simulated thread $Th_i$. Each thread $Th_i$ is the root element of its SRA tree (level 0 element). Each tree element contains an arbitrary sized list of SRAs issued by this tree element during its execution. Level 1 elements are directly created by $Th_i$ during the temporal decoupled thread execution phase, higher level elements originate from SRAs and are

created during the transaction processing phase. The list of SRAs registered by the parent SRA tree element $p$ is denoted as child list ${}_p\mathbf{cl}$ and defined as

$${}_p\mathbf{cl} = \left\{ x \in {}^{Th_i}\mathbf{T} | x \text{ has been registered by } p \right\}$$

${}_p\mathbf{cl}$ is a strictly weak ordered set of SRA elements sorted by the SRA start time $t_{start}$ in ascending order. The strict weak ordering by the ascending start time results from the sequential processing of the parent element $p$ and reflects the causality between successive child elements denoted as inter-child causality:

**Lemma 3.4.1** *Inter-child causality:*

$${}_p\mathbf{cl}\left[i\right].t_{start} \leq {}_p\mathbf{cl}\left[i+1\right].t_{start} \qquad (0 \leq i < |{}_p\mathbf{cl}| - 1)$$

Due to the deferment of the SRA execution to the transaction processing phase, the inter-child causality has to be assured by the ATD implementation. Similar, the ATD implementation has to assure the causality between the parent element $p$ and its child elements ${}_p\mathbf{cl}$, stating that the start time of all child elements is greater than or equal to the start time of the $p$ element:

**Lemma 3.4.2** *Parent child causality:*

$$\forall x \in {}_p\mathbf{cl} \quad : \quad p.t_{start} \leq x.t_{start}$$

As ATD supports synchronous and asynchronous resource accesses, any child list may contain synchronous and asynchronous elements. In Figure 3.9, asynchronous SRAs are marked with an arrow pointing to the respective asynchronous SRA. The subset of synchronous child SRAs ${}_{p,sync}\mathbf{cl}$ is defined as

$${}_{p,sync}\mathbf{cl} = \{x \in {}_p\mathbf{cl} | x \text{ is to be executed in synchronous way}\}$$

Concerning synchronous SRAs, there are additional constraints regarding the end time $t_{end}$ of the SRA. The start time of the next synchronous or asynchronous SRA within the same child list has to be greater than or equal to the synchronous SRA's end time:

**Lemma 3.4.3** *synchronous child exclusivity:*

$$\forall x \in {}_{p,sync}\mathbf{cl} \quad : \quad x.t_{end} \leq {}_p\mathbf{cl}\left[(x. < position\ in\ {}_p\mathbf{cl} >) + 1\right].t_{start}$$

In addition, all synchronous SRAs have to be finished before the parent element $p$ is finished:

**Lemma 3.4.4** *synchronous child inclusion by parent:*

$$\forall x \in {}_{p,sync}\mathbf{cl} \quad : \quad x.t_{end} \leq p.t_{end}$$

As SRA trees consist of threads and SRAs, both `ThreadDescriptor` and `SRA` classes inherit from the `SRATreeNode` class, which provides a common interface implemented by all SRA tree elements. Figure 3.10 shows a simplified class diagram. The `addChild(..)` and `getParent()` functions are used to build up the SRA tree and to navigate towards the root of the SRA tree, respectively. The abstract *`incrementEndTime(..)`* function is used during the transaction processing phase to propagate the time consumption after an SRA has been processed.

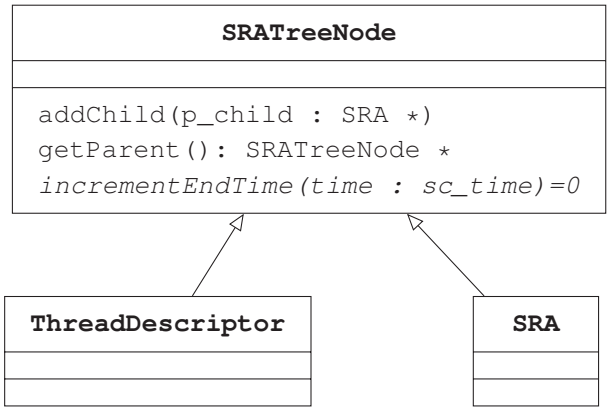



**Figure 3.10.:** Simplified SRATreeNode base class diagram.

#### 3.4.2.2. Shared Resource Access Matrix

The SRA matrix is used to detect and handle resource contention and is shown in Figure 3.11. Each column corresponds to one simulated thread $Th_i$ whereas each row corresponds to one temporal decoupled semaphore $TDSem_j$. Each cell corresponds to the set ${}^{Th_i}\mathbf{SRA}_{TDSem_j}$ which contains all pending SRAs registered by the thread $Th_i$ or any of its child SRAs at the temporal decoupled semaphore $TDSem_j$.

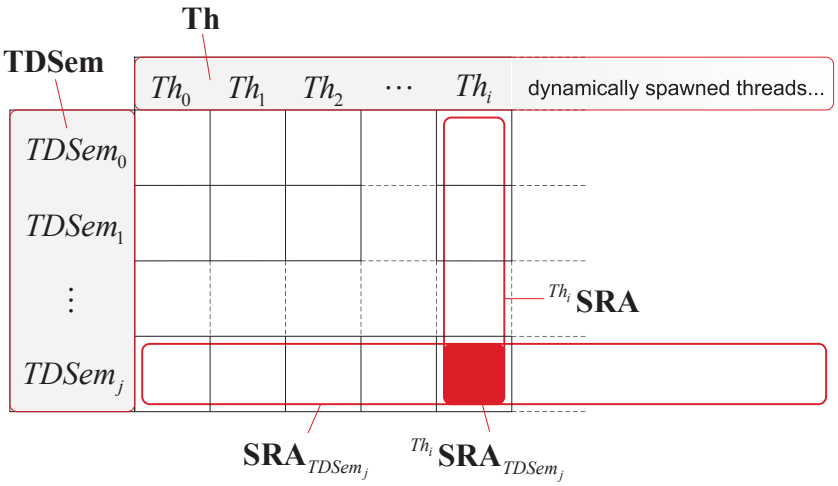



**Figure 3.11.:** SRA Matrix used for resource contention detection.

As the structure of the simulation model is fixed, in the sense that resources can not be added dynamically during simulation, the number of rows remains unchanged during the simulation. In contrast, the number of columns can be extended during simulation by dynamically spawning threads.

The set of all SRAs registered at $TDSem_j$ is defined as

$$\mathbf{SRA}_{TDSem_j} = \bigcup_{0 \leq x < |\mathbf{Th}|} {}^{Th_x}\mathbf{SRA}_{TDSem_j}$$

Similarly, the set of all SRAs registered by thread $Th_i$ or any of its child SRAs is defined as

$$^{Th_i}\mathbf{SRA} = \bigcup_{0 \leq x < |\mathbf{TDSem}|} {}^{Th_i}\mathbf{SRA}_{TDSem_x}$$

The filling of the SRA matrix primarily takes place during the temporal decoupled thread execution phase described in Section 3.5. The resource contention detection and handling takes place during the transaction processing phase described in Section 3.6.

### 3.4.3. Implementation

Figure 3.12 shows the simplified class diagram of the TDSem class which implements the `registerAccess_if` interface.

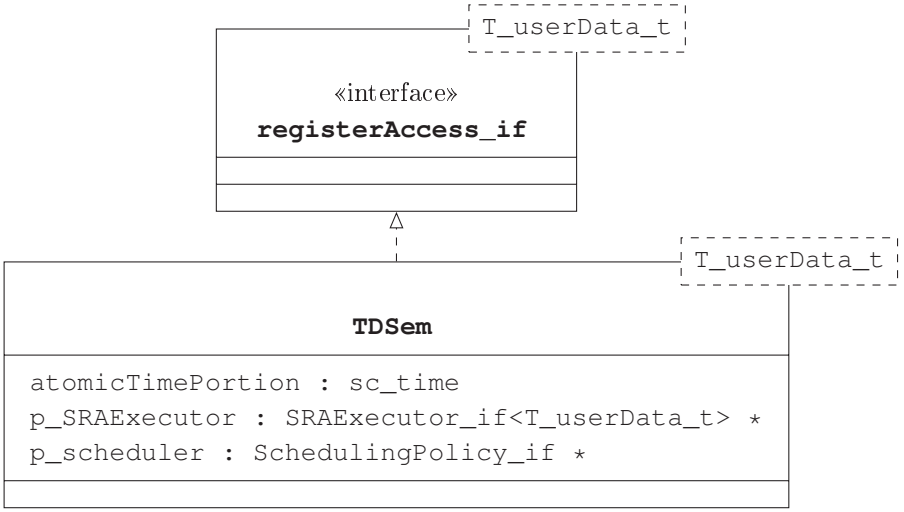



**Figure 3.12.:** Simplified TDSem class diagram.

The `atomicTimePortion` attribute specifies the minimum amount of time available for the non interruptible processing of preemptable SRAs which have the same start time. Typically, this amount of time corresponds to one clock period of the clock domain the resource belongs to. The `atomicTimePortion` will be used during the transaction processing phase for calculating the time budget (see Section 3.6.3).

The execution of pending SRAs is done calling the `p_SRAExecutor`, which has to be implemented by the designer adhering to the `SRAExecutor_if` interface shown in Figure 3.13.

The `pSRA` parameter of the `SRAExecutor_if` points to the SRA which is to be processed. The `duration` parameter is used to return the amount of simulation time consumed for the processing of the SRA. The `timeBudget` parameter specifies the amount of time available for the processing of the SRA without getting in conflict with any successive SRA. This information can be used to implement the preemption of SRAs. The calculation of the time budget is described in 3.6.3. The return value is used to indicate whether the SRA processing has

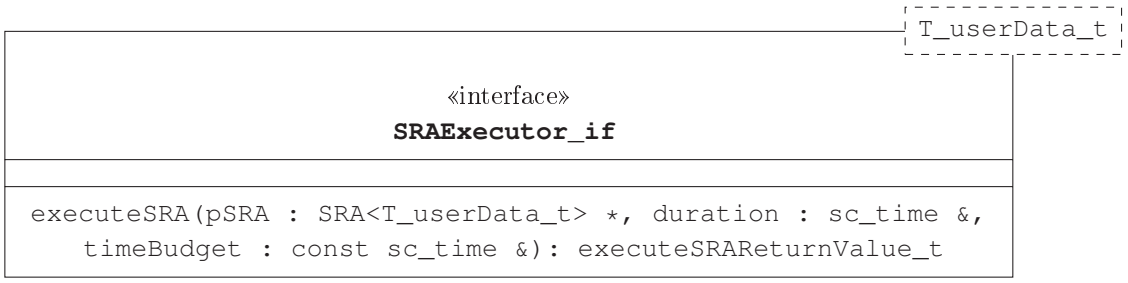



**Figure 3.13.:** SRAExecutor_if class diagram.

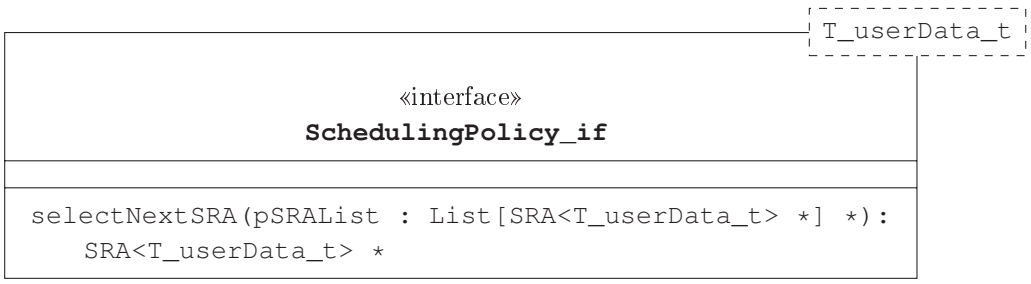



**Figure 3.14.:** SchedulingPolicy_if class diagram.

been completed (`SRA_EXECUTION_OK`) or has been preempted due to insufficient time budget (`SRA_EXECUTION_PREEMPTED`).

The `p_scheduler` attribute of the `TDSem` class points to the user defined scheduler which is invoked in case there are multiple SRAs having the same start time or resume time after a preemption has occurred. The user defined scheduler implements the `SchedulingPolicy_if` interface shown in Figure 3.14. The `pSRAList` is created during the next SRA selection procedure of the transaction processing phase described in Section 3.6.2 and contains the set of SRAs having the same start time or resume time after preemption. The implementation of the `selectNextSRA` function has to return one of the list entries according to the intended scheduling policy. The ATD implementation provides a default priority based scheduling policy.

## 3.5. Temporal Decoupled Thread Execution Phase

During the temporal decoupled thread execution phase, all runable threads are executed according to the ATD based thread execution semantic presented in Section 3.2.1. Thereby, occurring shared resource accesses are registered by temporal decoupled semaphores and processed during the transaction processing phase.

Due to the differences between the ATD based thread execution semantic and the conventional non temporal decoupled thread execution semantic,, special care has to be taken considering deadlocks.

### 3.5.1. Deadlock detection

A deadlock characterizes a constellation of a set of concurrent threads, where "each [thread] in the set is waiting for an event that only another [thread] in the set can cause" [Tan09]. For a deadlock to exist, all of the following four conditions must hold [CES71]:

1. resources are granted exclusively to the requesting thread
2. resources can not be forcibly removed from the corresponding thread
3. threads may acquire additional resources while holding exclusive access to other resources
4. a circular chain of threads exists where each thread has been granted exclusive access to at least one resource that is requested by another thread

As stated in Section 2.4.2.3 conventional temporal decoupled simulation might lead to erroneously disregarding mutual exclusive access to shared resources. This neglects the first condition for a deadlock to occur. Therefore, conventional temporal decoupled simulation might lead to undetected deadlocks.

As ATD adopts the concept of temporal decoupling, a detection mechanisms for deadlocks is required. To allow for cycle accurate simulation results, ATD does neither change the execution order of the shared resource accesses compared to the behavior of the real system, nor their start and end times with respect to the simulation time. For simulation performance reasons and to allow for the cycle accurate simulation of preemptions of the shared resource accesses, the processing of the shared resource accesses is deferred with respect to the wall clock time to the transaction processing phase. This means, that ATD does not add deadlocks to the behavior of the modeled system.

As the purpose of temporal decoupled semaphores is to provide mutual exclusive access to shared resources, the first condition is satisfied. Even though ATD supports preemptable resource accesses, the preemption can not be forced by the SystemC kernel but has to be implemented by the designer by adhering to the calculated time budget and cooperatively aborting the processing of the shared resource access. Hence, the second condition is fulfilled, too. The third condition is fulfilled due to the support of cascaded resources stating that the processing of a shared resource access might lead to subsequent accesses to other shared resources. To check whether a deadlock is present, the deadlock detection algorithm shown in Listing 3.2 is executed each time a cascaded SRA gets registered. This algorithm checks whether a circular chain of resource accesses exists. In case a circular chain of resource accesses is found, the forth condition is fulfilled and the existence of a deadlock is proven.

The temporal decoupled semaphore which is integrated into the shared resource that is to be accessed during the execution of an SRA is denoted as $TDSem_{dest}$. The shared resource which conducts the access is denoted as $TDSem_{src}$. The deadlock detection algorithm starts by calling the `checkForDeadlock(..)` member function of the $TDSem_{src}$ object. This function inspects the respective parent element $p$ of all SRAs currently pending at $TDSem_{src}$. In case $p$ is a synchronous SRA, it is checked whether the temporal decoupled semaphore which executed $p$ matches $TDSem_{dest}$. If this is the case, a circular shared resource access dependency is detected proving the existence of a deadlock. In case $p$ has been executed on another temporal

```
1  TDSem_src = resource which attempts to access TDSem_dest
2  TDSem_dest = resource which is to be accessed during SRA execution
3  TDSem_src.checkForDeadlock(TDSem_dest)
4
5  boolean TDSem::checkForDeadlock(TDSem_dest) {
6    ∀x ∈ SRA_TDSem_src {
7      while (x.p is a synchronous SRA) {
8        if (x.p.<accessed TDSem> == TDSem_dest)
9          return true // DEADLOCK DETECTED!
10       else
11         return x.p.<accessed TDSem>.checkForDeadlock(TDSem_dest)
12       x = x.p // climb up SRA tree
13     }
14   }
15   return false
16 }
```

**Listing 3.2:** ATD deadlock detection algorithm.

decoupled semaphore than $TDSem_{dest}$, a deadlock comprising more than two temporal decoupled semaphores might still exist. Therefore, the `checkForDeadlock(..)` function of the temporal decoupled semaphore on which $p$ has been executed ($x.p.$<accessed TDSem>) is called. This procedure is repeated traversing towards the SRA tree root until an asynchronous SRA or the root element is reached.

Figure 3.15 shows a simple scenario where a deadlock comprising two temporal decoupled semaphores arises. The threads $Th_1$ and $Th_2$ have registered the synchronous shared resource accesses

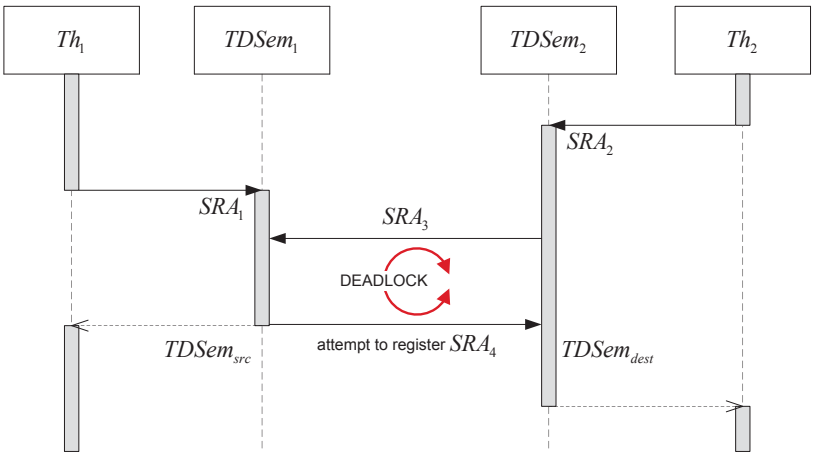



**Figure 3.15.:** Deadlock detection example. The registration of $SRA_4$ at $TDSem_2$ closes the circular resource dependency and leads to a deadlock.

$SRA_1$ and $SRA_2$ at the temporal decoupled semaphores $TDSem_1$ and $TDSem_2$, respectively. The execution of $SRA_2$ has led to the registration of the synchronous $SRA_3$ at $TDSem_1$. The

registration of the synchronous $SRA_4$ at $TDSem_2$ during the execution of $SRA_1$ results in a deadlock. When applying the deadlock detection algorithm during the registration of $SRA_4$, $TDSem_1$ corresponds to $TDSem_{src}$ and $TDSem_2$ corresponds to $TDSem_{dest}$. The set of pending SRAs registered at $TDSem_{src}$ comprises $SRA_3$ only which has been issued by $SRA_2$ which in turn corresponds to $x.p$. As $x.p$ has been executed on $TDSem_2$ which corresponds to $TDSem_{dest}$ the condition in Line 8 of Listing 3.2 holds and the circular resource dependency is detected as expected.

### 3.5.2. Transition to the Transaction Processing Phase

At the end of the temporal decoupled thread execution phase, all runable threads have been executed and are either waiting for the occurrence of an event or have reached a synchronization point. As the duration of pending SRAs is temporarily disregarded during the temporal decoupled thread execution phase, the current value of the local time $t_{local}$ of each thread is a lower bound for the actual simulation time of the corresponding thread. The local simulation time of a thread will become valid after all of its SRAs have been processed. As the global simulation time $t_{global}$ remains unchanged during the temporal decoupled thread execution phase, it can be guaranteed that the local simulation time of all threads is greater than the global simulation time $t_{global}$ returned by `sc_core::sc_time_stamp()`:

$$\forall x \in \mathbf{Th} \quad : \quad x.t_{local} > t_{global}$$

The synchronization points hit by threads executed during the current temporal decoupled thread execution phase are all in a non-influencing state as the corresponding resume events have not been notified, yet. As the resume events associated to non-influencing synchronization points are not contained in the event queue of the SystemC scheduler, the time to pending activity $t_{pa}$ returned by `sc_core::sc_time_to_pending_activity()` is not affected by threads that have reached a synchronization point during the current temporal decoupled thread execution phase. As SRAs are not reflected in the global SystemC event queue, this leads to a situation where pending SRAs having a start time within the $t_{pa}$ time interval might exist. The set of these SRAs is defined as

$$\mathbf{SRA}^{t_{start} < t_{SC\_ev}} = \left\{ x \in \mathbf{SRA} | x.t_{start} < t_{SC\_ev} \right\} \quad \text{where} \quad t_{SC\_ev} = t_{global} + t_{pa}$$

and will be processed during the following transaction processing phase.

## 3.6. Transaction Processing Phase

### 3.6.1. Overview

During the transaction processing phase, all pending SRAs which have a start time lower than $t_{SC\_ev}$ are processed in the correct temporal order. This section gives an overview of the transaction processing phase and its main algorithm which is shown in Listing 3.3. The four main functions of this algorithm are presented in dedicated sections.

```
1  while (SRA^{t_start<t_SC_ev} ≠ ∅) {
2    SRA_next = selectNextSRA(..) // refer to Section 3.6.2
3    t_budget = calculateTimeBudget(..) // refer to Section 3.6.3
4    result = executeSRA(SRA_next, t_duration, t_budget)
5    SRA_next.incrementEndTime(t_duration) // refer to Section 3.6.4
6    if (result == SRA_EXECUTION_PREEMPTED) {
7      SRA_next.state = SRA preempted
8    }
9    if (result == SRA_EXECUTION_OK) {
10     if (SRA_next.cl ≠ ∅) {
11       SRA_next.state = child SRAs pending
12     } else {
13       SRA_next.p.remove(SRA_next) // refer to Section 3.6.5
14     }
15   }
16 }
```

**Listing 3.3:** Main algorithm of the transaction processing phase.

The main algorithm consists of a while loop which is executed as long as there are pending shared resource accesses having a start time lower than $t_{SC\_ev}$. At first, the shared resource access which is to be processed next is identified. This SRA is denoted as $SRA_{next}$. $SRA_{next}$ is selected from $\mathbf{SRA}^{t_{start}<t_{SC\_ev}}$ using the selection algorithm presented in Section 3.6.2. Additionally, the time budget $t_{budget}$ is calculated which specifies the amount of simulation time that can be consumed by the SRA execution without getting in conflict with successive SRAs. The algorithm used for calculating the time budget is presented in Section 3.6.3.

Following to these preparatory steps, $SRA_{next}$ is executed and the processing duration $t_{duration}$ is determined. The execution takes place calling the `p_SRAExecutor` of the corresponding temporal decoupled semaphore. After the execution of $SRA_{next}$ has been completed, $t_{duration}$ is incorporated by calling the `incrementEndTime` function of $SRA_{next}$ which is described in Section 3.6.4.

Finally, the current SRA state is updated (see SRA lifecycle in Figure 3.6). In case the execution of $SRA_{next}$ returned `SRA_EXECUTION_PREEMPTED`, the state of the SRA is changed to **SRA preempted** and the unprocessed part of $SRA_{next}$ remains in $\mathbf{SRA}^{t_{start}<t_{SC\_ev}}$ and will be processed after the preempting SRA has been processed. In case the execution of $SRA_{next}$ has been completed, which is indicated by the return value `SRA_EXECUTION_OK`, the further simulation course depends on the amount of child SRAs contained in the child list of $SRA_{next}$. If the child list contains at least one unprocessed SRA, the state of $SRA_{next}$ is changed to **child SRAs pending** and $SRA_{next}$ remains within its SRA tree. Otherwise, $SRA_{next}$ is removed according to the algorithm described in Section 3.6.5. In case the last SRA issued by the current thread is removed, the resume event of the current thread is notified allowing the thread to leave the **force synchronization** of the ATD thread execution semantics shown in Figure 3.3. For simulation

runtime efficiency reasons, the actual ATD implementation merges some of these algorithms.

### 3.6.2. Next SRA Selection

The algorithm used to select $SRA_{next}$ consists of two steps. During the first step, all SRAs that could safely be executed without violating the *inter-child causality*, the *parent child causality*, the *synchronous child exclusivity*, and the *synchronous child inclusion by parent* Lemmata (see Section 3.4.2.1) are selected from the set of SRA trees and are aggregated in the set $\mathbf{SRA}_{runable}$. During the second step, $SRA_{next}$ is selected from this set involving the user defined scheduler if required.

#### Aggregation of $\mathbf{SRA}_{runable}$

To simplify matters, at first a set of SRA trees is assumed, where each SRA tree consists of synchronous SRAs only, hence Lemmata 3.4.1 to 3.4.4 hold without limitation. To comply with the execution semantics of conventional non temporal decoupled threads stating that at any given time each thread executes exactly one of its statements, at most one of the synchronous SRA tree elements comes into consideration for being added to $\mathbf{SRA}_{runable}$. Due to the inter-child causality stated in Lemma 3.4.1, the child SRAs contained within one child list have to be processed in temporal order. Combining this requirement concerning the temporal order with the synchronous child exclusivity stated in Lemma 3.4.3, only the earliest SRA of the child list being the left most element might be added to $\mathbf{SRA}_{runable}$. Due to the synchronous child inclusion by parent rule stated in Lemma 3.4.4, child elements have to be processed before preempted parent elements are resumed. The algorithm suitable for the aggregation of $\mathbf{SRA}_{runable}$ from a set of SRA trees only consisting of synchronous SRAs is shown in Listing 3.4. The algorithm starts at the root element of each tree and recursively descends to the left most leaf level element of each SRA tree which is then added to $\mathbf{SRA}_{runable}$ if the start time $t_{start}$ of this element is lower than $t_{SC\_ev}$.

In contrast to synchronous SRAs, Lemmata 3.4.3 and 3.4.4 do not hold in case of asynchronous SRAs. This means, that asynchronous SRAs neither inhibit the execution of sibling SRAs nor inhibit the execution of the issuing SRA tree element and therefore temporarily allow for parallelism controlled by the issuing element. Effectively, asynchronous SRAs span new SRA subtrees embedded into the original SRA tree which are to be processed in a parallel manner. This means, that an SRA tree consisting of synchronous and asynchronous SRAs can be conceptually divided into multiple SRA subtrees at asynchronous SRA boundaries. Each resulting SRA subtree consists of synchronous SRAs only and is either rooted on a thread or on an asynchronous SRA. Therefore, for each subtree there is at most one SRA which might be added to $\mathbf{SRA}_{runable}$. The algorithm used for the aggregation of $\mathbf{SRA}_{runable}$ from a set of SRA trees containing asynchronous SRAs is shown in Listing 3.5.

Similar to the algorithm suitable for SRA trees consisting of synchronous SRAs only, this algorithm descends towards the left most element of the SRA tree. In case an asynchronous SRA is reached, the algorithm descends into the corresponding subtree and additionally traverses the sibling elements of the asynchronous SRA until a synchronous SRA is reached (Lines 10-15). All

```
1  aggregate SRA_runable() {
2    foreach Th_i T {
3      getRunableSRAs(Th_i T.rootElement)
4    }
5  }
6
7  // only suitable for SRA trees consisting of synchronous SRAs only!
8  getRunableSRAs(SRATreeNode) {
9    if (this.t_start < t_SC_ev) {
10     if (_p cl ≠ ∅) {
11       getRunableSRAs(_this cl.first)
12     }
13     if (_p cl = ∅) {
14       SRA_runable.add(this)
15     }
16   }
17 }
```

**Listing 3.4:** $\mathbf{SRA}_{runable}$ aggregation algorithm suitable for SRA trees consisting of synchronous SRAs only.

SRAs reached during this procedure which are either leaf level elements or have been preempted and have no more synchronous and unfinished child SRAs are added to $\mathbf{SRA}_{runable}$ (Lines 16-18).

In the following, this algorithm is demonstrated using the exemplary SRA tree shown in Figure 3.16. The SRA tree consists of four SRA (sub)trees rooted on $SRA_1$, $SRA_3$, $SRA_7$, and $Th$, respectively. Due to the inter-child causality stated in Lemma 3.4.1, $SRA_3$ and $SRA_5$ must not be added to $\mathbf{SRA}_{runable}$ as their corresponding synchronous siblings $SRA_2$ and $SRA_4$ have not been completed, yet. Starting from the SRA tree root, the algorithm traverses the child list of $Th$ and descends into the SRA sub tree rooted on $SRA_1$. As this SRA sub tree contains synchronous SRAs only, the left most leaf level element being $SRA_4$ is added to $\mathbf{SRA}_{runable}$. Next the algorithm traverses to $SRA_2$ and descends to the leaf level asynchronous $SRA_7$ which is added to $\mathbf{SRA}_{runable}$. As the child list of $SRA_6$ does not contain any unfinished synchronous SRAs, it is added to $\mathbf{SRA}_{runable}$, too.

#### Selection of $SRA_{next}$

After the aggregation of $\mathbf{SRA}_{runable}$ has been completed, the further course of the transaction processing phase depends on the number of elements contained in this set. If the set is empty, as it is the case if the time of the next SystemC event is lower than the start time of any pending SRA, the transaction processing phase is concluded and the simulation resumes with the following temporal decoupled thread execution phase.

In case $\mathbf{SRA}_{runable}$ is not empty, the elements of the set are sorted by their start time $t_{start}$ in

```
1  aggregate SRA_runable() {
2    foreach Th_i T {
3      getRunableSRAs(Th_i T.rootElement)
4    }
5  }
6
7  //suitable for SRA trees containing synchronous and asynchronous SRAs
8  getRunableSRAs(SRATreeNode) {
9    if (this.t_start < t_SC_ev) {
10     if (_p cl ≠ ∅) {
11       i = 0
12       do {
13         getRunableSRAs(this cl[i])
14       } while (this cl[i++] is to be executed in asynchronous way)
15     }
16     if (_p cl = ∅) || ((_p,sync,unfinished cl = ∅) && (this.state = preempted)) {
17       SRA_runable.add(this)
18     }
19   }
20 }
```

**Listing 3.5:** `getRunableSRAs` implementation suitable for SRA trees containing synchronous and asynchronous SRAs.

ascending order. The lowest start time of all SRAs contained in $\mathbf{SRA}_{runable}$ is denoted as $t_{next}$. The subset of all SRAs starting at $t_{next}$ is denoted as $\mathbf{SRA}^{t_{next}}$:

$$\mathbf{SRA}^{t_{next}} = \{x \in \mathbf{SRA}_{runable} | x.t_{start} = t_{next}\} \tag{3.1}$$

Additionally, all elements of $\mathbf{SRA}^{t_{next}}$ are grouped by their associated temporal decoupled semaphore. The subset of elements contained in $\mathbf{SRA}^{t_{next}}$ which are registered at the same temporal decoupled semaphore $TDSem_j$ is denoted as $\mathbf{SRA}^{t_{next}}_{TDSem_j}$:

$$\mathbf{SRA}^{t_{next}}_{TDSem_j} = \left\{x \in \mathbf{SRA}^{t_{next}} | x.\text{<accessed TDSem>} = TDSem_j\right\} \tag{3.2}$$

After this segmentation procedure, $\mathbf{SRA}_{runable}$ can be divided as illustrated in Figure 3.17. The procedure of selecting $SRA_{next}$ is concluded by choosing $SRA_{next}$ from $\mathbf{SRA}^{t_{next}}_{TDSem_j}$. In case the number of elements contained in $\mathbf{SRA}^{t_{next}}_{TDSem_j}$ exceeds one, $SRA_{next}$ is selected calling the user defined scheduler of $TDSem_j$ using the `SchedulingPolicy_if` interface shown in Figure 3.14.

### 3.6.3. Time Budget Calculation

The time budget being the amount of simulation time available for the processing of $SRA_{next}$ without getting in conflict with any pending SRA or any SystemC event is calculated considering

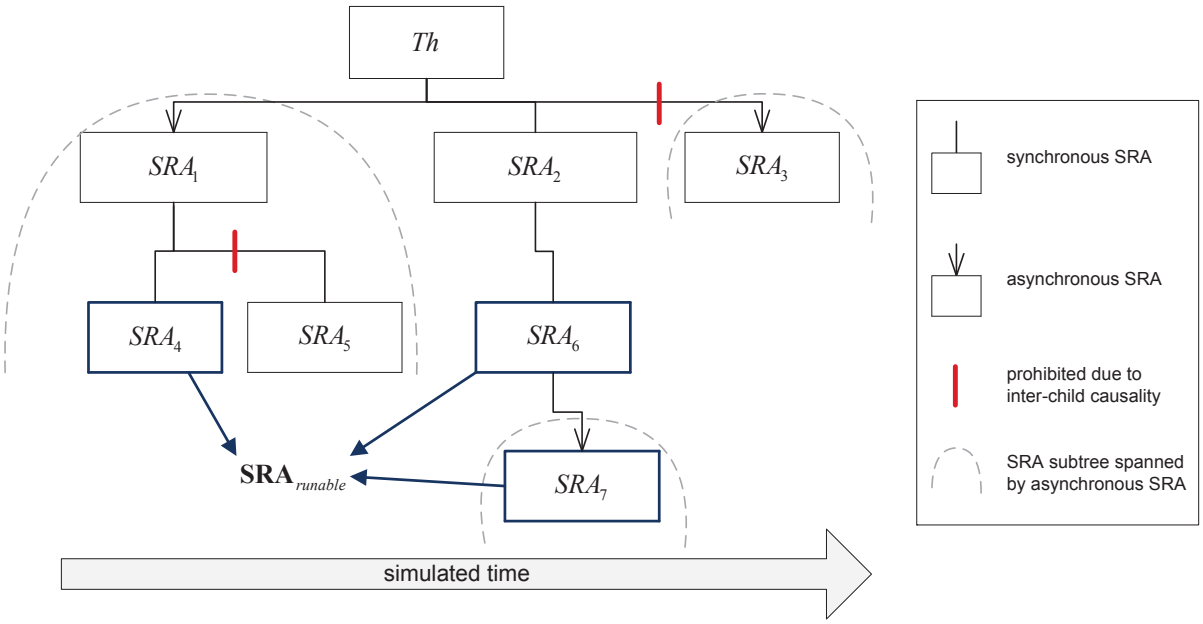



**Figure 3.16.: SRA$_{runable}$** aggregation example.

the classification of **SRA**$_{runable}$ arising during the selection of $SRA_{next}$. Due to the Definitions 3.1 and 3.2, the following condition always holds:

$$\left|\mathbf{SRA}^{t_{next}}_{TDSem_j}\right| \leq \left|\mathbf{SRA}^{t_{next}}\right| \leq \left|\mathbf{SRA}_{runable}\right| \tag{3.3}$$

Depending on the number of elements contained in these sets, the time budget varies according to the algorithm shown in Listing 3.6 where $t_{limit}$ denotes the second lowest start time of all elements contained in **SRA**$_{runable}$.

In case all SRAs having the start time $t_{next}$ are registered at the same $TDSem_j$, the condition in Line 1 holds and the time budget depends on the existence of runable SRAs having a start time greater than $t_{next}$. If there are no runable SRAs having a start time greater than $t_{next}$, the time budget is limited by the time of the next SystemC kernel event occurrence taking place at $t_{SC\_ev}$. Otherwise, the time budget is limited by the second lowest start time $t_{limit}$. In case the SRAs contained in **SRA**$^{t_{next}}$ are registered at different temporal decoupled semaphores, the time budget is limited to the `atomicTimePortion` of $TDSem_j$ being the smallest non-interruptible time portion. Typically this time portion equals to the clock period of the clock domain the corresponding temporal decoupled semaphore belongs to.

The presented time budget calculation algorithm is restrictive. According to that algorithm, the time budget is limited by any event contained in the global SystemC event list and by any pending SRA. Thereby, it does not matter if the pending SRAs which limit the time budget are to be executed on the same shared resource as $SRA_{next}$ or on different shared resources. The restrictive time budget calculation policy allows for a cycle accurate simulation of preemptions of $SRA_{next}$ resulting from the future execution of pending SRAs on a different shared resource. Figure 3.18 illustrates this situation.

Each of the two threads has registered a SRA at a separate TDSem during the temporal decoupled thread execution phase. As the future execution of $SRA_{pending}$ might lead to the creation of

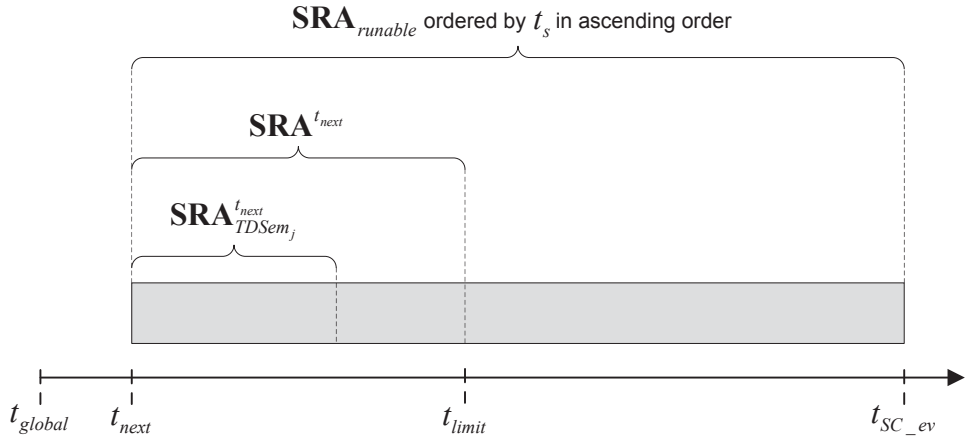



**Figure 3.17.:** Segmentation of $\mathbf{SRA}_{runable}$: $\mathbf{SRA}_{runable}$ is sorted by ascending start time $t_{start}$. The subset of SRAs having the lowest start time $t_{next}$ is denoted as $\mathbf{SRA}^{t_{next}}$. The second lowest start time is denoted as $t_{limit}$ and might influence the time budget available for the execution of $SRA_{next}$. $\mathbf{SRA}^{t_{next}}_{TDSem_j}$ denotes the subset of SRAs which have the lowest start time $t_{next}$ and which are registered at the same temporal decoupled semaphore $TDSem_j$.

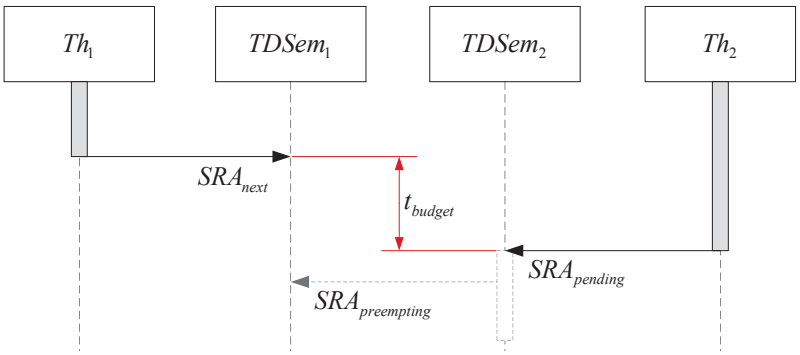



**Figure 3.18.:** Motivation for the restrictive time budget calculation policy. The time budget $t_{budget}$ available for the execution of $SRA_{next}$ is limited by the start time of $SRA_{pending}$ as the execution of $SRA_{pending}$ might lead to the creation of $SRA_{preempting}$ which might in turn preempt $SRA_{next}$.

$SRA_{preempting}$ which might in turn preempt the execution of $SRA_{next}$, the time budget available for the execution of $SRA_{next}$ is precautionary limited by the start time of $SRA_{pending}$.

### 3.6.4. Timing Adjustment

After the execution of $SRA_{next}$ is finished, the resulting duration $t_{duration}$ has to be incorporated. The timing adjustment procedure is started by calling the `incrementEndTime`($t_{duration}$) function of $SRA_{next}$. This function is declared within the `SRATreeNode` class and implemented by the `SRA` and `ThreadDescriptor` classes, respectively (see Figure 3.10). Listing 3.7 shows the algorithm implemented by the `incrementEndTime` function of the `SRA` class. At first, the end time attribute of the SRA is incremented by $t_{duration}$. Next, all siblings are delayed by $t_{duration}$. The set of siblings comprises all unprocessed SRAs contained within the same child list than the

```
1  if |SRA^{t_next}_{TDSem_j}| == |SRA^{t_next}| {
2    // => all SRAs having the start time t_next are registered at TDSem_j:
3    if |SRA^{t_next}| == |SRA_runable| {
4      // => all SRAs contained in SRA_runable have the same start time t_next
5      t_budget = t_SC_ev − t_next
6    } else {
7      // => as SRA_runable contains SRAs having a start time greater than
8      // t_next, the time budget is limited by t_limit:
9      t_budget = t_limit − t_next
10   }
11 } else {
12   // => as there are SRAs having the start time t_next which have been
13   // registered at a different temporal decoupled semaphore,
14   // t_budget is limited to the atomicTimePortion of TDSem_j:
15   t_budget = TDSem_j.atomicTimePortion
16 }
```

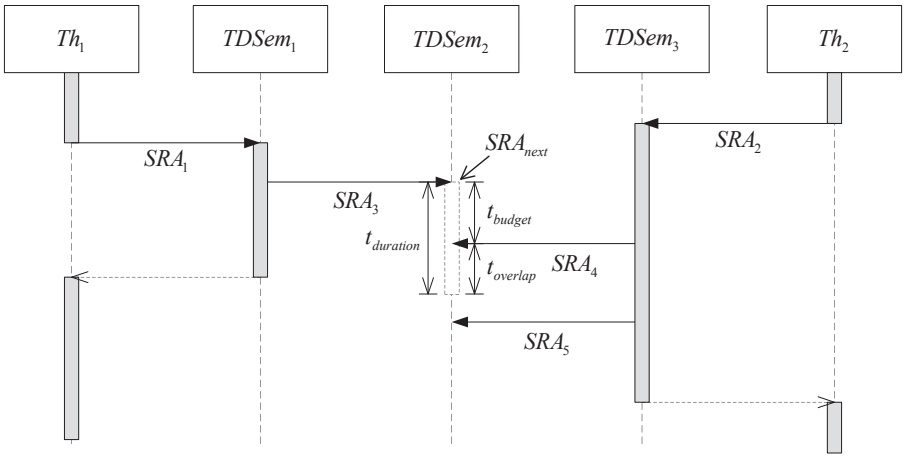
**Listing 3.6:** Time budget calculation algorithm based on the classification of $\mathbf{SRA}_{runable}$ arising during the selection of $SRA_{next}$.

current SRA. In case the SRA is to be executed in a synchronous way, the duration of $SRA_{next}$ is propagated towards the SRA tree root by recursively calling the `incrementEndTime` function of the respective parent element. This leads to the prolongation of the parent element and a delay of the parent's siblings. The recursive duration propagation towards the SRA tree root ends if either the root element or an asynchronous SRA is reached. In case the root element of the SRA tree is reached, the local simulation time $t_{local}$ of that thread is incremented by $t_{duration}$ as shown in Listing 3.8

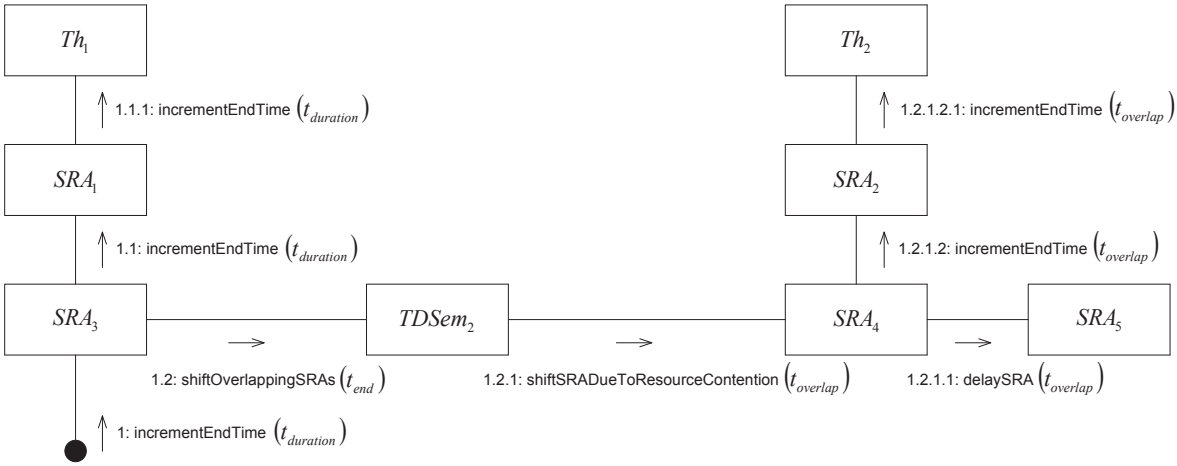
In case the SRA execution duration $t_{duration}$ exceeds the time budget, which might be the case for non-preemptable resource accesses, it has to be checked whether there are overlapping SRAs at any of the corresponding temporal decoupled semaphores at any level within the SRA tree. This is done by calling the `shiftOverlappingSRAs` function of the corresponding temporal decoupled semaphore. This function iterates over the set of currently pending SRAs contained within the SRA matrix row associated to the corresponding temporal decoupled semaphore $\mathbf{SRA}_{TDSem_j}$ and calls the `shiftSRADueToResourceContention` function of each overlapping SRA passing the overlapping time $t_{overlap}$.

At first, the overlapping SRA and all its siblings are delayed by $t_{overlap}$. Next, the overlapping time is propagated towards the root element of the SRA tree the overlapping SRA belongs to. This is done by calling the `incrementEndTime` function of the parent element of the overlapping SRA which in turn entails the check for overlapping SRAs for all parent elements of the overlapping SRA.

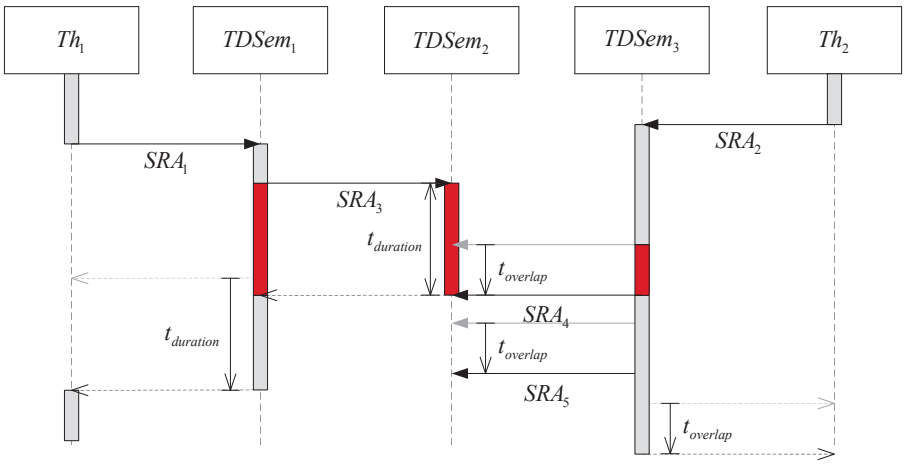
In the following, the timing adjustment procedure is demonstrated using the exemplary simulation constellation shown in Figure 3.19 (a). The considered system consists of two threads and three




**(a)** Simulation constellation right after the processing of $SRA_3$ has taken place. The timing adjustment procedure is used to propagate the duration $t_{duration}$ and the overlapping time $t_{overlap}$ to the affected SRAs and threads.




**(b)** Communication diagram of the timing adjustment procedure. At first, $t_{duration}$ is propagated towards the SRA tree root $Th_1$. Next, the overlapping time $t_{overlap}$ is propagated towards the SRA tree root $Th_2$.




**(c)** Simulation constellation after timing adjustment procedure is finished. The non-preemptive execution of $SRA_3$ affected both threads.

**Figure 3.19.:** Timing adjustment procedure example.

```
1  SRA::incrementEndTime(t_duration) {
2    t_end = t_end + t_duration
3    ∀x ∈ siblings {
4      x.delaySRA(t_duration)
5    }
6    if (!this->isAsync) {
7      // recursion towards SRA tree root:
8      p_parent->incrementEndTime(t_duration)
9    }
10   // call shiftOverlappingSRAs(t_end) which checks if overlapping SRAs
11   // exist and calls shiftSRADueToResourceContention(t_overlap) for all
12   // overlapping SRAs
13   pTDSem->shiftOverlappingSRAs(t_end)
14 }
```

**Listing 3.7:** `incrementEndTime` implementation of the `SRA` class.

```
1  ThreadDescriptor::incrementEndTime(t_duration) {
2    t_local = t_local + t_duration
3  }
```

**Listing 3.8:** `incrementEndTime` implementation of the `ThreadDescriptor` class.

temporal decoupled semaphores. During the preceding simulation course, both threads have been executed during the temporal decoupled thread execution phase and have registered $SRA_1$ at $TDSem_1$ and $SRA_2$ at $TDSem_3$, respectively. Furthermore, both SRAs have already been processed during the current transaction processing phase. The SRA execution took place in a non-preemptable way ignoring the time budget and led to the registration of $SRA_3$, $SRA_4$, and $SRA_5$ at $TDSem_2$. Afterwards, $SRA_3$ has been selected as $SRA_{next}$ and the time budget $t_{budget}$ for its execution has been calculated. Figure 3.19 (a) shows the simulation constellation right after the non-preemptable processing of $SRA_3$ has taken place resulting in the duration $t_{duration}$.

Figure 3.19 (b) shows the communication diagram of the timing adjustment procedure which is initiated by calling the `incrementEndTime` function of $SRA_3$ passing $t_{duration}$. At first, $SRA_1$ is prolonged and then the local simulation time of $Th_1$ is increased by $t_{duration}$. This is done by recursively calling the `incrementEndTime` function of the parent elements of $SRA_3$ (steps 1.1 and 1.1.1). Next, $TDSem_2$ is advised to shift all overlapping SRAs (step 1.2). As $SRA_4$ overlaps, the SRA is shifted due to resource contention by the overlapping time $t_{overlap}$ (step 1.2.1). The shifting of $SRA_4$ results in delaying $SRA_5$ which is the only sibling (step 1.2.1.1). Additionally, $t_{overlap}$ is propagated towards $Th_2$ (steps 1.2.1.2 and 1.2.1.2.1).

Figure 3.19 (c) shows the result of the timing adjustment procedure. The execution duration $t_{duration}$ of $SRA_3$ has been added to $SRA_1$ and $Th_1$. The overlapping time $t_{overlap}$ has been propagated within the SRA tree containing the overlapping $SRA_4$.

```
1  SRA::shiftSRADueToResourceContention(t_overlap) {
2    this->delaySRA(t_overlap)
3    ∀x ∈ siblings {
4      x.delaySRA(t_overlap)
5    }
6    // call incrementEndTime(t_overlap) of parent, which in turn recurses
       towards root of SRA tree containing the overlapping SRA
7    p_parent->incrementEndTime(t_overlap)
8  }
```

**Listing 3.9:** Algorithm used to propagate timing effects arising from resource contention.

Special care has to be taken in case the overlapping SRA is not the first element of a child list. In this case the delay caused by the overlapping is marked as preliminary as the earlier sibling of the overlapping SRA is still pending and its execution duration is not yet known. After the earlier sibling has been executed, the overlapping time is reevaluated and the timing of the overlapping SRA is adjusted accordingly.

### 3.6.5. SRA Deletion

After the execution of $SRA_{next}$ has been completed and the timing adjustment procedure is finished, the SRA is removed from the SRA management data structures. This is done by calling the `remove(`$SRA_{next}$`)` function of the parent element of $SRA_{next}$. In case the parent element of $SRA_{next}$ is an SRA as well, the appropriate algorithm for the deletion of $SRA_{next}$ is shown in Listing 3.10.

```
1  SRA::remove(child) {
2    remove child from corresponding SRA matrix cell
3    thiscl.delete(child)
4    if ((thiscl == ∅) && (state == child SRAs pending)) {
5      // all child SRAs of this SRA have been processed
6      // => remove this one as well:
7      state = SRA deleted
8      p_parent->remove(this)
9    }
10 }
```

**Listing 3.10:** `remove(child)` implementation of the `SRA` class. In case the last pending SRA of the parent's child list is removed and the parent is in state **child SRAs pending**, the SRA deletion continuous towards the SRA tree root.

At first, the $SRA_{next}$ is removed from the corresponding SRA matrix cell. After that, $SRA_{next}$ is deleted from the child list of the parent SRA. In case the child list is now empty and the

parent SRA is in state **child SRAs pending** waiting for the completion of its child SRAs, the state of the parent SRA is changed to **SRA deleted** following the SRA lifecycle presented in Figure 3.6. Additionally, the SRA deletion progresses towards the SRA tree root by calling the `remove(this)` function of the parent's parent element and the corresponding SRA tree branch is successively removed.

Listing 3.11 shows the `remove(child)` function of the `ThreadDescriptor` class which is called in case the SRA deletion reaches the root element of the SRA tree. As before, the `child`

```
1 ThreadDescriptor::remove(child) {
2   remove child from corresponding SRA matrix cell
3   this.cl.delete(child)
4   if (this.cl == ∅) {
5     resume_event.notify(t_local − t_global)
6   }
7 }
```

**Listing 3.11:** `remove(child)` implementation of the `ThreadDescriptor` class. In case the last pending SRA is removed, the resume event is notified to occur at $t_{local}$ and the thread will be resumed after $t_{global}$ has caught up.

element is removed from the corresponding SRA matrix cell and from the child list of the thread. In case the `child` element has been the last pending SRA issued by this thread, the child list of the thread becomes empty. If this is the case, the synchronization point hit by the thread during a preceding temporal decoupled thread execution phase is turned from the non-influencing state into the influencing state by notifying the thread's resume event to occur at $t_{local} - t_{global}$. In this way, the thread will again take part in the temporal decoupled thread execution phase starting after the global simulation time has caught up to the thread's local simulation time.

## 3.7. Basic Benchmarks

In the following, the simulation performance improvement achievable by ATD compared to cycle accurate transaction level modeling is investigated using the simple example system presented in Section 3.1. The example consists of two threads accessing a single shared resource. During each iteration, each thread transmits one application packet consisting of an adjustable number of words. As shown in Figure 3.20, the high priority application packet issued by thread $Th_2$ starts two clock cycles after the low priority application packet issued by thread $Th_1$.

Listings B.1 and B.2 on pages 110–113 show the conventional cycle accurate transaction level modeling and the ATD based implementation of the simple example system, respectively. The threads used in the conventional cycle accurate transaction level modeling implementation transfer the application packets on a word-by-word basis. The arbitration procedure is done using an additional `arbitrate()` thread, which selects the highest priority one word sized packet. Subsequently, the selected one word sized packet is processed and the duration is applied calling `sc_core::wait(CLOCK_PERIOD)`.

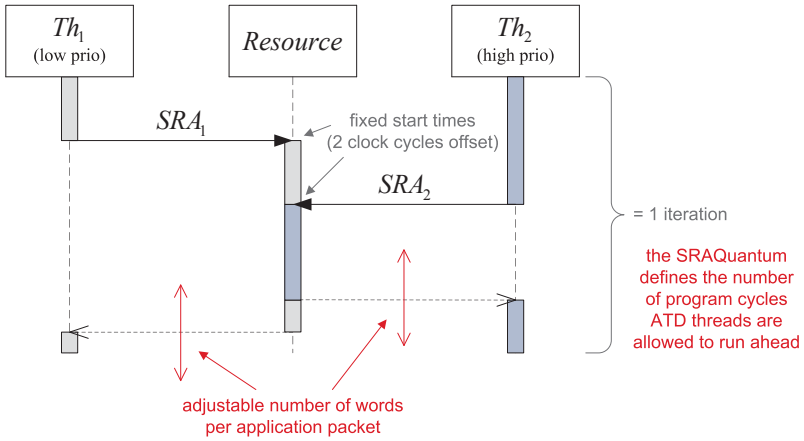



**Figure 3.20.:** The simple ATD benchmark system allows the variation of the number of words per application packet and the number of iterations ATD threads are allowed to run ahead without suspension.

The threads used in the ATD based implementation transfer their application packets en-bloc. Due to the time budget calculation mechanism provided by ATD, the `PreemptiveExecutor` is able to vary the data granularity of the application packet processing to achieve high simulation speed and optimal data granularity for cycle accurate behavior. Additionally, the ATD based implementation makes use of temporal decoupling by allowing each thread to execute multiple iterations without suspension. The maximum number of pending SRAs and thus the number of iterations executed without suspension is defined by the value of the $SRA_{quantum}$.

In the following, the simulation performance implications of different influencing factors are analyzed. At first, the effect of temporal decoupling on the performance of the ATD based implementation is investigated by varying the $SRA_{quantum}$. The number of words per application packet is fixed to three words leading to a preemption of the low priority access by the higher priority access after the first two words of the low priority access have been processed. Figure 3.21 shows the relative speed up achieved by the ATD implementation compared to the TLM-CA implementation.

In case temporal decoupling is disabled forcing the threads to suspend after each SRA registration by using a $SRA_{quantum}$ of one, a performance improvement of approximately 66% is achieved. The speed up rises to approximately 80% if the $SRA_{quantum}$ is chosen between two and five allowing temporal decoupling for up to five iterations. In contrast to inaccurate conventional temporal decoupled models, the speed up does not increase monotonously with the extend of temporal decoupling. Instead, the speed up achieved by ATD decreases if the $SRA_{quantum}$ is increased above five. This is due to an increasing number of SRAs which have to be updated during the timing adjustment procedure. Therefore, the $SRA_{quantum}$ does not only limit the memory footprint of the simulation but can also be used to achieve optimal performance by avoiding a performance penalty due to too many pending SRAs.

After investigating the implications of the $SRA_{quantum}$, the effect of the application packet size is analyzed. Figure 3.22 shows the number of application packets processed per second (wall clock time) contrasting the conventional TLM-CA implementation and the ATD implementation.

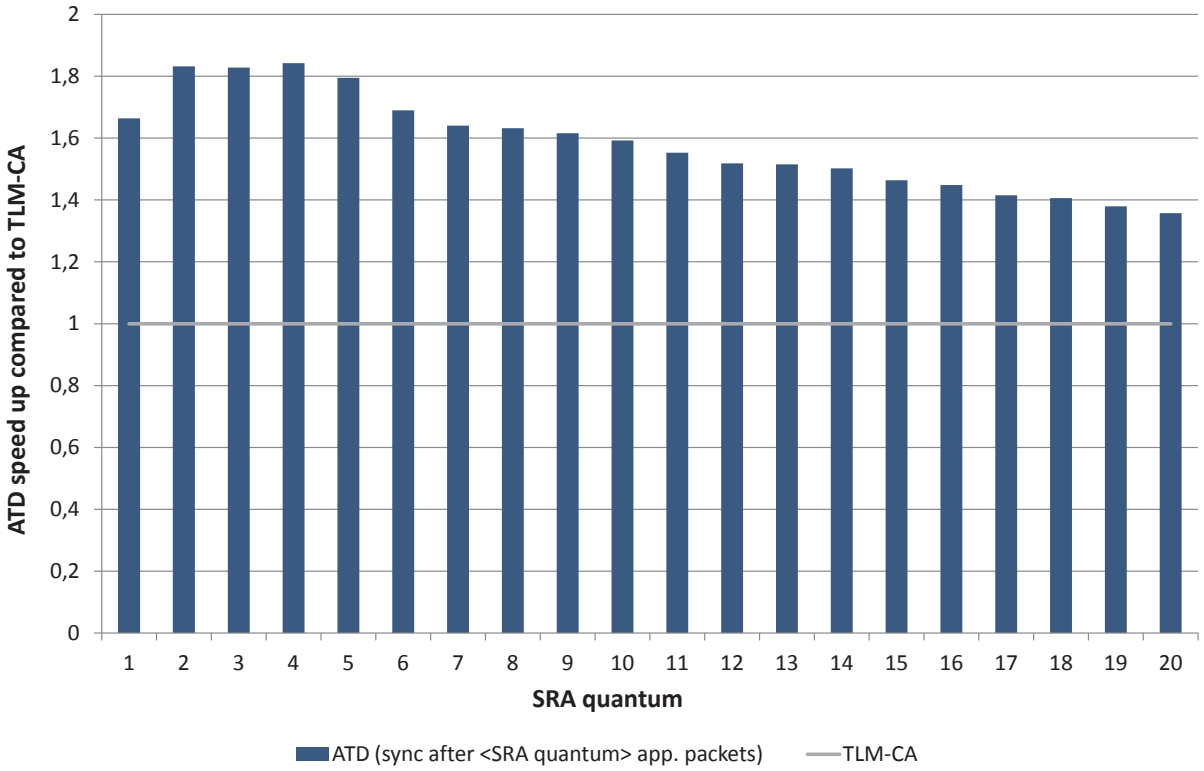



**Figure 3.21.:** ATD benchmark: three words per application packet and a varying $SRA_{quantum}$.

Thereby, the number of words per application packet is varied starting from single word sized application packets ranging to ten words per application packet. The ATD implementation is shown twice using different $SRA_{quantum}$ values. Setting the $SRA_{quantum}$ value to one effectively disables temporal decoupling and forces the synchronization after each application packet. As shown before, setting the $SRA_{quantum}$ value to three allows for the highest simulation performance.

The simulation time needed to process a fixed number of application packets using the conventional TLM-CA implementation grows approximately linearly with the number of words per application packet. This leads to a monotonous decrease of the number of application packets processed per second (wall clock time). In contrast the simulation performance of the ATD implementation is hardly influenced by the number of words per application packet but mainly depends on the amount of application packet preemptions occurring. A suitable indicator for the amount of occurring preemptions is the quotient of the average size of the actually processed application packets compared to the average size of the application packets registered by the simulation threads. If no preemption occurs as it is the case in this example for application packets consisting of less than three words, this quotient equals to one. If preemption is provoked by increasing the number of words per application packet, the quotient decreases and the simulation performance of the ATD based implementation is reduced.

Even for single word-sized application packets where there might not be any preemption at all, the ATD implementation achieves a better performance than the TLM-CA implementation. This is because ATD substitutes the global simulation time by local simulation times associated to each

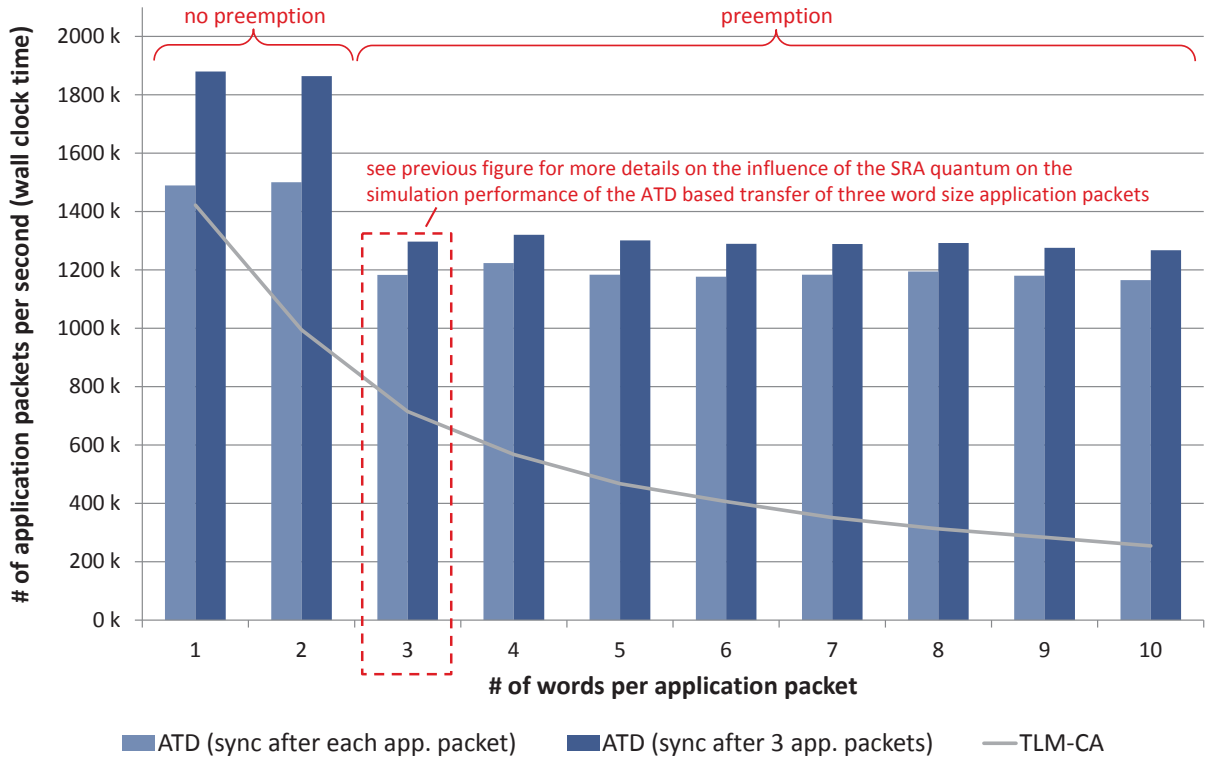



**Figure 3.22.:** ATD Benchmark: varying number of words per application packet. TLM-CA simulation performance continuously decreases with the number of words per application packet. ATD simulation performance generally depends on the amount of preemptions.

thread superseding the context switches entailed by calling `sc_core::sc_wait(<time>)`.

Besides the simple example system covered in this section, the performance improvement achievable by ATD is further investigated using an industrial application presented in Chapter 5.

# 4. Transparent Transaction Level Modeling

Besides extensive timing accuracy and high simulation speed provided by the ATD concept, a design flow allowing for the efficient creation of virtual prototypes is an important requirement in industrial environments. The Transparent Transaction Level Modeling (TTLM) methodology proposed by the author in [HBPG12, RRO+12] aims at the efficient creation of SystemC, TLM, SCML2 [Syn11], and ATD based virtual prototypes.

## 4.1. TTLM Design Flow

The efficient creation entails different aspects like the reduction of the implementation effort and the reduction of the amount of expert knowledge required for the creation of the virtual prototype. Additionally, the efficient creation of virtual prototypes depends on the available level of automation regarding validation and implementation tasks and the capabilities to collaborate with existing digital hardware design flows. The TTLM design flow shown in Figure 4.1 addresses these aspects.

The first step of the design flow is the formal specification of the hardware structure. This step is done by the designer. The structure of the components of the virtual prototype is defined using IP-XACT component descriptions and includes the interfaces of the components and the hierarchically structured memory maps of memory mapped target components. The structure of the virtual prototype is defined by instantiating and connecting the previously defined components using an IP-XACT design description. In contrast to a natural language based informal specification, IP-XACT represents a formal specification of the hardware structure. This allows for an automated validation of the specification and for an automated transformation into various design artifacts required during the development of digital hardware and hardware dependent software.

The TTLM Generator (see Section 4.3) is used to create the source code for the hardware structure of virtual prototypes based on the IP-XACT specification. The generated hardware structure includes module stubs for each component of the virtual prototype and a top level design file representing the netlist. The generated code comprises elements of different SystemC based modeling libraries. In case ATD is enabled using the TTLM generator configuration, temporal decoupled semaphores are automatically integrated into the corresponding shared resources. After the hardware structure has been generated, the designer implements the functionality of the hardware components. This can be done making use of the capabilities provided by the TTLM Library. The TTLM Library provides a set of convenience functions to simplify the implementation of different functionalities like the inter-component communication, the incorporation of the ATD simulation mechanism, the virtual prototype configuration, and the extraction of runtime information.

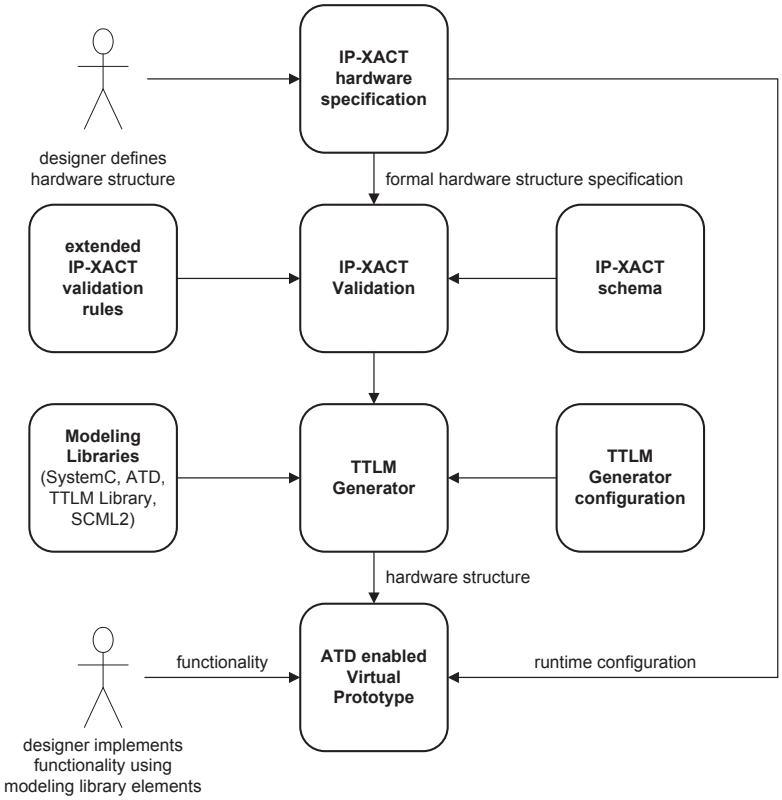



**Figure 4.1.:** TTLM design flow overview. The structure of the virtual prototype is derived from IP-XACT and assembled of SystemC, ATD, TTLM, and SCML2 [Syn11] library elements using the TTLM Generator. The functionality of the virtual prototype components is implemented by the designer making use of the convenience classes provided by the TTLM library.

## 4.2. Transparent Transaction Level Modeling Library

The objective of the TTLM library is to provide simple Application Programming Interfaces (APIs) which can be used to easily implement different functionalities required to build up virtual prototypes. This reduces the amount of source code which is to be implemented by the designer. Additionally, the expert knowledge required for the creation of SystemC based virtual prototypes is reduced as commonly used and complex code fragments are encapsulated within the library. This allows the designer to focus on the implementation of the functionality of the components.

To achieve this objective, the TTLM library combines a set of established libraries like SystemC TLM-2.0, the SystemC Modeling Library 2 [Syn11] (SCML2) [Syn11], the TLM Module Adapter library [Ker09] and the unitized approach [KBR09] with the ATD approach presented in Chapter 3. Figure 4.2 shows the basic structure of TTLM enabled virtual prototype components.

The functional description of the component's behavior is surrounded by a set of four APIs each having a distinct purpose. The inter-component communication API is used to accomplish the communication between different components of the virtual prototype in a SystemC TLM-2.0 compliant way and provides different means to ease the implementation of the initiator-side as well as the target-side communication end points. The configuration API enables changing component

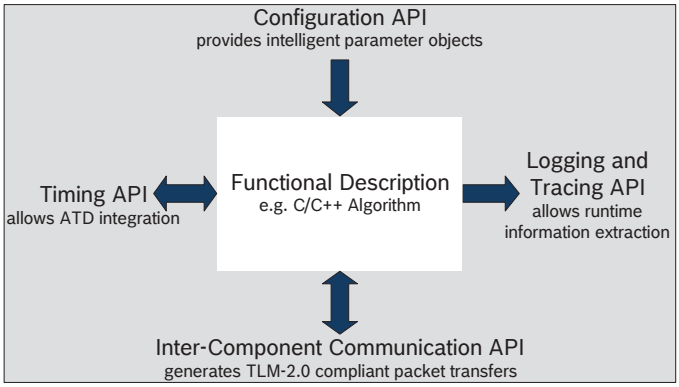



**Figure 4.2.:** Basic TTLM component structure. The TTLM library provides four distinct APIs for different purposes allowing the designer to focus on the functionality of the component.

parameters at runtime while the logging API provides means to extract runtime information. The timing API represents the user interface towards the advanced temporal decoupling class library.

### 4.2.1. Inter-Component Communication API

The implementation of the inter-component communication in conventional loosely-timed SystemC TLM-2.0 models is time-consuming and comprises various tasks. Table 4.1 summarizes the estimated effort required for the implementation of an inter-component communication in a conventional loosely-timed SystemC TLM-2.0 model containing a single initiator [Ayn08a, Ayn08b].

The implementation comprises tasks designated to the virtual prototype structure like socket instantiation and binding as well as tasks designated to communication occurrences like payload object creation, assembly, transfer, processing, and memory management. The implementation effort depends on several factors like the number of interconnection hops, the number of target components connected to each interconnect and the number of registers contained within each target module. Even in case of a simple virtual prototype containing one initiator only, approximately 76 Source Lines of Code (SLoC) are required to implement a conventional loosely-timed SystemC TLM-2.0 based inter-component communication.

The inter-component communication API of the TTLM library provides means to reduce the implementation effort required to implement the data transfer between different components of a SystemC TLM-2.0 based virtual prototype. The communication API consists of dedicated communication APIs for initiator-side and target-side communication.

#### 4.2.1.1. Initiator-side Communication

Figure 4.3 gives an overview of the initiator-side communication API. There are two APIs build upon each other and denoted as explicit and implicit initiator-side communication APIs, allowing to implement the functionality of the component in two different ways.

| task | approx. SLoC | remarks |
|---|---|---|
| socket instantiation | $2 \cdot \lvert\text{hops}\rvert$ | 1 initiator + 1 target socket per hop |
| socket binding | $3 \cdot \lvert\text{hops}\rvert$ | to modules and to complement socket |
| payload object creation | 2 | memory manager recommended |
| memory management | $2 + 2 \cdot \lvert\text{hops}\rvert$ | acquire and release payload |
| payload object assembly | 9 | set attributes like address, data, ... |
| payload object transfer | $\sum_{\text{hops}} (5 + 3 \cdot \lvert\text{targets}\rvert)$ | includes basic payload routing |
| payload object processing on target side: | | |
| validation | 27 | check attributes like address, data, ... |
| address decoding | $5 + 3 \cdot \lvert\text{registers}\rvert$ | |
| command decoding | 11 | read, write, ignore |
| response handling | 2 to 26 | 7 SystemC TLM-2.0 response codes |
| **lower bound** | 76 | $\lvert\text{hops}\rvert = \lvert\text{targets}\rvert = \lvert\text{registers}\rvert = 1$ |

$|\text{hops}|$ = number of hops the transaction traverses
$|\text{targets}|$ = number of target components connected to the interconnect
$|\text{registers}|$ = number of registers of the target

**Table 4.1.:** Estimation of the inter-component communication implementation effort for a conventional loosely-timed SystemC TLM-2.0 model containing one initiator. The effort is estimated based on [Ayn08a, Ayn08b] and measured in Source Lines of Code (SLoC).

### Explicit initiator-side Communication API

The explicit initiator-side communication API is provided by the `ttlm::MasterSocket` class shown in Figure 4.4. The `ttlm::MasterSocket` is based on the `master_module_socket` contained in the TLM Module Adapter library [Ker09]. Similar to the `master_module_socket`, the explicit TTLM initiator-side communication API provides `read(..)` and `write(..)` functions. These functions can be used to transfer arbitrary data using a single function call by specifying the destination or source address, the data buffer and the delay parameter. For features like the byte enable and streaming control (`p_byteEnable`, `byteEnableLength`, and `streamingWidth`) default values are provided to allow for a simple usage in case these SystemC TLM-2.0 features are not required.

Table 4.2 compares the features of the `ttlm::MasterSocket`, the `master_module_socket`, and the `initiator_socket` class provided by the SCML2 library.

Each of these initiator sockets allows to transfer the data by specifying at least the destination address, the data buffer, and the delay parameter required for a conventional temporal decoupled simulation. The `ttlm::MasterSocket` as well as the SCML2 `initiator_socket` provide a C++ template parameter for the type attribute of the data parameter and therefore do neither require manual transaction size calculation nor the explicit type conversion of the data which is to be transmitted. In this case, the `numElements` parameter of the `ttlm::MasterSocket` allows to initiate transfers of arrays comprising multiple objects of the same data type. The automatic payload object memory management improves the simulation performance by managing a pool

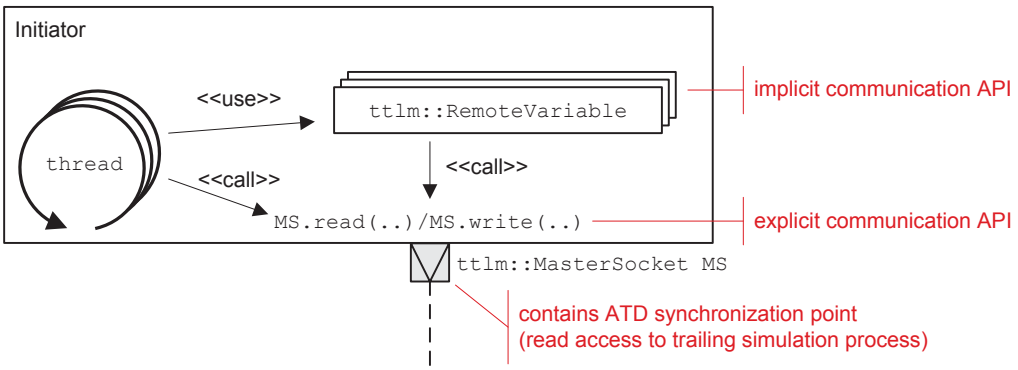



**Figure 4.3.:** Initiator-side communication overview. The implicit communication API consists of `ttlm::RemoteVariables` and uses the explicit communication API. Both APIs can be used to accomplish SystemC TLM-2.0 compliant communication.

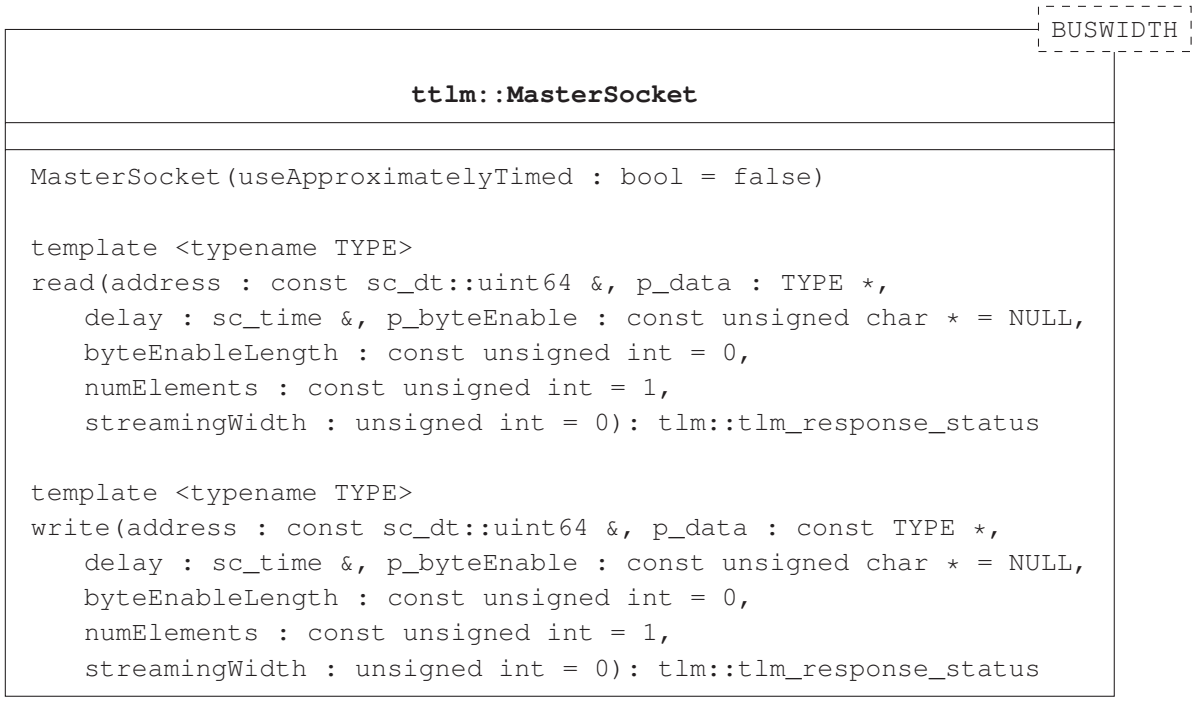



**Figure 4.4.:** Simplified MasterSocket class diagram.

of transaction objects as suggested by the TLM standard [IEE11]. If the optional payload ID extension is enabled, an unique ID is automatically added to each payload object to enable the traceability of the transactions.

Besides using the conventional loosely-timed modeling style, the TTLM `MasterSocket` can be configured to automatically transfer the data using the approximately-timed style by setting the `useApproximatelyTimed` constructor parameter. If the approximately-timed style is enabled, the communication takes place using the SystemC TLM-2.0 base protocol [IEE11] presented in Section 2.4.2. Thereby, the TTLM `MasterSocket` implementation automatically handles the base protocol phase transitions. The SCML2 `initiator_socket` allows to transfer the data using the approximately-timed style too, but requires the designer to implement the required

| feature | TTLM `MasterSocket` | TLM Module Adapter `master_module_socket` | SCML2 `initiator_socket` |
|---|---|---|---|
| read / write interface | x | x | x |
| data type template | x | | x |
| payload object memory management | x | x | |
| payload ID extension | x | x | |
| approximately-timed TLM support | x | | manual implementation required |
| ATD support | x | | |

**Table 4.2.:** Initiator socket feature overview comparing TTLM `MasterSocket`, TLM Module Adapter `master_module_socket` [Ker09] and SCML2 `initiator_socket` [Syn11].

phase transitions and to manually call the corresponding SystemC TLM-2.0 interfaces provided by the `initiator_socket`.

To allow for an ATD based simulation, the TTLM `MasterSocket` includes the ATD synchronization point required for reading access from a trailing shared resource. In case ATD is enabled and the `read(..)` function is called, the `MasterSocket` registers the reading access at the corresponding TDSem and the current thread is suspended waiting for the reading access to be finished.

### Implicit initiator-side Communication API

When using the `read(..)` and `write(..)` function calls provided by the explicit initiator-side communication API, the source code of the component's functional description explicitly reflects the communication occurrences. In case this is undesired or in case the source code describing the behavior of the component shall be reused requiring minimal code adaption only, the implicit initiator-side communication API can be used. The implicit initiator-side communication API is based on a set of remote variables. The remote variables are based on the unitized approach presented in [KBR09] as shown in Figure 4.5.

The `unitized` class maps a comprehensive set of object accesses and operations to appropriate `read(..)` and `write(..)` function calls using C++ operator overloading. Thereby, various constructors like the default and copy constructors and operators like assignment operators with or without arithmetic operations as well as increment and decrement operators are handled. The remote variables forward any object access which has been caught by the unitized class to the explicit initiator-side communication API. This results in the creation of SystemC TLM-2.0 compliant transactions.

In Listing 4.1 a simple example of the remote variable usage is shown. In Line 3 and Line 7 the `ttlm::MasterSocket` and the `ttlm::RemoteVariable` are instantiated. In Line 12 a value is written to the remote variable, which implicitly leads to a call to the `write(..)` function of

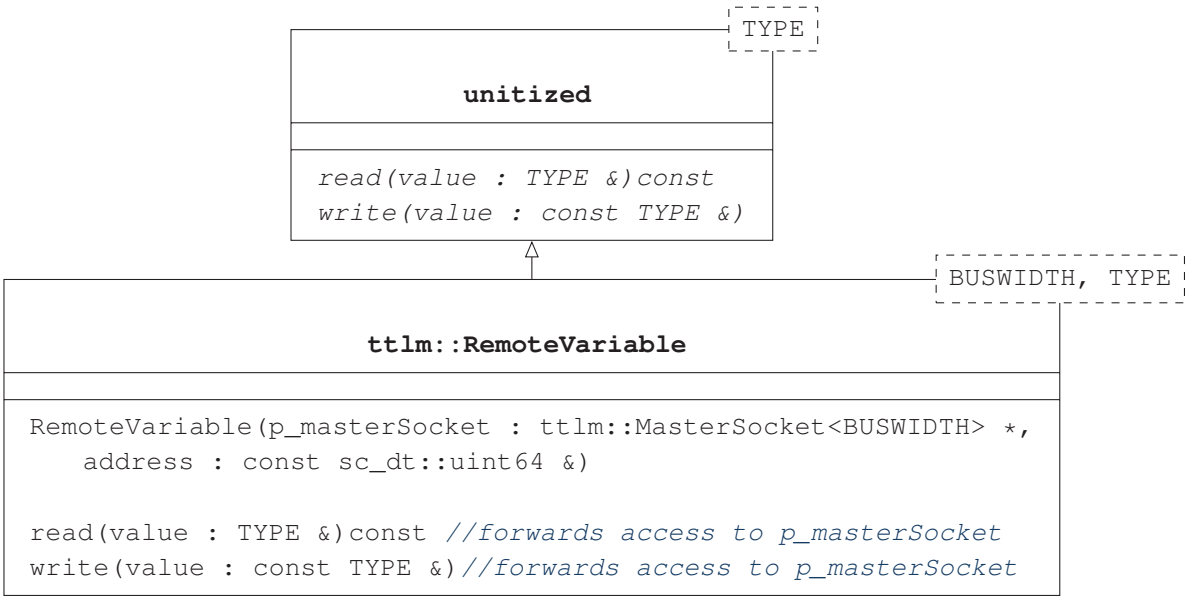



**Figure 4.5.:** Simplified RemoteVariable class diagram. The `ttlm::RemoteVariable` class is derived from the `unitized` class presented in [KBR09].

```
1  // the masterSocket provides the explicit initiator-side
2  // communication API used by the remote variable(s)
3  ttlm::MasterSocket<32> masterSocket;
4  // implicit initiator-side communication API remote variable:
5  // accessing r_var1 leads to read or write accesses to address 0x100
6  // utilizing the masterSocket
7  ttlm::RemoteVariable<32, int> r_var1(&masterSocket, 0x100);
8
9  // exemplary functional code
10 ...
11 int var1;
12 r_var1 = 42; // implicitly calls write(..) function of masterSocket
13 var1 = r_var1; // implicitly calls read(..) function of masterSocket
14 ...
```

**Listing 4.1:** Using the implicit initiator-side communication API.

the master socket, which in turn leads to a SystemC TLM-2.0 compliant transfer. The data is written to the address specified during the instantiation of the remote variable. Similarly, the read access shown in Line 13 leads to a reading access from the corresponding address.

The example shown in Listing 4.1 demonstrates that in case the `ttlm::RemoteVariables` of the implicit initiator-side communication API are used, the functional code does not reflect the communication occurrences explicitly. Instead, the communication occurrences are concealed by operations involving remote variable objects. This simplifies the reuse of the functional code, as only the declaration of memory mapped structs and variables are to be replaced by remote

variables.

To reduce the implementation effort for using the implicit initiator-side communication API in virtual prototypes comprising a considerable amount of memory mapped peripheral components, the TTLM generator can be configured to automatically create remote variable maps (see Section 4.3.2). These remote variable maps consist of multiple remote variables and represent the memory map structure of each peripheral component. They can be incorporated on the initiator-side to allow for a remote variable based access to the corresponding registers and bitfields.

#### 4.2.1.2. Target-side Communication

Figure 4.6 gives an overview of the target-side communication API. Similar to the initiator-side

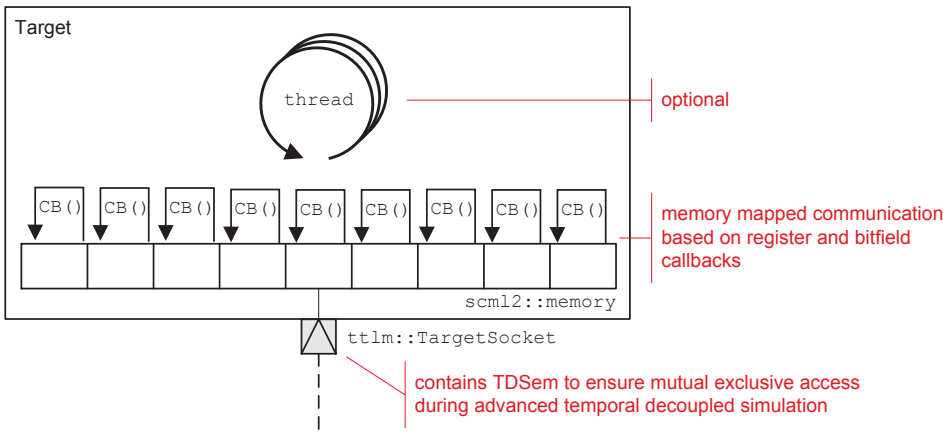



**Figure 4.6.:** Target-side communication overview.

communication API, the target-side communication API consists of two layers. The lower layer consists of an ATD enabled `ttlm::TargetSocket`, whereas the higher layer is represented by a memory map structure consisting of registers and bitfields. The `ttlm::TargetSocket` class is based on the `tlm_utils::simple_target_socket` class contained in the SystemC TLM-2.0 utility library and provides a plain SystemC TLM-2.0 transport interface. In contrast to the `tlm_utils::simple_target_socket`, the `ttlm::TargetSocket` integrates a temporal decoupled semaphore. If ATD is enabled, the integrated temporal decoupled semaphore ensures mutual exclusive shared resource accesses during a temporal decoupled simulation. Therefore, the `b_transport(..)` function is re-implemented to pass the incoming application packets to the temporal decoupled semaphore instead of immediately passing it to the user implemented `b_transport(..)` function. Additionally, the `executeSRA(..)` function of the `ttlm::TargetSocket` is implemented to incrementally pass possibly interleaved application packet fragments to the user implemented transport function of the target component. In this way, the `ttlm::TargetSocket` accomplishes dynamic application packet fragmentation adhering to the time budget calculated by the ATD mechanism. The fragment size is determined on a constant time per word basis:

$$fragmentSize = \left\lceil \frac{t_{budget}}{TDSem.\texttt{atomicTimePortion}} \right\rceil$$

To ease the modeling of memory mapped target components, the target-side communication API makes use of the memory map related elements provided by the SCML2 [Syn11] library. These elements comprise the memory, memory_alias, register, and bitfield classes and are used to create the register and bitfield structure of the memory mapped target component. Thereby, SCML2 supports an automated address and bitfield decoding including the management of accesses spanning multiple registers and / or bitfields. Optionally, a user defined functionality can be added to each register or bitfield using read and write callback functions, which are invoked in case the corresponding register or bitfield is accessed during a TLM transfer.

To reduce the coding effort required for the implementation of memory mapped target components, the TTLM Generator (see Section 4.3) allows for an automatic generation of the memory map related SCML2 code based on an IP-XACT component description.

### 4.2.2. Timing API enabling ATD integration

The usage of the timing API is optional and is only required in case TTLM is used in conjunction with ATD. The timing API allows for the integration of timing related ATD concepts. As shown in Chapter 3, the simulation time handling in ATD differs from the simulation time handling used in a conventional SystemC simulation. In ATD, the global simulation time is replaced by a set of local simulation times each associated to one simulation thread. Additionally, the consumption of local simulation time does not comprise a synchronization point superseding the context switch occurring during the time consumption in a conventional SystemC simulation. Furthermore, the processing of SRAs is deferred which renders the local simulation time of the issuing thread invalid. Therefore, a synchronization is required when reaching statements which depend on the validity of the local simulation time like it is the case for the last three synchronization points shown in Table 3.1 on page 37. To adapt the simulation time handling according to the ATD thread execution semantics, the TTLM Timing API provides three utility classes shown in Figure 4.7.

The `ttlm::sc_ttlm_module` and `ttlm::sc_ttlm_event` classes replace the corresponding classes of the SystemC library. The `sc_ttlm_module` class overrides the `wait(<time>)` function to increase the local simulation time of the calling thread without suspending the thread. Furthermore, an implementation of the `sc_time_stamp()` function is provided which returns the current local simulation time of the calling thread instead of the global simulation time. Additionally, the `sc_time_stamp()` function comprises an ATD synchronization point which suspends the calling thread in case there are pending SRAs registered by that thread. This allows the pending SRAs to be processed and thereby the local simulation time of the thread to become valid. Similarly, the `wait(<event>)` function implemented in the `sc_ttlm_module` and the `notify(<time>)` function implemented in the `sc_ttlm_event` class suspend the calling thread in case there are pending SRAs registered by that thread.

The `localVolatileVariable` utility class can be used during ATD simulation to exchange data between threads which might reside within the same module. Therefore, the utility class integrates a temporal decoupled semaphore to ensure read and write accesses to occur in the correct temporal order and at the intended global simulation time. Similar to the remote variable

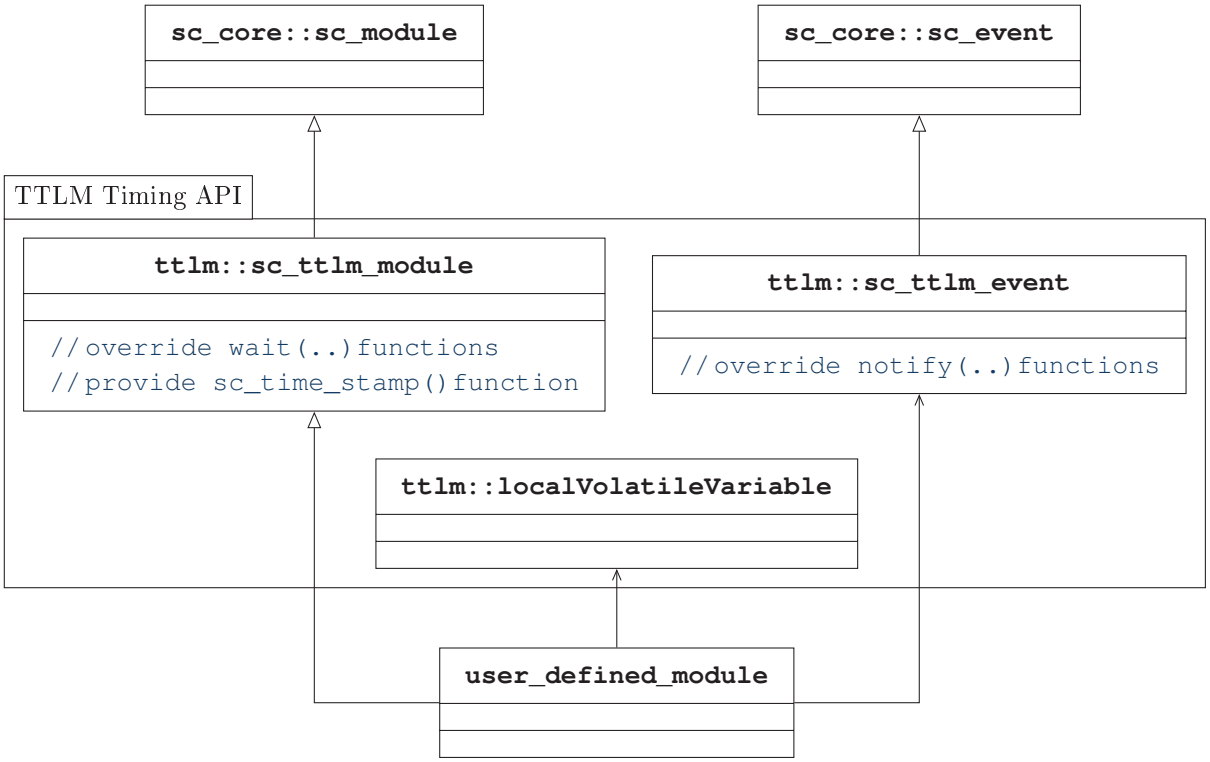



**Figure 4.7.:** Simplified TTLM Timing API class diagram. The utility classes `ttlm::sc_ttlm_module` and `ttlm::sc_ttlm_event` override the `wait(..)` and `notify(..)` functions of their respective SystemC parent classes to comply with the ATD semantics. The `ttlm::localVolatileVariable` class can be used to implement ATD compliant data exchange between local threads.

class provided by the implicit initiator-side communication API, the `localVolatileVariable` class is based on the unitized approach [KBR09] and hence can be used like a conventional variable.

To make use of the ATD concept in the context of TTLM, the timing API has to be incorporated by deriving the `user_defined_module` from the `ttlm::sc_ttlm_module` class instead of the `sc_core::sc_module` class. When using the TTLM generator, this can be achieved by enabling ATD in the TTLM generator configuration (see Section 4.3.2). Additionally, all `sc_core::sc_event` occurrences have to be replaced by `ttlm::sc_ttlm_event`.

### 4.2.3. Virtual Prototype Configuration API

The virtual prototype configuration API can be used to change parameter values without requiring the re-compilation of the virtual prototype. Figure 4.8 shows an example of the configuration API usage. The parameter values are stored in an external data source and loaded on simulation start-up using a global configuration object. The TTLM library provides a connector to use IP-XACT design description files as an external data source. The IP-XACT design description files are used to define the netlist of the virtual prototype and thereby allow for the specification of `<spirit:configurableElementValue>` objects for each component instance contained within the design.

There are two ways of accessing the parameter values from the virtual prototype implementation. The first and explicit way is calling the `getParameter(..)` interface implemented by the global configuration object (see Appendix A.2 for a detailed description). Alternatively, the `ttlm::Parameter<TYPE>` utility class, which is based on the unitized approach [KBR09], provides implicit parameter value access. Similar to the local volatile variables and to the remote variables, the parameter objects can be accessed like conventional variables and automatically obtain the parameter value from the external data source.

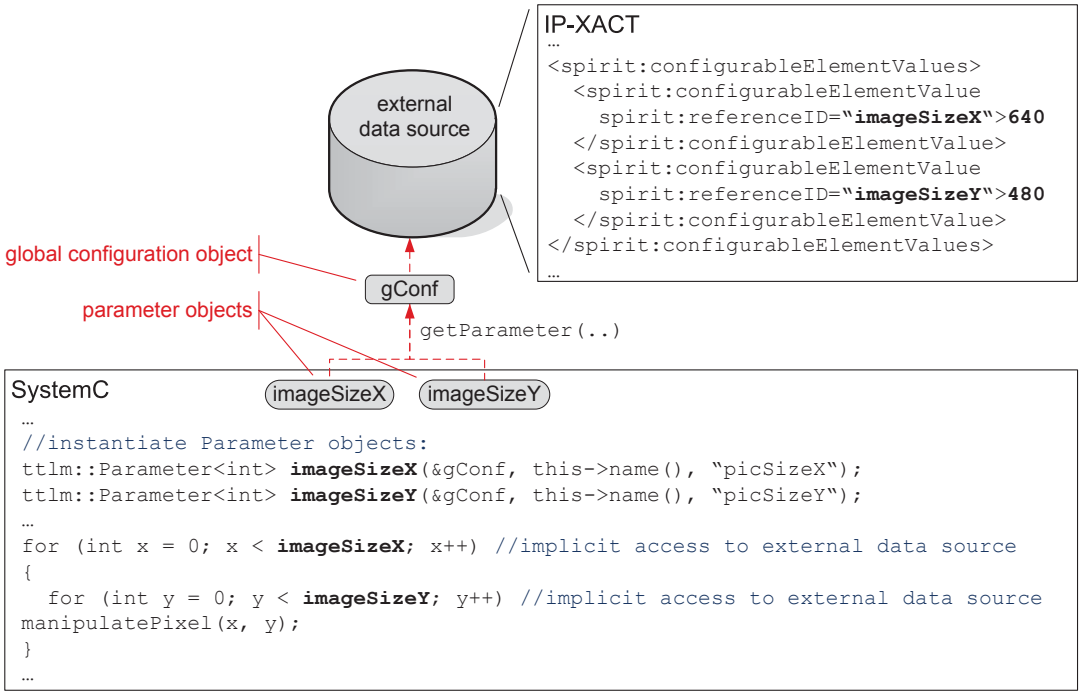



**Figure 4.8.:** Virtual prototype configuration example. The parameter values are stored in an external data source like an IP-XACT design description file and can be accessed either using the `getParameter(..)` interface of the global configuration object or by using `ttlm::Parameter` objects.

### 4.2.4. Runtime Information Extraction

The TTLM library provides different utilities to ease the aggregation of runtime information during virtual prototype execution. Textual information can be extracted using simple C++ preprocessor macros. Especially for the extraction of ATD related runtime information, the runtime information extraction API provides the `ttlm::Profiler` utility class. In case ATD is used and the ATD profiling is enabled by defining the `ENABLE_PROFILING` preprocessor symbol, the profiler gathers runtime information for offline analysis. Thereby, the following information can be extracted.

- the overall number of SRAs issued by each thread
- the overall utilization time and utilization ratio of each resource
- the utilization time and utilization ratio of each resource itemized by the accessing thread

- the number of deadlocks detected by the ATD deadlock detection algorithm described in Section 3.5.1
- for SRAs where `T_userData_t` matches `tlm::tlm_generic_payload` the following statistics are provided itemized by the accessed resource and the simulation thread which directly or indirectly registered the SRA
  - number of registered transactions itemized by the number of words per transaction
  - average number of words per registered transaction
  - number of processed transactions itemized by the number of words per transaction
  - average number of words per processed transaction

## 4.3. Transparent Transaction Level Modeling Generator

The TTLM generator reduces the implementation effort and expert knowledge required for the creation of a virtual prototype as major parts of the SystemC, SCML2, ATD, and TTLM related code is generated automatically. The TTLM generator operation is based on the formal specification of the structure of the virtual prototype in terms of a set of IP-XACT descriptions. The information contained in the formal specification is transformed into the SystemC based structure of the virtual prototype composed of ATD, TTLM, and SCML2 objects.

The TTLM generator is based on the Eclipse Modeling Framework (EMF) [EF13a] and the Xpand2, Xtend, and Check technology initially published by OpenArchitectureWare [Ope13]. Figure 4.9 gives an overview of the architecture of the TTLM generator. The IP-XACT component and design descriptions of the structure of the virtual prototype are transformed into an EMF model. This EMF model adheres to an EMF meta-model representing the IP-XACT schema [Acc13]. Before the transformation takes place, the content of the IP-XACT descriptions represented by the EMF model is validated against an extended set of IP-XACT validation rules as shown in Section 4.3.1. In case the IP-XACT descriptions comply to the extend IP-XACT validation rules, the transformation is started. The details of the transformation are presented in Section 4.3.2.

The benefits of using a code generator operating on a formal specification like IP-XACT are manifold. The employment of IP-XACT allows for an easy integration of the TTLM design flow into existing digital hardware design flows by reusing the IP-XACT descriptions. This results in IP-XACT being the central data source for digital hardware design, virtual prototyping, and driver software development.

### 4.3.1. Automated IP-XACT Validation

The automated IP-XACT validation improves the efficiency of the creation of a virtual prototype by revealing many specification errors at an early design stage. This reduces the debugging and error treatment effort at subsequent design stages. The validation applied by the TTLM generator exceeds usual compliance checks against the IP-XACT schema and comprises the validation of

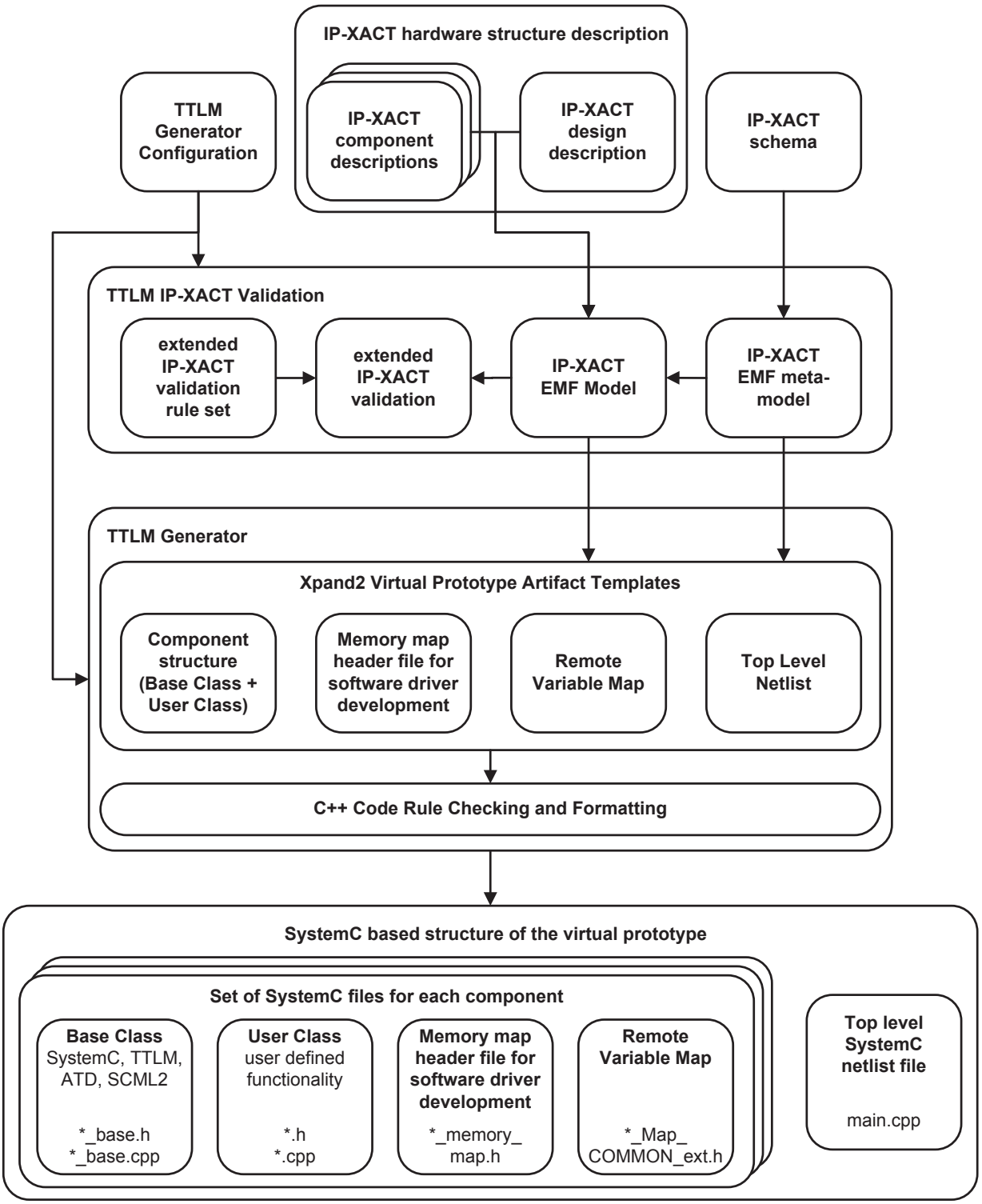



**Figure 4.9.:** TTLM generator overview. The IP-XACT component and design descriptions are validated using an extended IP-XACT validation rule set. Then, they are transformed into the structural code of an SystemC, TTLM, ATD, and SCML2 based virtual prototype. For each IP-XACT component description a set of SystemC artifacts is generated depending on the TTLM generator configuration.

more sophisticated requirements like the unambiguity and completeness of memory map elements. These additional checks are defined in a set of extended IP-XACT validation rules.

The extended IP-XACT validation rules are implemented using the Check [Ope13] syntax. Listing 4.2 exemplifies the validation of the unambiguity of register names within each address block. The first line of the check statement specifies the context or object type to which this check is to be applied and the corresponding severity level if the check fails. In this example, the check is applied to address block objects. The second line determines the error message which is to be displayed in case the check fails. The third line defines the condition which must be fulfilled to successfully pass the check. The unambiguity of object names is checked for a variety of IP-XACT

```
1 context index::AddressBlockType ERROR
2 "register names are not unique: " + this.register.name :
3 this.register.name.toSet().size == this.register.name.size
4 ;
```

**Listing 4.2:** Extended IP-XACT validation rule example: unambiguity check for address block names.

objects including memory map objects like address blocks, registers, bit fields, bus interfaces, as well as component instances and interconnection names used in the top level design specification.

Besides the checks assuring the unambiguity of object names, there are various other checks. One other class of checks verifies the existence of referenced IP-XACT objects. Thereby, objects contained within other IP-XACT files which are referenced using Vendor, Library, Name, and Version [IEE10] (VLNV) identifiers as well as references to objects residing within the same IP-XACT file like memory maps and address spaces referenced from bus interfaces are checked. Especially for memory maps extensive checks are applied. These memory map checks validate the absence of overlapping address blocks, registers, and bitfields as well as the sufficiency of the range attribute of address blocks to include all subordinate register addresses. Additionally, the validity of the callback types which are to be generated as part of the target side communication API for each memory map element is checked. Besides the mentioned checks, interconnection attributes like matching buswidths, the unambiguity of routing address ranges, and the interconnection of matching interface types like master interfaces to slave interfaces are validated.

### 4.3.2. SystemC Artifact Generation

After the validation of the IP-XACT description has been successfully completed, the generation of the virtual prototype artifacts is started. The TTLM code generation is integrated into an Eclipse [EF13b] based Bosch SystemC development environment and is based on the Modeling Workflow Engine (MWE) which is used to process the Xpand2 [Ope13] virtual prototype artifact templates. The configuration of the TTLM generation process is done using the graphical configuration user interface shown in Figure 4.10.

On the left side of the TTLM generator configuration window, different features can be enabled which are to be included into the generated code like logging statements executed upon memory

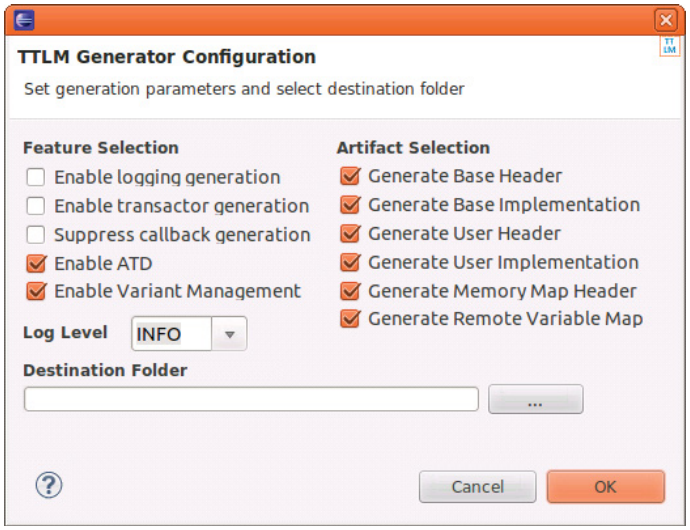



**Figure 4.10.:** The Graphical TTLM Generator Configuration interface allows for the selection of features and artifacts the generated virtual prototype structure consists of.

map object accesses, ATD support, and variant management support. On the right side of the window, the SystemC artifacts which are to be generated for each IP-XACT component description can be selected. Thereby, six component related SystemC file types are available.

- The **Base Header** file contains the class declaration of the base class of a component. The base class contains the SystemC, ATD, and TTLM related code required to implement the structure of the component including its communication interfaces and the SCML2 based memory maps. The Base Header file does not need to be edited by the designer.
- The **Base Implementation** file contains the definition of the base class functions of the component. This includes the code required to setup the communication interfaces and the SCML2 based memory map. Similar to the Base Header file, this file does not need to be edited by the designer.
- The **User Header** file contains the class declaration of the user class of the component. This class is derived from the base class and declares the functions which are to be implemented by the designer. In case the component entails memory mapped register structures accessible via a communication channel, this file contains the declarations of the callbacks of the memory mapped target-side communication API (see Section 4.2.1.2).
- The **User Implementation** file contains the definition of the user class functions of the component and is to be edited by the designer to implement the functionality of the component making use of the TTLM APIs presented earlier.
- The **Memory Map Header** file is supposed to be used for driver software development and contains preprocessor macros representing relevant memory map element properties like register addresses and reset values as well as bitfield attributes.
- The **Remote Variable Map** file contains a class consisting of multiple remote variables representing the register structure of memory mapped target components. This class can be used as an implicit initiator-side communication API and allows for a register and bitfield name based access to target components.

### Component Structural Code

The separation of the structural component code into base and user classes reduces the complexity of the code exposed to the designer as the SystemC, SCML2, ATD, and TTLM related structural code is concealed within the base class. This allows the designer to focus on the implementation of the functionality of the component. In case a new generation of the structural code is required due to changes in the IP-XACT description, the functionality which has already been implemented is preserved by using protected regions.

By default, the base class is derived from the `sc_core::sc_module` class. In case ATD is enabled during TTLM generation, the base class inherits from `ttlm::sc_ttlm_module`. This enables the TTLM Timing API described in Section 4.2.2 which replaces the `wait(<time>)`, `wait(<event>)`, and `sc_time_stamp()` functions by ATD compliant versions.

The communication interfaces of the component are instantiated in the base class. Thereby, `sc_in` and `sc_out` signal ports as well as transaction level interfaces are supported. The data type of the signal ports depends on the `<spirit:qualifier>` defined in the IP-XACT abstraction definition used for the corresponding interface. In case `<spirit:isClock>` or `<spirit:isReset>` is set to "true", the data type is set to `sc_time` or `bool`, respectively. If neither `<spirit:isClock>` nor `<spirit:isReset>` is set to "true", the data type is set to `sc_uint`. In this case, the width of the vector is specified by the `<spirit:width>` attribute of the IP-XACT abstraction definition.

Initiator-side transactional interfaces are implemented using the `ttlm::MasterSocket` class, which provides the explicit initiator-side communication API. Target side transactional interfaces are implemented using the `ttlm::TargetSocket` class. In case an IP-XACT component description contains a memory map which is referenced from the corresponding bus interface definition, a `scml2::tlm2_gp_target_adapter` is used to connect the socket to the hierarchical SCML2 memory map structure. The hierarchical SCML2 memory map structure is created by applying the IP-XACT to SCML2 mapping shown in Table 4.3.

To attain unambiguous names for the SCML2 objects, all names of hierarchically superior IP-XACT memory map objects are concatenated to fully qualified names (FQNs). The access right to the memory map objects is specified using the `<spirit:access>` attribute, which can be applied at any hierarchy level of the memory map. The access rights to child elements are inherited from the parent element but can be overridden by specifying a different access right type at a hierarchically subordinate level. If the access rights are restricted, the TTLM generator creates calls to the `set_disallow_read_access(..)` and `set_disallow_write_access(..)` functions for the corresponding memory map element accordingly.

To add user defined functionality to the created memory map elements, SCML2 provides the possibility to register callback functions at the memory map elements at each hierarchical level. Similar to the inheritance of the access type, the selected callback type is inherited towards the leaf level elements of the memory map including the possibility to override the inherited callback type at hierarchically subordinate levels. The creation of the callbacks by the TTLM generator can be configured by adding the IP-XACT code shown in Listing 4.3 to the `<spirit:parameters>` section of the corresponding memory map element.

| IP-XACT | SCML2 |
|---|---|
| <spirit:memoryMap> | scml2::memory |
| <spirit:name> | name of the scml2::memory object |
| <spirit:addressBlock> | scml2::memory_alias |
| <spirit:name> | name suffix of the scml2::memory_alias object |
| <spirit:baseAddress> | offset attribute of scml2::memory_alias constructor |
| <spirit:range> | size attribute of scml2::memory_alias constructor |
| <spirit:width> | transformed into appropriate datatype template argument of scml2::memory, scml2::memory_alias, scml2::reg, and scml2::bitfield |
| <spirit:access> | call to set_disallow_[read\|write]_access(..) |
| <spirit:register> | scml2::reg |
| <spirit:name> | name suffix of the scml2::reg object |
| <spirit:addressOffset> + <spirit:baseAddress> | offset attribute of scml2::reg constructor |
| <spirit:access> | call to set_disallow_[read\|write]_access(..) |
| <spirit:reset> | call to initialize(..) function of scml2::reg |
| <spirit:field> | scml2::bitfield |
| <spirit:name> | name suffix of the scml2::bitfield object |
| <spirit:bitOffset> | offset attribute of scml2::bitfield constructor |
| <spirit:bitWidth> | size attribute of scml2::bitfield constructor |
| <spirit:access> | call to set_disallow_[read\|write]_access(..) |

**Table 4.3.:** IP-XACT to SCML2 mapping used for the generation of TTLM target component memory maps.

```
1 <spirit:parameter>
2   <spirit:name>callback</spirit:name>
3   <spirit:value>callback_type</spirit:name>
4 </spirit:parameter>
```

**Listing 4.3:** Configuring the callback type using an IP-XACT parameter.

Thereby, the *callback_type* can be selected from the following list.

- read: create callbacks for all hierarchically subordinate memory map elements executed upon read accesses
- write: create callback for all hierarchically subordinate memory map elements executed upon write accesses
- read-write: create callbacks for all hierarchically subordinate memory map elements executed upon read or write accesses
- none: disable the callback generation in case a different callback type is specified at a hierarchically superior level.

The resulting callback functions are declared in the user header file. The user implementation file contains default implementations created by the TTLM generator implementing the appropriate read or write access. Listing 4.4 shows the structure of the generated callbacks. The functionality

```
1  bool component_name::memory_map_element_FQN_[read|write]CB (
2    addressBlock_width_dependent_data_type& value, sc_time& delay) {
3    // functionality which is to be executed before value is
4    // [read from | written to] the memory map element:
5    ... // <- to be implemented by the designer
6
7    // memory access:
8    // [read value from | write value to] the SCML2 memory map element:
9    [ value = *p_memory_map_element_FQN; |
10   *p_memory_map_element_FQN = value; ]
11
12   // functionality which is to be executed after memory access took
13   // place:
14   ... // <- to be implemented by the designer
15   return true;
16 }
```

**Listing 4.4:** Structure of the generated callbacks.

which is to be executed before the memory access takes place like range checks for write accesses or just-in-time calculations of values during reading accesses can be implemented in Line 5. In case the memory access shall be inhibited for example due to a failing check during a write access, the memory access can be aborted by returning false. The functionality which is to be executed after the memory access took place like for example the post processing of the written data can be implemented in Line 14. In combination with the simulation threads, the memory map element callbacks are the main place to implement user defined functionality. The generation of memory map element callbacks can be suppressed by enabling the corresponding checkbox of the TTLM generator configuration.

### Memory Map based Variant Management

To simplify the unified development of virtual prototypes representing device families, the TTLM generator supports a memory map based variant management. The variant specific structure of the memory map can be configured by adding the IP-XACT code shown in Listing 4.5 to the `<spirit:parameters>` section of the corresponding memory map elements. Thereby, the *list_of_variant_identifiers* is a comma separated list of arbitrary variant identifiers.

In case the variant management support is enabled using the TTLM generator configuration, a header file containing an enumeration of the available variant identifiers is generated for each component. Additionally, the generated SCML2 memory map code is capable of representing

```
1 <spirit:parameter>
2   <spirit:name>AvailabelOnVariants</spirit:name>
3   <spirit:value>list_of_variant_identifiers</spirit:name>
4 </spirit:parameter>
```

**Listing 4.5:** Configuring a variant specific memory map structure using IP-XACT parameters.

each available variant by instantiating and configuring the appropriate memory map elements which belong to the currently selected variant at simulation start-up. This allows for a flexible configuration of the virtual platform by selecting the appropriate variant of each component. The selection of the desired variant takes place during component instantiation in the top level SystemC file (see corresponding paragraph on page 85).

### Memory Map Header File and Remote Variable Map for Driver Software Development

To ease the driver software development for memory mapped target components, the TTLM generator is capable of transforming the IP-XACT memory map information into memory map header files and remote variable maps which can be used as the equivalent implicit initiator-side communication API of the memory map of the target component. The memory map header files contain C preprocessor definitions representing the address block and register addresses as well as the bit field positions, bit field masks, and reset values. In combination with an additional offset representing the base address of the target component within the global memory map, the address block and register addresses can be used in conjunction with the explicit initiator-side communication API. The bitfield related constants can be used for bit wise logic and shift operations required to access and manipulate single bitfield values.

The remote variable maps can be used to accompany C type declarations commonly used for driver software development which reflect the register structure of memory mapped peripherals. Listing B.3 on page 114 shows the structure of the C type declarations. For each register type a bitfield structure and a union are declared (see Lines 2 to 12). This allows the register to be accessed in its entirety or via the contained bitfields. The memory map layout type is declared as shown in Lines 17 to 21. Using these type declarations, the embedded software is able to access the registers using the code shown in Listing 4.6.

The remote variable map shown in Listing B.4 on page 114 consists of C++ classes which correspond to the C type declarations commonly used for driver software development shown in Listing B.3. For each register type a remote register class is generated containing unitized [KBR09] based member objects. Similar to the unions contained in the C type declaration, these member objects allow the register to be accessed in its entirety or via the contained bitfields (see lines 8 to 17). These objects map any access to the TTLM `MasterSocket` which in turn generates SystemC TLM-2.0 compliant communication packets (see Lines 20 to 50). The class shown in lines 66 to 78 assembles the remote register classes to form the remote variable map according to the memory map. When using the remote variable map instead of the C type declarations,

```
1 ...
2 // use pointer of declared C type, pointing to the base address of the
3 // memory mapped peripheral:
4 componentName_MemMap_st *p_component =
5   (componentName_MemMap_st *)base_address of memory mapped peripheral
6 ...
7 p_component->register1_u.as_u32 = 42 // register access
8 p_component->register1_u.as_s.bitfield1 = 42 // bitfield access
9 ...
```

**Listing 4.6:** Using a pointer of the declared C type to access the registers of the memory mapped peripheral.

only the declaration and initialization of the p_component pointer of the example shown in Listing 4.6 needs to be changed according to Listing 4.7. The remaining driver software code typically remains unchanged.

```
1 ...
2 ttlm::masterSocket<32> socket;
3 // use pointer to the generated remote variable map to access the
4 // memory mapped peripheral:
5 componentName_RemoteVariableMap_st *p_component =
6   new componentName_RemoteVariableMap_st(&socket, base_address
      ofmemory mapped component)
7 ...
8 // functional code remains unchanged:
9 p_component->register1_u.as_u32 = 42 // register access
10 p_component->register1_u.as_s.bitfield1 = 42 // bitfield access
11 ...
```

**Listing 4.7:** Using the generated remote variable map to access the registers of the memory mapped peripheral.

The remote variable map can be thought of as an automatically generated implicit to explicit initiator-side communication API mapping and can be employed for a host-compiled execution of low level software like hardware drivers as part of a virtual prototype. Compared to the software execution on an ISS, a host-compiled execution provides a significantly higher simulation speed [Ger10].

#### Top Level SystemC File

The top level SystemC file is used to instantiate and interconnect the previously generated components to form the virtual prototype. The required information is extracted from the IP-XACT

design description. The instantiated components are connected by binding the corresponding TTLM sockets and by using signals to connect the corresponding ports.

In case the variant management support is enabled during TTLM generation, the component instantiation contained in the top level SystemC file is extended to allow for the selection of the desired variant of each component instance. Thereby, the currently selected variant of each component instance is not specified within the SystemC code but within the IP-XACT design description file. This is done by adding the IP-XACT code shown in Listing 4.8 to the `<spirit:configurableElementValues>` list of the corresponding component instance.

```
1 <spirit:configurableElementValue spirit:referenceId="selectedVariant">
2   variant_name
3 </spirit:configurableElementValue>
```

**Listing 4.8:** Component instance variant selection in IP-XACT design description file.

Within the top level SystemC file shown in Listing 4.9, the variant information is retrieved using the TTLM Virtual Prototype Configuration API presented in Section 4.2.3. At first, the TTLM Virtual Prototype Configuration API is set up (see Lines 4 to 6). In Lines 10 and 12, the variant string retrieved from the IP-XACT design description via the TTLM parameter is converted to the *componentName*_var type and passed to the constructor of the component.

```
1  #include "componentName_variants.h"
2  ...
3  // setup TTLM Virtual Prototype Configuration API:
4  ttlm::IPXACTConfig gConf("<path_to_IP-XACT_design_file>.xml");
5  ttlm::Parameter<std::string> instanceName_var_parameter(&gConf,
6    "instanceName", "selectedVariant");
7  ...
8
9  // fetch selected variant from IP-XACT design description:
10 componentName_var instanceName_var(instanceName_var_parameter);
11 // component instantiation:
12 componentName *p_instanceName = new componentName("instanceName",
13   instanceName_var);
14 ...
```

**Listing 4.9:** Top level SystemC file including component variant management support.

This variant selection technique allows for the configuration of the currently selected component variants in the IP-XACT design description file and neither requires a new generation nor a recompilation of the virtual prototype.

## 4.4. Implementing a simple Virtual Prototype using TTLM

In the following, the implementation effort reduction and usability improvement achievable by the TTLM design flow is demonstrated. Therefore, the TTLM design flow is employed for the creation of a virtual prototype of a simple example system. Figure 4.11 shows the architecture of the system. The system consists of a Timer component connected to a Testbench component via a generic bus connection consisting of a TLM connection and a clock connection. The Timer component contains a single register returning the current timer value upon read access.

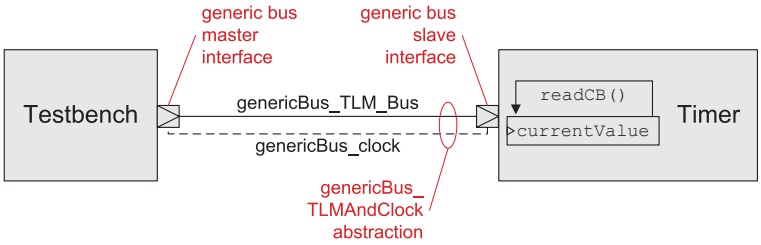



**Figure 4.11.:** Architecture of the simple example system.

According to the TTLM design flow shown in Figure 4.1, the system architecture is specified using a set of IP-XACT descriptions. Compared to the manual implementation of the structure using SystemC, the specification of the structural aspects using IP-XACT is much more efficient. This is because the formal semantic of IP-XACT is limited to the structural aspects of digital hardware reducing the implementation choices and thus the error-proneness for a given task. Additionally, this limitation allows for extensive IP-XACT editing tool support ranging from code completion to graphical memory map editors. As shown in Section 4.3.1 the tool support might also comprise IP-XACT validation, which automatically reveals many specification errors of the structural aspects at an early design stage. Listing B.5 on page 116 shows the IP-XACT component description of the timer component. According to the IP-XACT standard, the Timer component is identified by a Vendor, Library, Name, and Version [IEE10] (VLNV) identifier (see Lines 3 to 6). The generic bus slave interface of the Timer component is specified in Lines 7 to 16 by referencing the appropriate IP-XACT bus definition and abstraction definition. The referenced abstraction definition contains a TLM interface denoted as genericBus_TLM_Bus and a clock port denoted as genericBus_clock. The generic bus slave interface is used to provide access to the registers contained in the memory map of the Timer Component. The memory map is specified in Lines 17 to 38 and contains the specification of the `currentValue` register shown in Lines 25 to 35. The configuration of the TTLM memory mapped target-side communication API which is to be created by the TTLM generator is shown in Lines 29 to 34. According to Listing 4.3, the IP-XACT parameter tag is used to advice the TTLM generator to create a read callback which is called upon a read access to the address of the corresponding register.

After the specification of the system architecture has been completed, the TTLM generator is used to create the structural code of the Testbench and Timer components as well as the top level SystemC file containing the instantiation and interconnection of the components. The result of the TTLM generator employment is exemplified in Listings B.6 to B.9. The listings show the generated structural code of the Timer component separated into header files and implementation files for the

base class and the user class. Table 4.4 lists the Timer component related implementation tasks automatically carried out by the TTLM generator and provides references to the corresponding source code lines.

| **automated implementation task** | **timer_base.h** Listing B.6 | **timer_base.cpp** Listing B.7 | **timer.h** Listing B.8 | **timer.cpp** Listing B.9 |
|---|---|---|---|---|
| module infrastructure | Lines 21-28 | Lines 9-14 | Lines 7-14 | Lines 3-4 |
| communication interfaces | Lines 33-35 | Line 16 | | |
| clock handling | Lines 30-31 | Lines 3-7, 22-24 | | |
| memory map | | | | |
| instantiation | Lines 44-46 | Lines 29-30 | | |
| initialization | | Lines 19-20 | | |
| connection to communication interface | Lines 48-49 | Lines 26-27 | | |
| register callback function | Lines 37-41 | Line 32 | Lines 16-20 | Lines 6-23 |

**Table 4.4.:** Timer component related implementation tasks carried out by the TTLM generator.

In case ATD is enabled during the code generation, the base class of the Timer and the Testbench components are derived from the `ttlm::sc_ttlm_module` class enabling the TTLM timing API. This allows for the incorporation of the ATD simulation mechanism.

After the structural code has been generated, the functionality of the components is implemented by the designer. In case of the basic Timer component, this functionality comprises only a single line of code embedded into the read callback function of the `currentValue` register (see Listing B.9 Line 16). This statement is used to perform the just-in-time calculation of the current timer value when a read access to this register takes place.

The implementation of the functionality of the Testbench is shown in Listing B.10. The automatically generated base class and the user class header files of the Testbench have been skipped, as they are similar to the corresponding files of the Timer component and do not need to be edited by the designer. In Lines 20 and 21 the explicit initiator-side communication API is used to read the current timer value by issuing a SystemC TLM-2.0 compliant read access to the corresponding register of the Timer component. The same behavior can be achieved using the implicit initiator-side communication API as shown in Line 23. The SystemC TLM-2.0 compliance can be checked by integrating the SystemC TLM-2.0 base protocol checker [Ayn13] into the genericBus_TLM_Bus connection. After the current timer value has been read, the Testbench thread waits for ten clock periods (see Line 25). As the Testbench module is derived from the `ttlm::sc_ttlm_module` class which enables the TTLM timing API, calling the `wait(<time>)` function does not suspend the thread but increments its local simulation time according to the ATD semantics.

Table 4.5 summarizes the characteristics of the implementation of the simple example system. The specification of the architecture consists of 105 IP-XACT tags, which are transformed into 218 logical SLoC[1]. To implement the SystemC TLM-2.0 compliant data transfer, only 12 additional

[1] In this work, the logical Source Lines of Code (SLoC) have been quantified using the Metriculator Eclipse plugin [met13].

SLoC are required resulting in an overall SLoC number of 230. The fraction of generated code for this simple example system is 94.8% but heavily depends on the complexity of the functionality of the components.

| **implementation characteristic** | **value** |
|---|---|
| number of IP-XACT tags | 105 |
| logical SLoC of structural code | 218 |
| overall logical SLoC | 230 |
| fraction of generated code | 94.8% |

**Table 4.5.:** Implementation characteristics of the simple example system.

In the following section, the efficiency improvement achievable by the TTLM design flow is demonstrated using the automotive Night Vision system.

# 5. Application and Evaluation

In this chapter, the simulation performance improvement achievable by incorporating the Advanced Temporal Decoupling simulation technique as well as the implementation effort reduction achievable by employing the Transparent Transaction Level Modeling methodology are analyzed. Both methodologies are applied during the creation of a virtual prototype of the Night Vision system [Rob05]. The Night Vision system is a well-established driver assistance system for premium segment cars [Dai08]. In Section 5.1, a short overview of the Night Vision functionality and system architecture is given. This includes the presentation of a preexisting cycle accurate virtual prototype published in [Ban09]. This virtual prototype is used as a reference for the comparison of the results of the ATD and TTLM based virtual prototype presented in Section 5.2. The evaluation results contrasting both virtual prototypes are presented in Section 5.3.

## 5.1. Night Vision System and Preexisting Virtual Prototype

The objective of the Night Vision system is to improve the visibility of obstacles at challenging lighting conditions especially at night. Figure 5.1 gives an overview of the Night Vision system architecture as presented in [Rob05, Ban09].

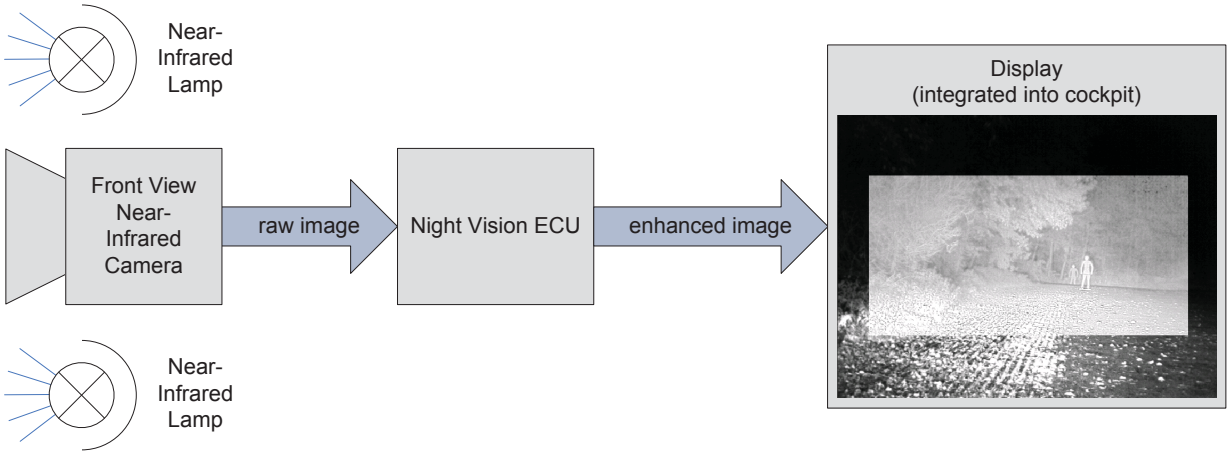



**Figure 5.1.:** Night Vision System Architecture (according to [Rob05, Ban09]). The front view near-infrared (NIR) camera captures a raw image of the scenery in front of the car which is illuminated by near-infrared lamps. The raw image is transferred to and processed by the Night Vision Electronic Control Unit (ECU). Finally, the enhanced image is transferred to the display which is integrated into the cockpit of the car.

Near-infrared lamps emitting light invisible for the human eye are used to illuminate the scenery in front of the car. The front view near-infrared camera captures a raw image at a constant frame rate comprising the near-infrared (NIR) and the visible spectrum. The raw image is transferred to the Night Vision Electronic Control Unit (ECU). The Night Vision ECU processes the raw image using a set of specific Night Vision algorithms. The enhanced image is transferred to the display which is integrated into the cockpit of the car.

The Night Vision ECU provides the following hardware features [Ban09]:

- dedicated FPGA used for the reception of the raw image from the camera, for basic video processing algorithms, and for the transfer of the enhanced image to the display,
- MPC5200 [Fre06] microcontroller containing a PowerPC [Fre94] CPU operated at 396 MHz having a 32 bit address bus and data bus. The CPU is used for complex video processing algorithms,
- PowerPC instruction cache and data cache having a size of 16 kB each,
- data cache supports modified-exclusive-invalid coherency protocol,
- 64 bit wide 60X Local Bus [Fre05] (XLB) processor bus [Fre05] operated at 132 MHz,
- Direct Memory Access (DMA) controller for CPU independent data transfers,
- 2x 8 MByte of external SDRAM, one connected to the FPGA and one to the PowerPC.

In [Ban09] a preexisting virtual prototype of the Night Vision system is presented which focuses on the simulation of the Night Vision ECU. The front view near-infrared (NIR) camera is emulated using a stub module which loads *.bmp image files and transfers them as a raw image stream to the Night Vision ECU model. The display is emulated using a graphics library bringing the enhanced image to the screen of the simulation host.

The preexisting virtual prototype is based on the module adapter methodology which is presented in [Ban09], too. The module adapters are intelligent communication sockets, which provide means to transfer arbitrarily sized application packets in a cycle accurate way. Therefore, the arbitrarily sized application packets are automatically divided into multiple bus word sized packets which are in turn transferred from the initiator module to the target module. Even though the module adapters are not based on the SystemC TLM-2.0 standard, the interfaces provided by the module adapters correspond to the abstraction levels commonly used in TLM (see Section 2.4.1). The interface used to connect the module adapter to the functional description of the module is comparable to the TLM - Programmer's View [MCG05] (TLM-PV) abstraction level. The interface related to the lower abstraction level is used to conduct the module-to-module communication based on bus word sized packets and is comparable to the TLM - Cycle Accurate [MCG05] (TLM-CA) abstraction level. To influence the timing aspects of the module-to-module communication, a set of communication timing parameters is provided.

Figure 5.2 shows the hardware architecture of the Night Vision ECU as implemented in the preexisting virtual prototype. The virtual prototype is divided into the FPGA subsystem and the PowerPC subsystem. The FPGA subsystem contains the camera FIFO and display link communication controllers, which are used to receive the raw image data from the NIR camera and to transfer the enhanced image to the display, respectively. The FPGA memory is connected via the FPGA bus and serves as a temporary storage for the raw and enhanced images. The

FPGA subsystem is connected to the PowerPC subsystem via a 32 bit wide PCI bus using the PCI to FPGA and XLB to PCI bus bridges.

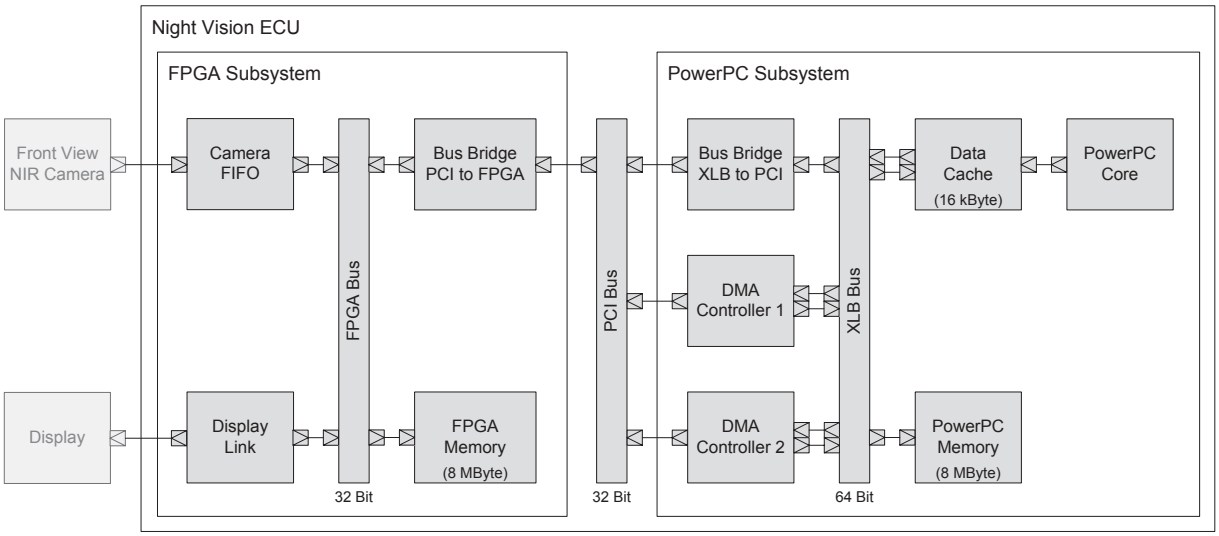



**Figure 5.2.:** Simplified block diagram of the Night Vision ECU architecture (based on [Ban09]). The FPGA subsystem which is used for basic video processing and data transfer is connected to the PowerPC subsystem via a PCI bus.

The main part of the PowerPC subsystem is the PowerPC core, which executes complex Night Vision video processing algorithms. To allow for a high conformance between the virtual prototype and the physical implementation of the Night Vision ECU, the original source code of the Night Vision algorithms as used in the series product is integrated into the PowerPC core model. This source code is executed in a host-compiled way, omitting the microarchitectural aspects of the PowerPC core. This includes the instruction cache and the instruction memory. In addition, the realtime operating system is omitted as it implies a negligible performance penalty of less than 5% according to measurements performed using the physical implementation of the Night Vision ECU [Ban09]. To mimic the time consumption which would arise from the execution of the Night Vision algorithms on a physical PowerPC Core or on a PowerPC ISS, the corresponding time consumption is either emulated using the module adapters or directly annotated to the original source code using `wait(<time>)` statements. The PowerPC core is connected to the 16 kByte sized, four way set associative data cache model. The PowerPC memory contains a set of frame buffers for the raw and enhanced images and is connected via the XLB bus. The PowerPC subsystem is completed by two DMA controller models representing the two channels of the DMA controller used in the physical microcontroller.

Each Night Vision frame is processed according to the five steps illustrated in Figure 5.3.

1. **Camera to FPGA memory:** The NIR camera sends the raw image data at a constant frame rate $t_{Frame\_rate}$. The camera FIFO buffers this data and performs basic video processing algorithms. The raw image data and the results of the basic Night Vision video processing algorithms are transferred to the FPGA memory.

2. **FPGA memory to PowerPC memory:** After $t_{DMA1\_offset}$ has elapsed and a significant

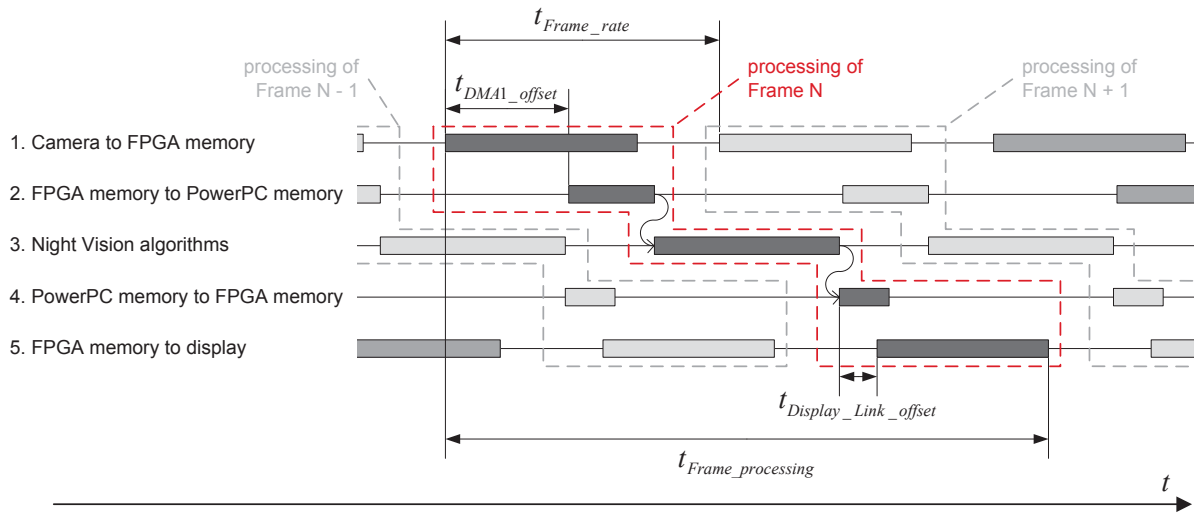



**Figure 5.3.:** Interleaved Night Vision frame processing (according to [Ban09]).

amount of the data belonging to one raw frame has been received from the NIR camera, the data transfer to the PowerPC subsystem is started. Therefore, the DMA controller 1 is used to copy the raw image data and the results of the basic video processing algorithm from the FPGA memory to the PowerPC memory via the FPGA bus, the PCI to FPGA bus bridge, the PCI bus, and the XLB bus.

3. **Night Vision algorithms:** After the raw image data and the results of the basic video processing algorithms have been copied to the PowerPC memory, the complex Night Vision video processing takes place. The raw image data is processed by a set of three Night Vision algorithms which are sequentially executed on the PowerPC core. The execution time of two Night Vision algorithms is constant, whereas the execution time of the third Night Vision algorithm depends on the content of the raw image frame [Ban09]. The resulting enhanced image is stored in the PowerPC memory.

4. **PowerPC memory to FPGA memory:** After the Night Vision processing is finished, the enhanced image frame is transferred from the PowerPC memory to the FPGA memory using the DMA controller 2.

5. **FPGA memory to display:** The transfer of the enhanced image frame to the display is started at $t_{Display\_Link\_offset}$ after the transfer of the enhanced image to the FPGA memory has been started. The display link module fetches the enhanced image frame from the FPGA memory and transfers it to the display.

The overall processing time of one frame is denoted as $t_{Frame\_processing}$. As shown in Figure 5.3, the five Night Vision processing steps in parts temporally overlap. Especially, this is the case for the frame transfer from the NIR camera to the FPGA memory and the transfer from the FPGA memory to the PowerPC memory. Similarly, the frame output to the display is started before the DMA based transfer of the enhanced image from the PowerPC memory to the FPGA memory is completed. As this temporal overlapping inherently occurs during the processing of a single frame, it is denoted as *intra-frame overlapping* hereafter.

As the time between two consecutive frames $t_{Frame\_rate}$ is lower than $t_{Frame\_processing}$, the processing of consecutive frames temporally overlaps as well. For example, the execution of the Night Vision algorithms for frame $N$ temporally overlaps with the transfer of the raw image data of frame $N+1$ from the NIR camera to the FPGA memory, as well as with the transfer of the enhanced image data of frame $N-1$ from the FPGA memory to the display. The temporal overlapping of processing steps belonging to different frames is denoted as *inter-frame overlapping*.

## 5.2. ATD and TTLM based Virtual Prototype

The recent ATD and TTLM based virtual prototype of the Night Vision system is used to evaluate the simulation performance improvement achievable by ATD and the implementation effort reduction achievable by TTLM.

The recent virtual prototype has been created using the TTLM design flow presented in Section 4.1. Figure 5.4 shows the structure of this virtual prototype which has been generated from a set of IP-XACT artifacts and equals to the structure of the preexisting virtual prototype. In contrast to the preexisting virtual prototype, the structural code of the recent virtual prototype uses a SystemC TLM-2.0 compliant communication infrastructure.

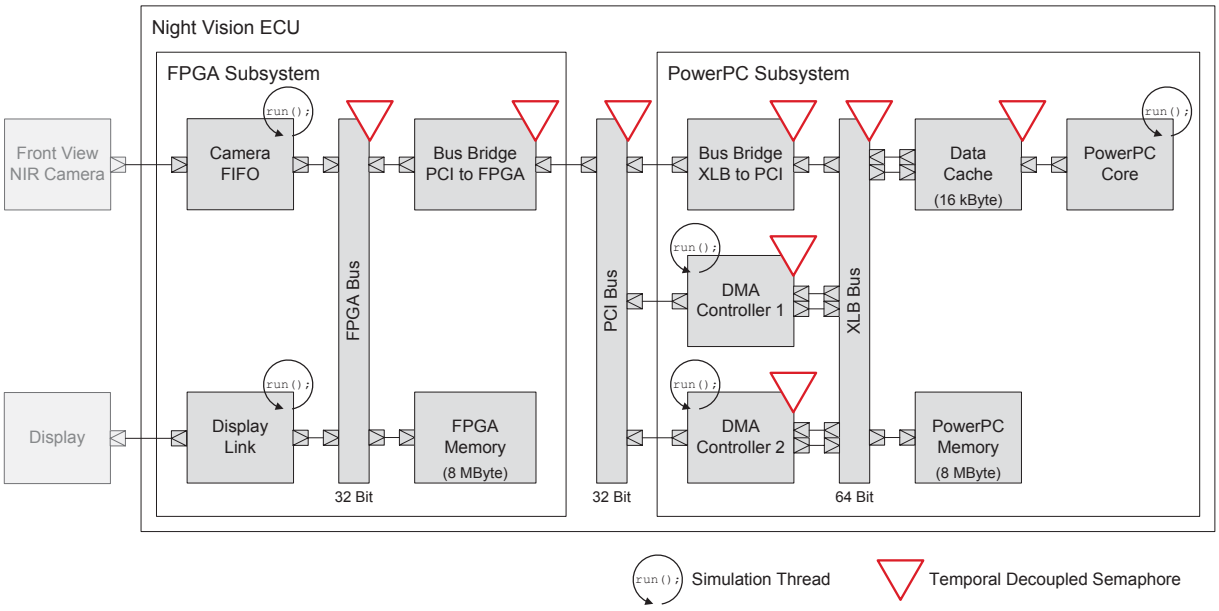



**Figure 5.4.:** Architecture of the ATD and TTLM based Night Vision ECU virtual prototype. To allow for an ATD based simulation, the shared resources contain temporal decoupled semaphores.

To allow for an ATD based simulation, the TTLM timing API as presented in Section 4.2.2 is incorporated by enabling the ATD support during TTLM generation. As a result, all modules of the recent virtual prototype are derived from the `ttlm::sc_ttlm_module` class instead of the `sc_core::sc_module` class. In addition, all shared resources which are either accessible via multiple target sockets or allow for asynchronous resource accesses contain temporal decoupled semaphores. Remarkably, neither the FPGA memory nor the PowerPC memory contain a

temporal decoupled semaphore. This is because both memories only comprise one target socket and only allow for synchronous accesses. Additionally, the memory modules are only accessible via the corresponding buses which are in turn shared resources containing a temporal decoupled semaphore which ensures mutual exclusive access to the corresponding bus and the connected memory too.

Similar to the preexisting virtual prototype, the PowerPC core module of the ATD and TTLM based virtual prototype contains a simulation thread executing the original source code of the Night Vision algorithms as used in the series product. As this source code is executed in a host-compiled way, an appropriate source code instrumentation is required to redirect the memory accesses arising from the software to the bus interface of the PowerPC core module. To achieve this redirection, the implicit initiator side communication API provided by the TTLM library is used (see Section 4.2.1.1).

Besides the simulation thread contained in the PowerPC core module, there are four additional simulation threads each implementing an independent course of action of the Night Vision system. The threads contained in the camera FIFO and in the display link modules are used to implement the raw image transfer to the FPGA memory and the enhanced image transfer to the display, respectively. Similar to the preexisting virtual prototype, the transfer from the NIR camera is emulated by using a stub module loading a *.bmp file and the transfer to the display is emulated redirecting the enhanced image data to the simulation host screen. The threads contained in the DMA controllers are used to implement the data transfers between the FPGA memory and the PowerPC memory. The control flow between these simulation threads is implemented using the ATD enabled `ttlm::sc_ttlm_event` class.

The described deployment of the temporal decoupled semaphores and the simulation threads results in the eight-by-five shaped SRA matrix with an overall number of 40 cells shown in Figure 5.5. The architecture of the Night Vision system inhibits the usage of 18 matrix cells as no communication channels from the module containing the corresponding thread to the module containing the corresponding TDSem are available. For example, none of the threads contained in the camera FIFO and display link modules can access the TDSems contained in the PowerPC subsystem. Even though the Night Vision system architecture allows for accesses from the corresponding thread to the corresponding slave, 5 of the matrix cells remain unused by the application. Therefore, only 17 out of the 40 matrix cells are actually used by the application.

The five simulation threads imply a set of five SRA trees. As many of the algorithms executed during the ATD transaction processing phase traverse the SRA trees towards the SRA tree root, the height of the SRA trees has a direct impact on the overall simulation performance. The SRA trees associated to the threads of the camera FIFO and the display link modules have a height of one. This is because the FPGA memory which is the only shared resource used by these threads is accessible directly without requiring cascaded resource accesses. The SRA trees belonging to the threads contained in the DMA controllers have a height of three as accesses to the FPGA bus and the FPGA memory involve traversing the PCI bus and the PCI to FPGA bus bridge. The height of the SRA tree belonging to the thread of the PowerPC core module is five and is only used to its full extent during the configuration of the PCI to FPGA bus bridge which takes place during system power up. After the system power up procedure is finished, the thread of the PowerPC

| TDSem \ Thread | Camera FIFO | DMA 1 | DMA 2 | PowerPC Core | Display Link |
|---|---|---|---|---|---|
| FPGA Bus & Memory | | | | | |
| Bus Bridge PCI to FPGA | | | | | |
| PCI Bus | | | | | |
| Bus Bridge XLB to PCI | | | | | |
| XLB Bus & Memory | | | | | |
| Data Cache | | | | | |
| DMA 1 | | | | | |
| DMA 2 | | | | | |

= the architecture of the virtual prototype inhibits access (e.g. no communication link available)
= the application does not make use of the accessibility provided by the architecture
= accesses take place

**Figure 5.5.:** SRA Matrix of the Night Vision application.

core primarily accesses the PowerPC memory via the data cache and the XLB bus resulting in a SRA tree height of two.

## 5.3. Evaluation Results

### 5.3.1. Implementation effort reduction

In this section, the reduction of the implementation effort achievable by the employment of the TTLM methodology is investigated. According to [Ban09], the creation of the module adapter based preexisting virtual prototype of the Night Vision system and all of its contained modules required six months approximately. This includes the code required for the structure of the Night Vision system and its contained modules as well as the code required for the implementation of the corresponding functionality.

The creation of the ATD and TTLM based virtual prototype approximately required three months. Thereby, the implementation effort needed for the creation of the functionality is only reduced slightly, whereas the implementation effort for the creation of the structural code is reduced considerably. The slight effort reduction concerning the implementation of the functionality arises from the different data transfer technique used in ATD compared to the module adapter approach. The module adapters are designed to split up the arbitrarily sized application packets sent by the initiator into multiple bus words, which are successively transferred and eventually need to be merged on target-side. In contrast, in ATD the application packets are not split up automatically, superseding the merging of the received data on target-side.

The implementation effort reduction concerning the structural code arises by the usage of the TTLM generator. As shown in Table 5.1, the structural aspects of the Night Vision system including the memory maps and the interconnection of all modules are described using a set of 1145 IP-XACT tags. Using the TTLM code generator, the IP-XACT descriptions are transformed

into 2392 logical SLoC[1] of structural SystemC, TTLM, SCML2, and ATD code. Considering the overall logical SLoC of 5110, this results in a fraction of 46.8% of generated code.

| implementation characteristic | ATD and TTLM based virtual prototype | preexisting virtual prototype |
|---|---|---|
| number of IP-XACT tags | 1145 | n.a. |
| logical SLoC of structural code | 2392 | n.a. |
| overall logical SLoC | 5110 | 6961 |
| fraction of generated code | 46.8% | 0% |
| overall implementation duration | ≈ 3 months | ≈ 6 months [Ban09] |

**Table 5.1.:** Comparison of the implementation characteristics of the virtual prototypes of the Night Vision system.

Besides the virtual prototype of the Night Vision system, the TTLM generator has been successfully used for various other virtual prototypes. The number of IP-XACT tags required for the specification of the structural aspects of these virtual prototypes varied from a few hundred up to approximately 38.000 IP-XACT tags.

### 5.3.2. Simulation performance improvement

In this section, the simulation performance improvement of the virtual prototype of the Night Vision system achievable by ATD is investigated[2]. As shown in Section 3.7, the simulation performance improvement achievable by using ATD depends on various factors. One of these factors is the value of the $SRA_{quantum}$, which specifies the maximum number of pending SRAs per simulation thread before synchronization is forced. To achieve the highest simulation performance the $SRA_{quantum}$ value has been set to three.

Besides the freely selectable value of the $SRA_{quantum}$, there are other factors which depend on the characteristics of the implemented virtual prototype like the size of the resulting SRA matrix and the respective heights of the SRA trees. Furthermore, the size of the application packets registered by the threads has a significant impact on the simulation performance. According to the ATD semantics, application packets might only be disaggregated into multiple smaller application packets but multiple application packets are not aggregated to one larger application packet. Thus, the size of the application packets registered by the threads determines the maximum size of the application packets processed as a whole. In turn, this limits the achievable simulation speed up compared to the cycle accurate simulation of the preexisting virtual prototype.

In the following, the simulation runtime for each of the five processing steps of the Night Vision system is analyzed. To increase the accuracy of the simulation time measurement, the host CPU intensive emulation of the display module using a graphics library is deactivated. According to Figure 5.3, the transfer of each frame from the camera to the FPGA memory partially overlaps

[1] In this work, the logical Source Lines of Code (SLoC) have been quantified using the Metriculator Eclipse plugin [met13].

[2] Both virtual prototypes have been executed using the simulation host characteristics shown in Appendix C.

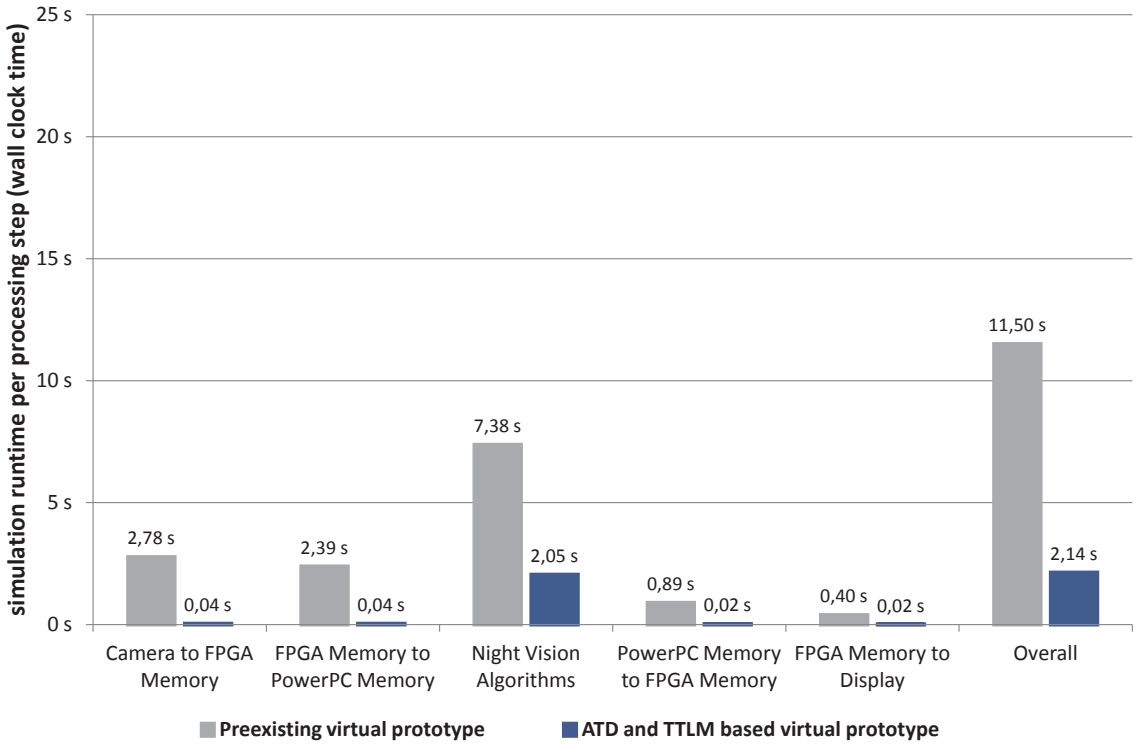
with the execution of the Night Vision algorithms of the previous frame. Similarly, the transfer of the current frame from the FPGA memory to the display partially overlaps with the execution of the Night Vision algorithms of the successive frame. To temporarily eliminate the interference between the execution of the Night Vision algorithms for one frame and the data transfers of the adjacent frames, the inter-frame overlapping is removed by increasing the time interval between two consecutive frames $t_{Frame\_rate}$ to a value larger than $t_{Frame\_processing}$. Figure 5.6 (a) compares the simulation runtime (wall clock time) required by the preexisting virtual prototype and the ATD and TTLM based virtual prototype for each of the Night Vision processing steps. Due to the still occurring intra-frame overlapping caused by the temporally overlapping data transfers from and to the FPGA memory, the sum of the simulation runtimes (wall clock time) required for each processing step is larger than the overall simulation runtime (wall clock time) required for the processing of a single frame. Figure 5.6 (b) shows the speed up factor achievable by ATD compared to the cycle accurate implementation of the preexisting virtual prototype.

The raw image frame delivered by the camera consists of 640 x 480 pixels where each pixel is represented by two bytes. This results in 153600 words having 32 bits each which are to be transferred to the FPGA memory. This transfer takes place using data blocks comprising 32 FPGA bus words. The execution of this procedure takes $2.78s$[3] using the preexisting virtual prototype and $0.04s$ using the ATD and TTLM based virtual prototype. The significant difference between these durations results from the fact, that in case of the preexisting virtual prototype each word of each 32 word sized data block is transferred separately entailing various context switches between the threads contained in the involved module adapters. In case of the ATD and TTLM based virtual prototype, each data block is treated as one preemptable SRA. Especially during the time before the frame transfer from the FPGA memory to the PowerPC memory is started, there is no further independent course of action. Thus, the time budget for the transfer of each data block is not limited and each data block is transferred en bloc. After the frame transfer from the FPGA memory to the PowerPC memory has been started, access conflicts to the FPGA bus occur. These conflicts have to be resolved by the TDSem integrated in the FPGA bus. The ATD based transfer of the raw image frame from the camera to the FPGA memory results in a simulation runtime reduction factor of 63.1 compared to the cycle accurate implementation used in the preexisting virtual prototype.

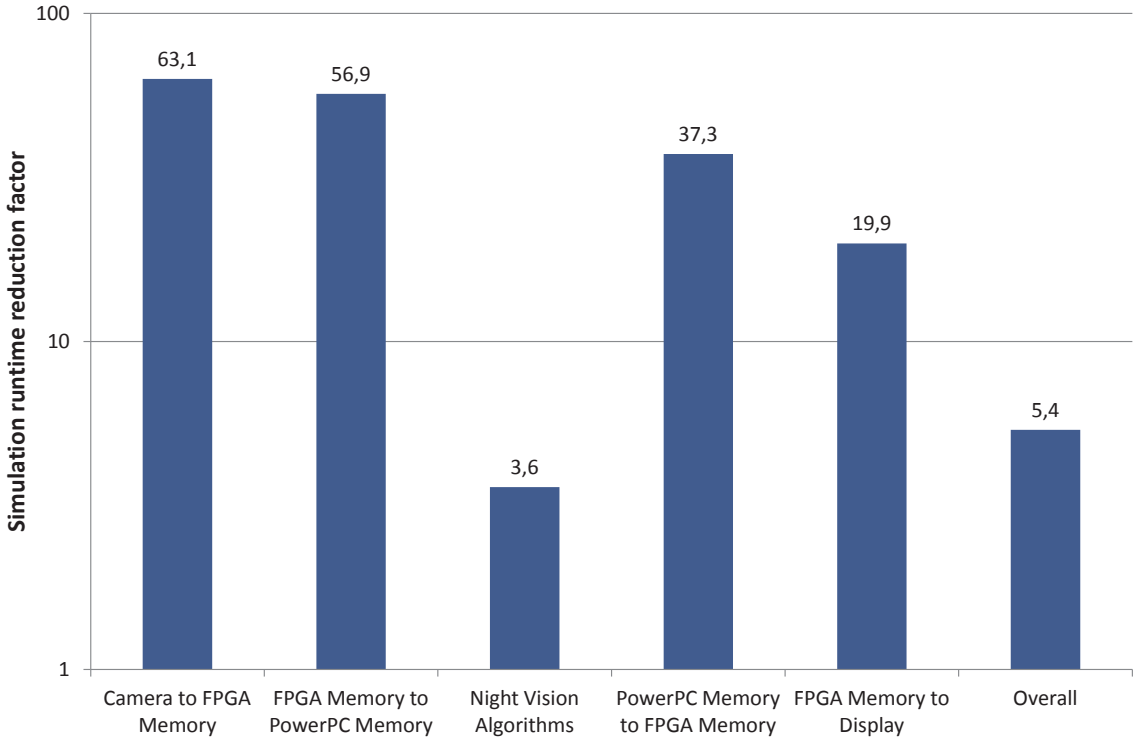
The DMA based transfer of the raw image and the results of the basic Night Vision video algorithms from the FPGA memory to the PowerPC memory takes place using data blocks consisting of 128 bytes each. This results in a data block size of 32 words on the 32 bit wide FPGA and PCI buses and a data block size of 16 words on the 64 bit wide XLB bus. The comparison of the durations resulting from the execution of this procedure using the preexisting virtual prototype ($2.39s$) and the ATD and TTLM based virtual prototype ($0.04s$) results in a simulation runtime reduction factor of 56.9.

The execution of the Night Vision algorithms requires the longest simulation time. Using the preexisting virtual prototype, the execution of the Night Vision algorithms takes $7.38s$ in contrast

[3] In this work, simulation runtimes have been quantified using `clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ..)` [get14], which allows for high resolution CPU time measurement. To mitigate the effect of simulation runtime deviations between multiple executions of the same simulation, the average simulation runtime of ten simulation runs are presented.




(a) Simulation runtime per Night Vision processing step required for the simulation of one frame.




(b) The large number of bus word sized transactions between the PowerPC core and the data cache during the processing of the Night Vision algorithms limit the achievable simulation runtime reduction.

**Figure 5.6.:** Simulation performance comparison **without inter-frame overlapping**.

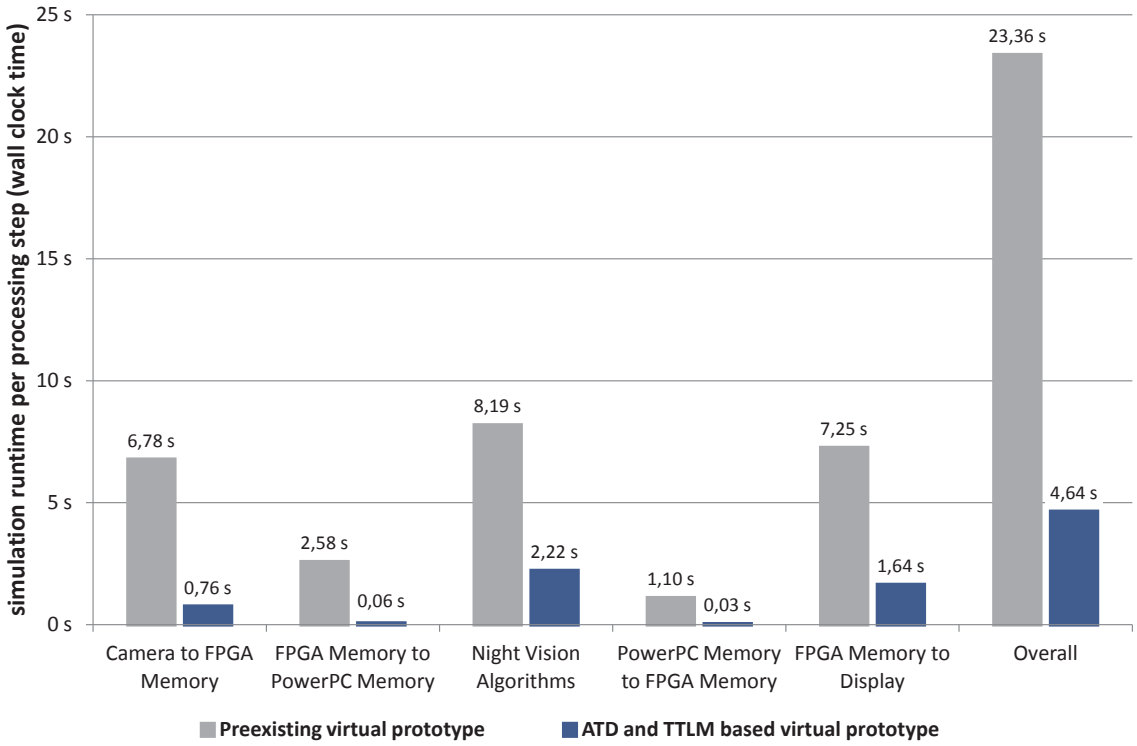
to 2.05$s$ required by the ATD and TTLM based virtual prototype. During this phase, the majority of transactions takes place between the PowerPC core and the data cache. As each of the data cache accesses caused by the Night Vision software comprises a single word only, the simulation runtime reduction achievable by ATD only results from the temporal decoupled execution of the thread contained in the PowerPC core module. Besides the transactions between the PowerPC core and the data cache, communication takes place between the data cache and the PowerPC memory in case of a data cache miss. Each cache line of the data cache contains 64 bytes of data. In combination with the 64 bit data bus width of the XLB bus, loading or storing a data cache line leads to eight word sized burst transfers between the data cache and the PowerPC memory. Similar to the burst transfers occurring during the frame transfer from the camera to the FPGA memory and from the FPGA memory to the PowerPC memory, the simulation of the burst transfers resulting from the data cache misses can be accelerated by ATD. Despite of the low average transaction size during the execution of the Night Vision algorithms, the simulation runtime reduction factor is still 3.6.

Similar to the transfer of the raw image from the FPGA memory to the PowerPC memory, the DMA based transfer of the enhanced image from the PowerPC memory to the FPGA memory takes place using a DMA buffer size of 128 bytes. The enhanced image consists of 800 x 400 pixels where each pixel is represented by one byte instead of two bytes as it is the case for the raw image. Therefore, only 40000 words on the XLB bus and 80000 words on the FPGA and PCI buses need to be transferred. The DMA buffer size of 128 bytes results in a data block size of 32 words on the FPGA and PCI buses and 16 words on the XLB bus. Due to the lower amount of data which is to be transferred, the required simulation time is lower than the simulation time required for the transfer of the raw image. This step takes 0.89$s$ using the preexisting virtual prototype and 0.02$s$ using the ATD and TTLM based virtual prototype, resulting in a simulation runtime reduction factor of 37.3.

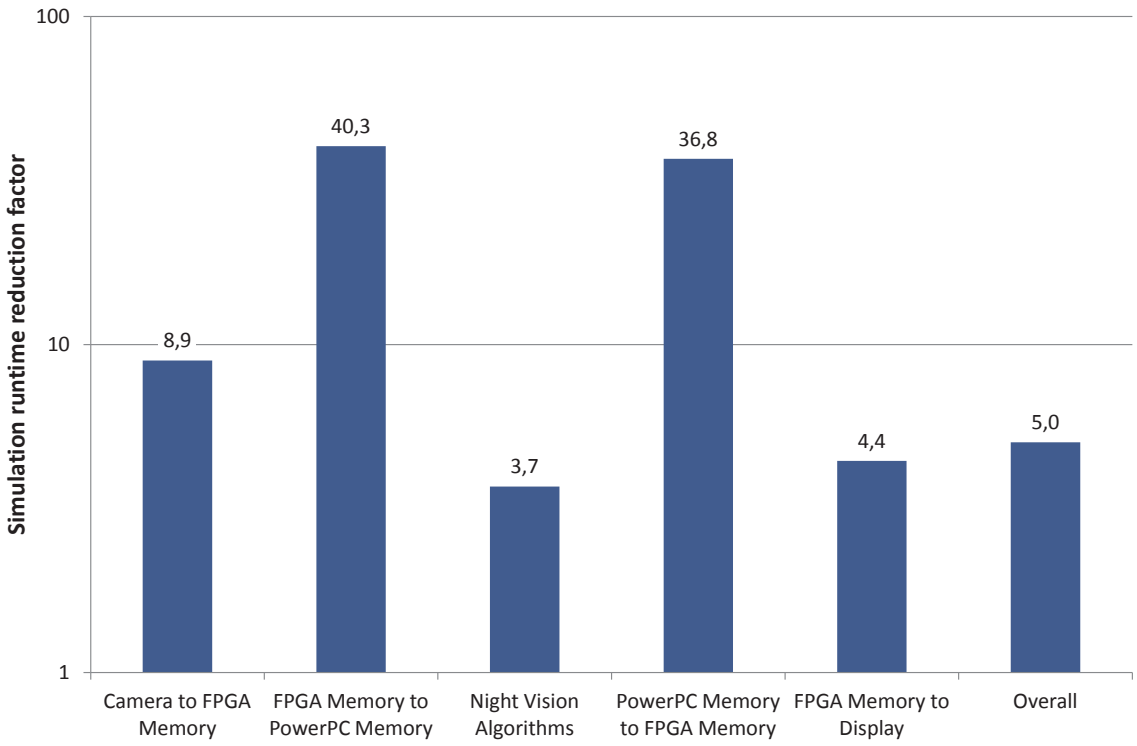
The transfer of the enhanced image from the FPGA memory to the display via the FPGA bus and the display link module takes place using data blocks consisting of 64 bytes resulting in burst transfers of 16 FPGA bus words each. This transfer requires 0.4$s$ of simulation runtime using the preexisting virtual prototype and 0.02$s$ using the ATD and TTLM based virtual prototype. Similar to the simulation runtime reduction of the other data transfer intensive Night Vision steps, the simulation runtime reduction factor of this last step is high (19.9).

Overall, the simulation of the processing of a single frame without the interference caused by the inter-frame overlapping using the preexisting virtual prototype requires 11.5$s$ (wall clock time). In contrast, the simulation of the processing of the same frame using the ATD and TTLM based virtual prototype requires 2.14$s$ (wall clock time) only. Thus, the overall simulation runtime reduction factor is 5.4. This simulation speed up results from the high acceleration of block based data transfers as well as the temporal decoupled execution of single word transactions achievable by ATD.

After comparing the simulation performance of the virtual prototypes of the Night Vision at frame repetition times longer than the frame processing time, the effects of the inter-frame overlapping are investigated. Therefore, the offset between the start times of the frame transfers of adjacent frames from the NIR camera to the FPGA memory $t_{Frame\_rate}$ is reset to its original value.




(a) Simulation runtime per Night Vision processing step required for the simulation of one frame.




(b) Simulation runtime reduction.

**Figure 5.7.:** Simulation performance comparison **taking inter-frame overlapping into account**.

Figure 5.7 (a) shows the simulation runtimes per processing step in case the inter-frame overlapping occurs. Figure 5.7 (b) shows the resulting simulation runtime reduction factors.

Due to the resulting pipelined frame processing, the transfer of the current frame from the camera to the FPGA memory and the transfer from the FPGA memory to the display in parts temporally overlaps with the execution of the Night Vision algorithms of the adjacent frames on the PowerPC core. Therefore, the simulation time required for the concurrently executed processing of the Night Vision algorithms of the adjacent frames is partially contained in the measured simulation time required for the transfers of the current frame.

Besides the influence of the concurrent execution of the Night Vision algorithms of the adjacent frames, the pipelined frame processing caused by the inter-frame overlapping has another effect on the simulation performance. According to Section 3.6, the calculation of the time budget requires the consideration of the time of the next SystemC event and the start times of any pending SRAs to allow for full cycle accuracy. In case of the Night Vision prototype, the implementation of the shared resource allows for the preemption of the current SRA. Therefore, many unnecessary SRA execution interruptions occur caused by SRAs and events resulting from the processing of another frame.

The temporal overlapping of the execution of the Night Vision algorithms with the data transfers from the camera to the FPGA memory and from the FPGA memory to the display of adjacent frames as well as the reduction of the average time budget available for the transfer of the data blocks during those frame transfers, have a negative impact on the simulation performance. The simulation runtime required for the transfer from the camera to the FPGA memory rises from $2.78s$ to $6.78s$ for the preexisting virtual prototype and from $0.04s$ to $0.76s$ for the TTLM and ATD based virtual prototype. For the transfer from the FPGA memory to the display, the simulation runtimes are increased from $0.4s$ to $7.25s$ and from $0.02s$ to $1.64s$, respectively. The simulation runtime reduction factors of the frame transfers are reduced from 63.1 to 8.9 and from 19.9 to 4.4, respectively. The overall simulation runtime per frame rises from $11.5s$ to $23.36s$ for the preexisting virtual prototype and from $2.14s$ to $4.64s$ for the ATD and TTLM based virtual prototype. The overall simulation runtime reduction factor is therefore reduced from 5.4 to 5.0.

# 6. Conclusion and Future Work

Due to the exponential growth of the number of transistors which can be integrated into a single chip [Moo65] and in consequence the rising complexity of today's electronic systems, the development of electronic systems becomes more and more challenging. This requires the employment of new efficiency-raising design methods like virtual prototyping. The creation of a virtual prototype requires finding a suitable trade-off between partially contradictory optimization objectives including simulation timing accuracy, simulation performance, and modeling efficiency.

## 6.1. Conclusion and Discussion

In this work, a framework for the efficient creation of accurate and high-performance virtual prototypes has been presented. This framework consists of two techniques denoted as Advanced Temporal Decoupling (ATD) and Transparent Transaction Level Modeling (TTLM). The ATD simulation methodology presented in Chapter 3 allows for a cycle accurate shared resource access management in the context of temporal decoupling. As shown in Sections 3.7 and 5.3.2, the simulation performance achievable by an ATD based virtual prototype is significantly higher than the simulation performance achievable by a virtual prototype based on conventional cycle accurate TLM. To achieve this simulation performance advantage, the ATD processing strategy for shared resource accesses fundamentally differs from the shared resource access processing strategy used in conventional cycle accurate TLM models. In conventional cycle accurate TLM models, shared resource accesses are typically implemented using sequences of atomic and single bus word sized transactions. The atomic transactions are successively arbitrated and processed by the shared resource allowing for the preemption of the shared resource accesses at atomic transaction borders. In ATD, the shared resource accesses do not need to be divided into atomic transactions. Instead, the shared resource accesses are registered at temporal decoupled semaphores. The temporal decoupled semaphores exploit the look-ahead arising from the temporal decoupled simulation to calculate the time budget available for the processing of the shared resource access before preemption occurs. In case the time required to entirely process a shared resource access exceeds the time budget, the time budget can be used to implement cycle accurate preemption. As a result of ATD, the size of the processed shared resource access fragments is as small as needed to obtain cycle accurate simulation results and as large as possible to allow for high simulation performance. Thereby, ATD provides comprehensive support of shared resource and shared resource access properties including the preemption of shared resource accesses, synchronous and asynchronous shared resource accesses, and inter-resource dependencies.

The TTLM design flow presented in Chapter 4 allows for the efficient creation of virtual prototypes. This design flow consists of a modeling library providing a set of convenience functions and

an accompanying code generator. The TTLM library provides means for encapsulating SystemC, TLM-2.0, and SCML2 specific communication code. Additionally, the TTLM library provides a set of programming interfaces for the integration of the ATD simulation technique, for runtime configuration, and for runtime information extraction. The employment of these programming interfaces allows the designer to focus on the implementation of the functionality of the virtual prototype without requiring SystemC expert knowledge. The TTLM generator is capable of generating the structural code of the virtual prototype including the register architecture and the top-level netlist. The required information is retrieved from a set of IP-XACT models representing a formal specification of the structural aspects of the virtual prototype. The resulting model is compliant to the SystemC standard in general and to the SystemC TLM-2.0 standard in particular [IEE11]. Therefore, they can be used in combination with models implemented without ATD or TTLM.

Similar to the classification of the conventional TLM modeling styles shown in Figure 2.9 on page 23, the classification of the ATD and TTLM based virtual prototypes is shown in Figure 6.1. As the actual values for the shown metrics heavily depend on the characteristics of the modeled system, only a qualitative classification is given.

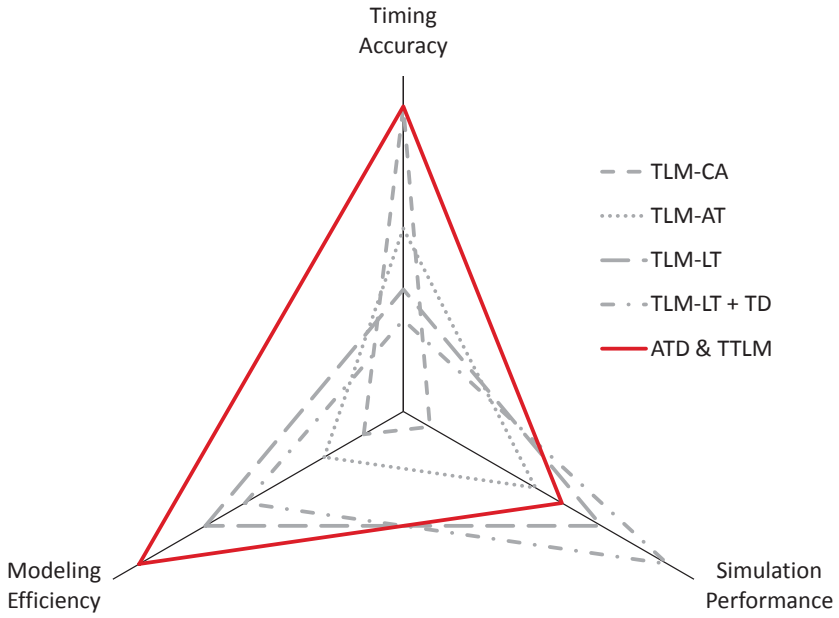



**Figure 6.1.:** Qualitative comparison of ATD and TTLM with conventional TLM modeling styles.

The simulation timing accuracy of ATD enabled models exceeds the simulation timing accuracy achievable by approximately-timed (TLM-AT), loosely-timed (TLM-LT), and temporal decoupled loosely-timed TLM models (TLM-LT + TD) and is equal to the timing accuracy achievable by cycle accurate TLM models (TLM-CA). Depending on the characteristics of the modeled system, the simulation performance of ATD enabled models can be significantly higher than the simulation performance achievable by TLM-CA. However, ATD enabled models in most cases will be slower than TLM-LT and TLM-LT + TD models. Due to the implementation effort reduction achievable by the incorporation of the TTLM library and the TTLM generator, the modeling efficiency can

be raised substantially regardless to which TLM modeling style it is compared to.

Besides the benefits of ATD and TTLM compared to conventional TLM modeling styles, there are some challenges and limitations when incorporating these methods. The usage of the TTLM generator and the virtual prototype configuration API provided by the TTLM library requires knowledge about IP-XACT which allows for the formal specification of structural aspects of digital hardware.

The application of ATD to legacy models requires code changes. For instance, the syntax and semantic of the `executeSRA(..)` function required for the implementation of ATD enabled shared resources differs from the syntax and semantic of the `(n)b_transport(..)` function used in conventional TLM modeling styles. Especially it is required to incorporate the time budget calculated by ATD to implement the preemption of shared resource accesses. Besides the changes concerning the interfaces used for the implementation of shared resource accesses, ATD relies on the existence of the advance time notification interface which has to be provided by the underlying SystemC kernel. The advance time notification is used for the separation of the temporal decoupled thread execution phase and the transaction processing phase. Although there is a non-intrusive method to implement the advance time notification interface (see Appendix A.1), the SystemC kernel needs to be adapted in order to achieve the best simulation performance when using ATD.

As shown in Sections 3.7 and 5.3.2, the simulation performance improvement achievable by ATD depends on the characteristics of the modeled system. To achieve a high simulation performance when using ATD, the size of the shared resource accesses requested by an initiator has to be maximized. The larger the size of the shared resource accesses, the larger the size of the shared resource access fragments which can be executed en bloc in case sufficient time budget is available. If the size of the shared resource accesses is low, there is no possibility to process large shared resource access fragments en bloc and the simulation performance will only slightly outrun the simulation performance achievable by a conventional cycle accurate TLM model. The differences of the simulation runtime reduction between the processing of the Night Vision algorithms and the data transfer centric steps shown in Figure 5.6 (b) on page 99 illustrates the impact of a low shared resource access size on the simulation performance. However, the achievement of cycle accurate simulation results is not compromised by the size of the shared resource accesses requested by the initiator. Instead, cycle accuracy is guaranteed by ATD without any additional user implementation required.

Besides the impact of a low shared resource access size, the simulation performance achievable by ATD depends on the time budget available for the processing of pending shared resource accesses. The time budget calculation mechanism presented in Section 3.6.3 is restrictive. As shown in Figure 3.18, the restrictive time budget calculation policy even allows for the cycle accurate simulation of preemptions resulting from the future execution of pending SRAs on different shared resources. However, the restrictive time budget calculation policy increases the number of SRA processing interruptions. In case the later execution of pending SRAs does not lead to the creation of SRAs which will preempt the currently executed SRA, these interruptions are unnecessary. Even though these unnecessary interruptions of the SRA processing have no impact on the simulation timing accuracy, they have a negative impact on the simulation performance.

## 6.2. Future Work

To overcome the negative simulation performance impact of the restrictive ATD time budget calculation algorithm presented in Section 3.6.3, further application specific knowledge could be taken into account during the calculation of the time budget. Considering the example shown in Figure 3.18 on page 57, it might be the case that the creation of a SRA during the future execution of a pending SRA which will preempt the currently executed SRA is precluded. As shown in the SRA matrix of the ATD and TTLM based virtual prototype of the Night Vision system (see Figure 5.5 on page 96), this preclusion can result from architectural limitations like the absence of appropriate communication channels. One possible extension of the ATD concept would be to automatically take these architectural limitations into account during the calculation of the time budget by ignoring the start times of pending SRAs which definitely can not lead to the preemption of the currently executed SRA.

Besides the possible influence of the architectural aspects on the time budget calculation, application specific characteristics leading to an increased time budget could be taken into account. Therefore, ATD might be extended by an additional API allowing the designer to provide additional information about the characteristics of the available shared resources. For example, the API might allow the classification of the shared resources into reacting and non-reacting shared resources. Accesses to reacting shared resources might lead to cascaded shared resource accesses, whereas accesses to non-reacting shared resources will not lead to cascaded shared resource accesses and therefore do not need to be considered during the time budget calculation. In addition, the API might allow the classification of the shared resources into preemptable and non-preemptable shared resources. In case $SRA_{next}$ is to be executed on a non-preemptable shared resource, the time budget calculation might be omitted.

A further simulation performance improvement might be achieved by the parallelization of the execution of the simulation threads during the temporal decoupled thread execution phase using one of the techniques presented in the first paragraph of Section 2.4.5.1. As the set of synchronization points defined for the ATD based thread execution semantics (see Table 3.1 on page 37) is a subset of the synchronization points required in conventional TLM models, ATD reduces the amount of required synchronization compared to conventional TLM models and therefore allows for a more efficient parallel execution of the simulation threads.

# Appendices

# A. API Reference

## A.1. Advance Time Notification Interface

ATD depends on the invocation of the `AdvanceTimeNotification_if` by the underlying simulator. This interface comprises following abstract function

```
virtual void onAdvanceTime(const sc_time &timeOfNextEvent) = 0;
```

There are two alternative implementations for the invocation of this interface. The "delta-cycle hopping" implementation is shown in the following listing

```
1  // modules constructor:
2  ...
3  SC_THREAD(AdvanceTimeNotifier);
4  ...
5  // delta-cycle hopping thread:
6  void AdvanceTimeNotfier(void)
7  {
8    while(true) {
9      if ((!sc_pending_activity_at_current_time()) &&
           (sc_time_to_pending_activity() > 0)) {
10       onAdvanceTime(sc_time_to_pending_activity()+sc_time_stamp());
11       wait(sc_time_to_pending_activity())
12     } else {
13       wait(SC_ZERO_TIME);
14     }
15   }
16 }
```

This implementation incorporates an `SC_THREAD` which is resumed during each delta cycle and checks for pending activity at current simulation time. If there is pending activity at current simulation time the thread reschedules itself to the next delta cycle. If there is no pending activity at current simulation time, it invokes the Advance Time Notification Interface and waits until time of next event. Using this approach, the SystemC library remains unchanged. Unfortunately, there are some shortcomings of the "delta-cycle hopping" approach:

- it is very inefficient as it incorporates one additional context switch per delta cycle

- the number of AdvanceTimeNotifier threads is limited to one per simulation

Alternatively, the invocation of the Advance Time Notification Interface can be integrated into the SystemC library. This is done by replacing

```
if( m_runnable->is_empty() ) {
  // no more runnable processes
  break;
}
```

in notification phase of `sc_simcontext::crunch(bool)` function by

```
1 if( m_runnable->is_empty() ) {
2   // no more runnable processes
3   onAdvanceTime(sc_time_to_pending_activity()+sc_time_stamp());
4   if(( m_runnable->is_empty() ) && (m_delta_events.size() == 0)) {
5     // no threads have become runable during onAdvanceTime call
6     break;
7   }
8 }
```

Additionally, the **do .. while**(..) loop at the end of `sc_simcontext::simulate(const sc_time&)` function has to be removed.

## A.2. Get Parameter Interface

```
1 template <class TYPE>
2 bool Config::getParameter(const std::string instanceName, const
     std::string parameterName, TYPE &value);
```

The `getParameter` interface is part of the TTLM configuration API and is used to obtain parameter values from an external data source. The `instanceName` parameter specifies the name of the netlist instance which provides the parameter defined by `parameterName`. The `value` parameter contains the parameter value as obtained from the external data source.

# B. Code Listings

## B.1. ATD Benchmark

For simulation host characteristics see Appendix C. To reduce the printed code size and to eliminate the processing overhead of the TLM sockets, the decomposition into multiple modules connected by TLM sockets has been omitted.

### Conventional TLM-CA Implementation of Simple Example System

```
1  #define PROGRAMM_CYCLES 20000000
2  #define CLOCK_PERIOD sc_core::sc_time(50,sc_core::SC_NS)
3
4  class Access {
5  public:
6    tlm::tlm_generic_payload *data;
7    int priority;
8  };
9
10 class TLM_CA : public sc_core::sc_module {
11 public:
12   SC_HAS_PROCESS(TLM_CA);
13   TLM_CA(sc_module_name mn, int wordsPerAppPacket) : sc_module(mn)  {
14     m_noOfWordsPerApplicationPacket = wordsPerAppPacket;
15     SC_THREAD(thread0);
16     SC_THREAD(thread1);
17     SC_THREAD(arbitrate);
18     p_payload0 = new tlm::tlm_generic_payload;
19     p_payload1 = new tlm::tlm_generic_payload;
20     buffer0 = 0;
21     buffer1 = 0;
22     dataCounter0 = 0;
23     dataCounter1 = 0;
24   }
25
26   void thread0() {
27     for (int cycle = 0; cycle < PROGRAMM_CYCLES; cycle++) {
28       sc_core::wait(2*CLOCK_PERIOD); // initial delay
29       for (unsigned int i = 0; i < m_noOfWordsPerApplicationPacket; i++) {
30         buffer0 = dataSend0++;
31         p_payload0->set_data_length(1);
32         p_payload0->set_data_ptr((unsigned char *)&buffer0);
33         accessResourcebyThread0(p_payload0,0);
34       }
35       if (m_noOfWordsPerApplicationPacket < 3) {
36         sc_core::wait(2*CLOCK_PERIOD);
37       }
38     }
```

```
39      sc_core::wait();
40    }
41
42    void thread1() {
43      for (int cycle = 0; cycle < PROGRAMM_CYCLES; cycle++) {
44        sc_core::wait(4*CLOCK_PERIOD); //initial delay
45        for (unsigned int i = 0; i < m_noOfWordsPerApplicationPacket; i++) {
46          buffer1=dataSend1++;
47          p_payload1->set_data_length(1);
48          p_payload1->set_data_ptr((unsigned char *)&buffer1);
49          accessResourcebyThread1(p_payload1,1);
50        }
51        if (m_noOfWordsPerApplicationPacket >= 3) {
52          sc_core::wait((m_noOfWordsPerApplicationPacket - 2)*CLOCK_PERIOD);
53        }
54      }
55      sc_core::wait();
56    }
57
58    void arbitrate() {
59      boost::container::list<Access *>::iterator it;
60      while(true) {
61        sc_core::wait(m_receivedTransaction_ev);
62        while(accessList.size() != 0) {
63          Access *highestPrioAccess = accessList.front();
64          for (it = accessList.begin(); it != accessList.end(); it++) {
65            if (highestPrioAccess->priority < (**it).priority)
66              highestPrioAccess = (*it);
67          }
68          //mimic work
69          if (highestPrioAccess->priority == 1) {
70            memcpy(&dataRcv1,highestPrioAccess->data->get_data_ptr(),sizeof(unsigned long long));
71            m_thread1Granted_ev.notify();
72          } else {
73            memcpy(&dataRcv0,highestPrioAccess->data->get_data_ptr(),sizeof(unsigned long long));
74            m_thread0Granted_ev.notify();
75          }
76          //mimic duration of the above data processing
77          sc_core::wait(CLOCK_PERIOD);
78          accessList.remove(highestPrioAccess);
79        }
80      }
81    }
82
83    void accessResourcebyThread0(tlm_generic_payload * data, int prio)  {
84      Access *newAccess = new Access;
85      newAccess->data = data;
86      newAccess->priority = prio;
87      accessList.push_back(newAccess);
88      m_receivedTransaction_ev.notify();
89      sc_core::wait(m_thread0Granted_ev);
90    }
91
92    void accessResourcebyThread1(tlm_generic_payload * data, int prio)  {
93      Access *newAccess = new Access;
94      newAccess->data = data;
95      newAccess->priority = prio;
96      accessList.push_back(newAccess);
97      m_receivedTransaction_ev.notify();
98      sc_core::wait(m_thread1Granted_ev);
99    }
100
```

```
101   boost::container::list<Access *> accessList;
102   unsigned int m_noOfWordsPerApplicationPacket;
103   sc_core::sc_event m_receivedTransaction_ev;
104   sc_core::sc_event m_thread0Granted_ev;
105   sc_core::sc_event m_thread1Granted_ev;
106   sc_core::sc_event m_programmCycleFinished_ev;
107   tlm::tlm_generic_payload *p_payload0;
108   tlm::tlm_generic_payload *p_payload1;
109   unsigned long long buffer0;
110   unsigned long long buffer1;
111   unsigned long long dataSend0;
112   unsigned long long dataSend1;
113   unsigned long long dataRcv0;
114   unsigned long long dataRcv1;
115 };
```

**Listing B.1:** TLM-CA implementation of the simple example system.

## ATD based Implementation of Simple Example System

```
1  #define PROGRAMM_CYCLES 20000000
2  #define CLOCK_PERIOD sc_core::sc_time(50,sc_core::SC_NS)
3
4  class ATD : public ttlm::sc_ttlm_module {
5  public:
6    typedef unsigned long long BUFFER_t[1000];
7
8    SC_HAS_PROCESS(ATD);
9    ATD(sc_core::sc_module_name mn, unsigned int SRAQuantum, unsigned int
         noOfWordsPerApplicationPacket) : sc_ttlm_module(mn), m_exec(this), m_TDSem("TDSem",
         &m_exec)  {
10     SC_THREAD(thread0);
11     SC_THREAD(thread1);
12     m_TDSem.setClockPeriod(CLOCK_PERIOD);
13     m_noOfWordsPerApplicationPacket = noOfWordsPerApplicationPacket;
14     p_payloadPool = new tlm::tlm_generic_payload[2 * m_SRAQuantum];
15     p_bufferPool = new BUFFER_t[2 * m_SRAQuantum];
16     dataCounter0 = 0;
17     dataCounter1 = 0;
18   }
19
20   void thread0() {
21     for (int cycle = 0 ; cycle < PROGRAMM_CYCLES; cycle++) {
22       tlm::tlm_generic_payload *p_payload = &p_payloadPool[cycle % m_SRAQuantum];
23       unsigned long long *p_buffer = p_bufferPool[cycle % m_SRAQuantum];
24       for (unsigned int i = 0; i < m_noOfWordsPerApplicationPacket; i++) {
25         p_buffer[i] = dataCounter0++;
26       }
27       p_payload->set_data_ptr((unsigned char *)p_buffer);
28       p_payload->set_data_length(sizeof(unsigned long long) * m_noOfWordsPerApplicationPacket);
29       // register access starting 2 clock periods later and advance local time of thread0:
30       m_TDSem.registerAccess(2*CLOCK_PERIOD,0,p_payload);
31       if (m_noOfWordsPerApplicationPacket < 3) {
32         wait(2*CLOCK_PERIOD);
33       }
34     }
35   }
36
37   void thread1() {
```

```
38      for (int cycle = 0 ; cycle < PROGRAMM_CYCLES; cycle++) {
39        tlm::tlm_generic_payload *p_payload = &p_payloadPool[cycle % m_SRAQuantum + m_SRAQuantum];
40        unsigned long long *p_buffer = p_bufferPool[cycle % m_SRAQuantum + m_SRAQuantum];
41        for (unsigned int i = 0; i < m_noOfWordsPerApplicationPacket; i++) {
42          p_buffer[i] = dataCounter1++;
43        }
44        p_payload->set_data_ptr((unsigned char *)p_buffer);
45        p_payload->set_data_length(sizeof(unsigned long long) * m_noOfWordsPerApplicationPacket);
46        // register access starting 4 clock periods later and advance local time of thread1:
47        m_TDSem.registerAccess(4*CLOCK_PERIOD,1,p_payload);
48        if (m_noOfWordsPerApplicationPacket >= 3) {
49          wait(CLOCK_PERIOD*(m_noOfWordsPerApplicationPacket - 2));
50        }
51      }
52    }
53
54    template <class T_userData_t>
55    class PreemptiveExecutor : public ttlm::SRAExecutor_if<T_userData_t> {
56    public:
57      PreemptiveExecutor(ATD * parent) {
58        mp_parent = parent;
59      }
60
61      ttlm::executeSRAReturnValue_t executeSRA(ttlm::SRA<T_userData_t> *pSRA, sc_core::sc_time
            &duration, const sc_core::sc_time &timeBudget) {
62        tlm::tlm_generic_payload *p_payload = (tlm::tlm_generic_payload *)pSRA->getUserData();
63        duration = CLOCK_PERIOD * ceil(p_payload->get_data_length() / sizeof(unsigned long long));
64        unsigned int processedWords = ceil(min(timeBudget.value(),duration.value()) /
              CLOCK_PERIOD.value());
65        //mimic work...
66        if (pSRA->getPriority() == 0) {
67          memcpy(buffer0,p_payload->get_data_ptr(),processedWords * sizeof(unsigned long long));
68        } else {
69          memcpy(buffer1,p_payload->get_data_ptr(),processedWords * sizeof(unsigned long long));
70        }
71        if (duration > timeBudget) {
72          duration = timeBudget;
73          p_payload->set_data_length(p_payload->get_data_length() - processedWords *
                sizeof(unsigned long long));
74          p_payload->set_data_ptr((unsigned char *)((unsigned long long
                *)p_payload->get_data_ptr() + processedWords));
75          return ttlm::SRA_EXECUTION_PREEMPTED;
76        }
77        return ttlm::SRA_EXECUTION_OK;
78      }
79      BUFFER_t buffer0;
80      BUFFER_t buffer1;
81      ATD* mp_parent;
82    };
83
84    PreemptiveExecutor<tlm::tlm_generic_payload *> m_exec;
85    ttlm::TDSem<tlm::tlm_generic_payload *> m_TDSem;
86    unsigned int m_noOfWordsPerApplicationPacket;
87    unsigned int m_SRAQuantum;
88    unsigned long long dataCounter0;
89    unsigned long long dataCounter1;
90    tlm::tlm_generic_payload *p_payloadPool;
91    BUFFER_t *p_bufferPool;
92  };
```

**Listing B.2:** ATD based implementation of the simple example system.

## B.2. Remote Variable Map Listings

```
// register1 layout
typedef volatile struct {
  u32_t bitfield1     : bitfield1_width ;
  u32_t bitfield2     : bitfield2_width ;
  ...
} register1_st;

// union type allowing bit field and entire register accesses
typedef volatile union {
  u32_t                as_u32;  // access entire register
  register1_st         as_s;    // access via bit fields
} register1_ut;

// further register structs and unions here ...

// memory map layout
typedef volatile struct {
  register1_ut         register1_u;
  register2_ut         register2_u;
  ...
} componentName_MemMap_st;
```

**Listing B.3:** C type definition allowing the structured memory map access in embedded software or ISS-based simulations of driver software.

```
// register1 layout
class RemRegister1_ut {
private:
  register1_ut m_value;            // see see Listing B.3 for C register type declaration
  sc_dt::uint64 m_address;                // register address
  ttlm::MasterSocket<32> *mp_sock; // master socket to be used to access the component

protected:
  struct RemRegister1_ut_as_s {    // bitfield declarations: similar to TTLM remote variables
    UnitizedMember<RemRegister1_ut, u32_t> bitfield1;
    UnitizedMember<RemRegister1_ut, u32_t> bitfield2;
    ...
  };

public:
  UnitizedMember<RemRegister1_ut, u32_t> as_u32;
  RemRegister1_ut_as_s as_s;

protected:
  // as_u32
  inline void read_as_u32(u32_t& value) {
    mp_sock->read(m_address, &m_value.as_u32);
    value = m_value.as_u32;
  }
  inline void write_as_u32(const u32_t& value) {
    m_value.as_u32 = value;
    mp_sock->write(m_address, &m_value.as_u32);
  }

  // as_s.bitfield1
  inline void read_as_s_bitfield1(u32_t& value) {
    mp_sock->read(m_address, &m_value.as_u32);
    value = m_value.as_s.bitfield1;
```

```
34    }
35    inline void write_as_s_bitfield1(const u32_t& value) {
36      m_value.as_s.bitfield1 = value;
37      mp_sock->write(m_address, &m_value.as_u32);
38    }
39
40    // as_s.bitfield2
41    inline void read_as_s_bitfield2(u32_t& value) {
42      mp_sock->read(m_address, &m_value.as_u32);
43      value = m_value.as_s.bitfield2;
44    }
45    inline void write_as_s_bitfield2(const u32_t& value) {
46      m_value.as_s.bitfield2 = value;
47      mp_sock->write(m_address, &m_value.as_u32);
48    }
49
50    ...
51  public:
52    // constructor takes pointer to ttlm::MasterSocket and address of the register
53    inline RemRegister1_ut(ttlm::MasterSocket<32> *p_sock, sc_dt::uint64 address) {
54      as_u32.registerReadWrite(this, &RemRegister1_ut::read_as_u32,
            &RemRegister1_ut::write_as_u32);
55      as_s.bitfield1.registerReadWrite(this, &RemRegister1_ut::read_as_s_bitfield1,
            &RemRegister1_ut::write_as_s_bitfield1);
56      as_s.bitfield2.registerReadWrite(this, &RemRegister1_ut::read_as_s_bitfield2,
            &RemRegister1_ut::write_as_s_bitfield2);
57      ...
58      mp_sock = p_sock;
59      m_address = address;
60    }
61  };
62
63  // further register layouts here ...
64
65  // remote variable map layout
66  class componentName_RemoteVariableMap_st {
67  public:
68    // remote register class instances:
69    RemRegister1_ut register1_u;
70    RemRegister2_ut register2_u;
71    ...
72    // constructor takes pointer to ttlm::MasterSocket and base address of the component
73    componentName_RemoteVariableMap_st(ttlm::MasterSocket<32> *p_sock, sc_dt::uint64 baseAddress)
          :
74        register1_u(p_sock, baseAddress + register1_offset)
75        register2_u(p_sock, baseAddress + register2_offset)
76        ...
77    { }
78  };
```

**Listing B.4:** UnitizedMember class is based on the unitized approach [KBR09] and allows the registration of `read(..)` and `write(..)` functions (see lines 54 to 57) which are called in case the `UnitizedMember` object is accessed.]Remote variable map accompanying plain C type definition allowing structured memory map access in host-compiled execution of driver software in the context of virtual prototyping. The `UnitizedMember` class is based on the unitized approach [KBR09] and allows the registration of `read(..)` and `write(..)` functions (see lines 54 to 57) which are called in case the `UnitizedMember` object is accessed.

## B.3. TTLM Example

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <spirit:component xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
       http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd">
3    <spirit:vendor>Robert_Bosch_GmbH</spirit:vendor>
4    <spirit:library>demo</spirit:library>
5    <spirit:name>Timer</spirit:name>
6    <spirit:version>1.0</spirit:version>
7    <spirit:busInterfaces>
8      <spirit:busInterface>
9        <spirit:name>genericBus</spirit:name>
10       <spirit:busType spirit:version="1.0" spirit:library="demo" spirit:name="genericBus"
             spirit:vendor="Robert_Bosch_GmbH"/>
11       <spirit:abstractionType spirit:version="1.0" spirit:library="demo"
             spirit:name="genericBus_TLMAndClock" spirit:vendor="Robert_Bosch_GmbH"/>
12       <spirit:slave>
13         <spirit:memoryMapRef spirit:memoryMapRef="MM1"></spirit:memoryMapRef>
14       </spirit:slave>
15     </spirit:busInterface>
16   </spirit:busInterfaces>
17   <spirit:memoryMaps>
18     <spirit:memoryMap>
19       <spirit:name>MM1</spirit:name>
20       <spirit:addressBlock>
21         <spirit:name>AB1</spirit:name>
22         <spirit:baseAddress>0</spirit:baseAddress>
23         <spirit:range>4</spirit:range>
24         <spirit:width>32</spirit:width>
25         <spirit:register>
26           <spirit:name>currentValue</spirit:name>
27           <spirit:addressOffset>0</spirit:addressOffset>
28           <spirit:size>32</spirit:size>
29           <spirit:parameters>
30             <spirit:parameter>
31               <spirit:name>callback</spirit:name>
32               <spirit:value>read</spirit:value>
33             </spirit:parameter>
34           </spirit:parameters>
35         </spirit:register>
36       </spirit:addressBlock>
37     </spirit:memoryMap>
38   </spirit:memoryMaps>
39 </spirit:component>
```

**Listing B.5:** IP-XACT component description of the Timer module of the TTLM example.

```
1  #ifndef __TIMER_BASE_H
2  #define __TIMER_BASE_H
3
4  #include "systemc.h"
5  #include "tlm.h"
6  #include "scml2.h"
7  // for bit field callbacks
8  #include "scml2/bitfield_callback_functions.h"
9
10 #include "TTLMAdapter.h"
11
```

```
12 #include "logging.h"
13
14 #include "timer_memory_maps.h"
15 #include "timer_variants.h"
16
17 using namespace sc_core;
18 using namespace sc_dt;
19 using namespace std;
20
21 lst:class Timer_base: public ttlm::sc_ttlm_module {
22 public:
23   //
24   SC_HAS_PROCESS (Timer_base);
25
26   // constructor
27   Timer_base(sc_core::sc_module_name name);
28   virtual ~Timer_base();
29
30   // clock change handler
31   virtual void genericBus_clock_changeHandler();
32
33   //Communication Ports
34   ttlm::TargetSocket<32> genericBus_TLM_Bus;
35   sc_in<sc_core::sc_time> genericBus_clock;
36
37   /**
38    * Callback function
39    * Started on read access to Timer__MM1__AB1__currentValue (address 0).
40    */
41   virtual bool Timer__MM1__AB1__currentValue_readCB(unsigned int& value, const unsigned int&
       byteEnables, sc_core::sc_time& delay) = 0;
42
43 protected:
44   // SCML2 memory regions, register, and bitfields
45   scml2::memory<unsigned int> Timer__MM1_memory;
46   scml2::reg<unsigned int> *p_Timer__MM1__AB1__currentValue;
47
48   //SocketAdapter:
49   scml2::tlm2_gp_target_adapter<32> targetSocketAdaptergenericBus_TLM_Bus;
50 };
51
52 #endif //__TIMER_BASE_H
```

**Listing B.6:** "timer_base.h": header file of the timer base class. This file is completely generated by the TTLM generator and does not need to be edited by the designer.

```
1  #include "timer_base.h"
2
3  //clock change handlers:
4  void Timer_base::genericBus_clock_changeHandler() {
5    //update TargetSocket timePerWord:
6    genericBus_TLM_Bus.setClockPeriod(genericBus_clock.read());
7  }
8
9  Timer_base::~Timer_base() {
10   delete p_Timer__MM1__AB1__currentValue;
11 }
12
13 Timer_base::Timer_base(sc_core::sc_module_name name) :
14     sc_ttlm_module(name)
15     // Socket initialisation
```

```
16      , genericBus_TLM_Bus("genericBus_TLM_Bus")
17      , targetSocketAdaptergenericBus_TLM_Bus("TargetSocketAdaptergenericBus_TLM_Bus",
            genericBus_TLM_Bus)
18
19      // setup scml2 memory objects
20      , Timer__MM1_memory("Timer__MM1_memory", MEM__TIMER__MM1__SIZE >> 2)
21  {
22    // register clock change handler
23    SC_METHOD (genericBus_clock_changeHandler);
24    sensitive << genericBus_clock;
25
26    // connect targetSocketAdapters to corresponding memory structure:
27    targetSocketAdaptergenericBus_TLM_Bus (Timer__MM1_memory);
28
29    //setup SCML2 memory structure:
30    p_Timer__MM1__AB1__currentValue = new scml2::reg<unsigned
        int>("Timer__MM1__AB1__currentValue", Timer__MM1_memory,
        REG__TIMER__MM1__AB1__CURRENTVALUE__OFFSET >> 2);
31    scml2::set_disallow_write_access (*p_Timer__MM1__AB1__currentValue);
32    scml2::set_read_callback(*p_Timer__MM1__AB1__currentValue,
        SCML2_CALLBACK(Timer__MM1__AB1__currentValue_readCB), scml2::SELF_SYNCING);
33  }
```

**Listing B.7:** "timer_base.cpp": implementation file of the timer base class. This file is completely generated by the TTLM generator and does not need to be edited by the designer.

```
1   /*PROTECTED REGION ID(Timer_h) ENABLED START*/
2   #ifndef __TIMER_H
3   #define __TIMER_H
4   // include base class
5   #include "timer_base.h"
6
7   class Timer: public Timer_base {
8   public:
9     SC_HAS_PROCESS (Timer);
10
11    // constructor
12    Timer(sc_core::sc_module_name name);
13    // Empty virtual destructor, required for some compilers
14    virtual ~Timer(void) { }
15
16    /**
17     * Callback function.
18     * Started on read access to Timer__MM1__AB1__currentValue (address 0).
19     */
20    virtual bool Timer__MM1__AB1__currentValue_readCB(unsigned int& value, const unsigned int&
        byteEnables, sc_core::sc_time& delay);
21  };
22  #endif //__TIMER_H
23  /*PROTECTED REGION END*/
```

**Listing B.8:** "timer.h": header file of the timer user class. This file can be edited by the designer. Manual changes are preserved using a protected region (see Lines 1 and 23).

```
1   /*PROTECTED REGION ID(Timer_cpp) ENABLED START*/
2   #include "timer.h"
3   Timer::Timer(sc_core::sc_module_name name) : Timer_base(name) {
4   }
5
6   /**
```

```
7   * Callback function.
8   * started on read access to Timer__MM1__AB1__currentValue (address 0)
9   */
10
11  bool Timer::Timer__MM1__AB1__currentValue_readCB(unsigned int& value,
12      const unsigned int& byteEnables, sc_core::sc_time& delay) {
13    SC_LOG_DEBUG(name(), "Timer__MM1__AB1__currentValue_readCB called ");
14    //implement functionality which is to be executed before memory access takes place
15
16    *p_Timer__MM1__AB1__currentValue = (sc_time_stamp() + delay) / genericBus_clock.read();
17
18    //do memory access:
19    value = *p_Timer__MM1__AB1__currentValue;
20    //implement functionality which is to be executed after memory access took place
21
22    return true;
23  }
24  /*PROTECTED REGION END*/
```

**Listing B.9:** "timer.cpp": implementation file of the timer user class. The functionality implemented by the designer is printed in italics (see Line 16). The rest of the code is generated by the TTLM generator.

```
1   /*PROTECTED REGION ID(Testbench_cpp) ENABLED START*/
2   #include "testbench.h"
3   #include "timer_memory_maps.h"
4
5   // constructor
6   Testbench::Testbench(sc_core::sc_module_name name) :
7       Testbench_base(name) {
8     SC_THREAD (run);
9   }
10
11  void Testbench::run() {
12    //implement functionality here:
13    int currentValue_explicit;
14    ttlm::RemoteVariable<32, int>
        currentValue_implicit(&genericBus_TLM_Bus,REG__TIMER__MM1__AB1__CURRENTVALUE__OFFSET);
15    sc_core::sc_time clockPeriod = sc_core::sc_time(10,sc_core::SC_NS);
16    genericBus_clock.write(clockPeriod);
17
18    while (true) {
19      // explicit initiator-side communication API:
20      genericBus_TLM_Bus.read(REG__TIMER__MM1__AB1__CURRENTVALUE__OFFSET, &currentValue_explicit);
21      SC_LOG_DEBUG("Testbench", "currentTimer value = " << currentValue_explicit);
22      // implicit initiator-side communication API:
23      SC_LOG_DEBUG("Testbench", "currentTimer value = " << currentValue_implicit);
24      // non blocking wait provided by sc_ttlm_module class (TTLM timing API):
25      wait(10 * clockPeriod);
26    }
27    wait();
28  }
29  /*PROTECTED REGION END*/
```

**Listing B.10:** "testbench.cpp": implementation file of the testbench user class. The functionality implemented by the designer is printed in italics (see Lines 13 to 26). The rest of the code is generated by the TTLM generator.

# C. Simulation Host Characteristics

All performance measurements presented in this work have been conducted using the following host system:

**Hardware**

- HP Z800 workstation
- 2 x Intel Xeon E5620 quadcore (2,4 GHz per core, hyperthreading)
- 96 GB DDR3 RAM
- 2 x 300 GB SAS HDD (15000 rpm), HW-RAID level 0

**Software**

- OS: Linux 3.2.0-60-generic Ubuntu Server 12.04 LTS 64 Bit
- Compiler: GCC version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
- SystemC v2.3 kernel configuration command for quick thread usage:
  - 64 bit: `./configure --enable-shared=no --disable-async-updates`
  - 32 bit: `./configure --enable-shared=no --disable-async-updates --host i386-linux --build i386-linux --target i386-linux`

# List of Figures

# Bibliography


[Acc09] Accellera Systems Initiative. SystemRDL v1.0: A specification for a Register Description Language. , March 2009.

[Acc13] Accellera Systems Initiative. IP-XACT schema location. `http://www.accellera.org/XMLSchema/SPIRIT/1.5`, September 2013.

[AEG+10] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin. Increasing Design Productivity through Core Reuse, Meta-data Encapsulation, and Synthesis. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 538 –543, 2010.

[Agn13] Agnisys. IDesignSpec Product Website. `http://www.agnisys.com/idesignspec`, April 2013.

[ANMD07] Rabie Ben Atitallah, Smail Niar, Samy Meftali, and Jean-Luc Dekeyser. An MPSoC Performance Estimation Framework Using Transaction Level Modeling. In *An MPSoC Performance Estimation Framework Using Transaction Level Modeling*, September 2007.

[ARM13] ARM Ltd. ARM fast models Website. `http://www.arm.com/products/tools/models/fast-models/index.php`, May 2013.

[Ayn08a] John Aynsley. Getting Started with TLM-2.0 Tutorial 1 - Sockets, Generic Payload, Blocking Transport. `http://www.doulos.com/knowhow/systemc/tlm2/tutorial__1/`, June 2008. T147.

[Ayn08b] John Aynsley. Getting Started with TLM-2.0 Tutorial 3 - Routing Methods through Interconnect Components. `http://www.doulos.com/knowhow/systemc/tlm2/tutorial__3/`, June 2008. T148.

[Ayn13] John Aynsley. Doulos TLM-2.0 base protocol checker Website. `https://www.doulos.com/knowhow/systemc/tlm2/base_protocol_checker/`, January 2013.

[Ban09] Nico Bannow. *Modellierung und Optimierung eines Multiprozessorsteuergerätes mit SystemC*. PhD thesis, Tübingen, 2009.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560 – 599, June 1984.

[BM10] Brian Bailey and Grant Martin. IP Meta-Models for SoC Assembly and HW/SW Interfaces. In *ESL Models and their Application*, Embedded Systems, pages 33 – 82. Springer US, 2010.

[Bob07] Alex Bobrek. *A Statistical Approach to Contention Modeling for High-Level Heterogeneous Multiprocessor Simulation.* PhD Thesis, Carnegie Mellon University, 2007.

[BPT07] Alex Bobrek, JoAnn M. Paul, and Donald E. Thomas. Shared resource access attributes for high-level contention models. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, page 720–725, New York, NY, USA, 2007. ACM.

[BSS07] G. Beltrame, D. Sciuto, and C. Silvano. Multi-Accuracy Power and Performance Transaction-Level Modeling. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(10):1830 – 1842, 2007.

[Cad11] Cadence. Creating SystemC TLM-2.0 Peripheral Models - System Design and Verification - Cadence Community. `http://www.cadence.com/Community/blogs/sd/archive/2011/07/14/creating-systemc-tlm-2-0-peripheral-models.aspx`, July 2011.

[Cad13] Cadence Design Systems Inc. Cadence Website. `http://www.cadence.com/products/sd/virtual_system/pages/default.aspx`, April 2013.

[CAT09] CATRENE. European Design Automation Roadmap – 6th Edition. , 2009.

[CCZ06] B. Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In *Proceedings of the 6th international conference on Computational Science - Volume Part IV*, ICCS'06, page 653–660, Berlin, Heidelberg, 2006. Springer-Verlag.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67 – 78, June 1971.

[CG03] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 19 –24, October 2003.

[CKT03] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 190 – 195, 2003.

[Dai08] Daimler AG. Daimler 360 Grad - Fakten zur Nachhaltigkeit 2008. `http://nachhaltigkeit.daimler.com/daimler/annual/2013/nb/German/pdf/1557742_daimler_sust_2008_reports_nachhaltigkeitsbericht2008fakten_de.pdf`, 2008.

[DG90] N.D. Dutt and D.D. Gajski. Design synthesis and silicon compilation. *Design Test of Computers, IEEE*, 7(6):8–23, 1990.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. September 1965.

[EES+10] Wolfgang Ecker, Volkan Esen, Robert Schwencker, Thomas Steininger, and Michael Velten. TLM+ modeling of embedded HW/SW systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 75–80, April 2010.

[EF13a] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/`, September 2013.

[EF13b] The Eclipse Foundation. Eclipse Project Website. `http://www.eclipse.org/`, September 2013.

[Fis01] George S. Fishman. *Discrete-Event Simulation – Modeling, Programming, and Analysis*. Springer Series in Operations Research. Springer, 2001.

[Fre94] Freescale Semiconductor Inc. PowerPC™ 603 RISC Microprocessor Technical Summary. `http://www.freescale.com/files/32bit/doc/data_sheet/MPC603.pdf`, 1994.

[Fre05] Freescale Semiconductor. Application Note 2458 - MPC5200 Local Plus Bus Interface (Rev. 3). `http://www.freescale.com/files/32bit/doc/app_note/AN2458.pdf`, April 2005.

[Fre06] Freescale Semiconductor. MPC5200 Users Guide Rev 3.1. , March 2006.

[GAGS09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design, Modeling, Synthesis and Verification*. Springer, 1 edition, 2009.

[Ger10] Andreas Gerstlauer. Host-compiled simulation of multi-core platforms. In *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on*, pages 1 – 6, 2010.

[get14] get_clocktime Linux Manual Page. `http://man7.org/linux/man-pages/man2/clock_gettime.2.html`, June 2014.

[GHT12] Jens Gladigau, Christian Haubelt, and Jürgen Teich. Model-based Virtual Prototype Acceleration. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, 2012.

[GMPCM09] V. Galiano, H. Migallón, D. Pérez-Caparrós, and M. Martínez. Distributing SystemC structures in parallel simulations. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim '09, page 173:1–173:8, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[Gre12] GreenSocs Ltd. GreenSocket. `http://www.greensocs.com/projects/greenSocket`, 2012.

[Gro01] System-Level Design Development Working Group. VSI Alliance Model Taxonomy Version 2.1. , July 2001.

[HBHT08] Kai Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably distributed SystemC simulation for embedded applications. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pages 271 –274, June 2008.

[HBPG12] Simon Hufnagel, Nico Bannow, Hendrik Post, and Christoph Grimm. TTLM - Transparent Transaction Level Modeling. 25th European SystemC Users Group Workshop (ESCUG) 2012, March 2012.

[HPBG13] Simon Hufnagel, Hendrik Post, Nico Bannow, and Christoph Grimm. Improving Timing Accuracy for TLM-LT Models - The Advanced Temporal Decoupling (ATD) approach. 27th European SystemC Users Group Workshop (ESCUG) 2013, March 2013.

[IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology. , 1990.

[IEE06] IEEE 1364-2005 Standard Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[IEE09] IEEE 1076-2008 Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[IEE10] IEEE 1685-2009 Standard IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. , 2010.

[IEE11] IEEE 1666-2011 Standard SystemC 2.3 Language Reference Manual. *IEEE Std 1666™-2011 (Revision of IEEE Std 1666-2005)*, 2011.

[IEE12] IEEE 1800-2012 Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, 2012.

[Imp14] Imperas Software Limited. Open Virtual Platforms. `http://www.ovpworld.org/welcome`, May 2014.

[ITR11a] ITRS Technology Working Group. Design. In *International Technology Roadmap For Semiconductors (ITRS)*. 2011 edition, 2011.

[ITR11b] ITRS Technology Working Group. Executive Summary. In *International Technology Roadmap For Semiconductors (ITRS)*. 2011 edition, 2011.

[Jon11] Samuel Jones. Optimistic Parallelisation of SystemC. Technical report, 2011.

[JS05] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):114 – 129, March 2005.

[KBR09] Christian Kerstan, Nico Bannow, and Wolfgang Rosenstiel. Efficient Architecture Evaluation Using Functional Mapping. In *Languages for Embedded Systems and their Applications - Selected Contributions on Specification, Design, and Verification from FDL '08*, pages 167–182. Springer Science, 2009.

[Ker09] Christian Kerstan. *Effiziente Bewertung von Videoarchitekturen mit SystemC*. PhD thesis, Tuebingen, 2009.

[Kir11] Tobias Kirchner. *A Framework for Refinement of Mixed Signal Systems*. PhD thesis, Technical University Vienna, 2011.

[Kro03] Klaus Kronlöf. SYDIC - Telecom Glossary. In *System Level Design Model with Reuse of System IP*. Springer US, 1 edition, 2003.

[KvdWdK+08] Wido Kruijtzer, Pieter van der Wolf, Erwin de Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge de Paoli, and Emmanuel Vaumorin. Industrial IP Integration flows based on IP-XACT standards. In *Proceedings of the conference on Design, automation and test in Europe*, pages 32–37, April 2008.

[LBC+10] V. Lefftz, J. Bertrand, H. Casse, C. Clienti, P. Coussy, L. Maillet-Contoz, P. Mercier, P. Moreau, L. Pierre, and E. Vaumorin. A design flow for critical embedded systems. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 229 –233, July 2010.

[LCH08] Kuen-Huei Lin, Siao-Jie Cai, and Chung-Yang Huang. Speeding up SoC virtual platform simulation by data-dependency aware virtual synchronization. In *SoC Design Conference, 2008. ISOCC '08. International*, volume 01, pages I–256 –I–259, November 2008.

[LMGS12] Kun Lu, D. Muller-Gritschneder, and U. Schlichtmann. Accurately timed transaction level models for virtual prototyping at high abstraction level. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 135 –140, March 2012.

[LRD01] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(6):768 – 783, 2001.

[LTdS11] J. Le Tallec and R. de Simone. SCIPX: A systemc to IP-Xact extraction tool. In *Electronic System Level Synthesis Conference (ESLsyn), 2011*, pages 1 –6, June 2011.

[Man09] Peter Mandl. Konzepte und Modelle verteilter Kommunikation. In *Masterkurs Verteilte betriebliche Informationssysteme*, pages 27–171. Vieweg+Teubner, 2009.

[Mar10] John Markoff. Google Cars Drive Themselves, in Traffic. *New York Times*, October 2010.

[MCG05] Laurent Maillet-Contoz and Frank Ghenassia. Transaction Level Modeling - An Abstraction beyond RTL. In *Transaction Level Modeling with SystemC - TLM Concepts and Applications for Embedded Systems*, pages 23 – 55. Springer US, 2005.

[Men13] Mentor Graphics. Mentor Graphics Website. `http://www.mentor.com/esl/vista/virtual-prototyping/`, April 2013.

[met13] Metriculator Eclipse CDT Plugin Website. `http://marketplace.eclipse.org/content/metriculator`, December 2013.

[Mey05] Trevor Meyerowitz. Transaction-Level Modeling Definitions and Approximations. Technical Report EE290A, May 2005.

[MMGP10] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel Simulation of SystemC TLM 2.0 compliant MPSoC on SMP Workstations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 606 –609, March 2010.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[NPJS10] M. Nanjundappa, H.D. Patel, B.A. Jose, and S.K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 149 – 154, January 2010.

[NVI13] NVIDIA Corporation. NVIDIA CUDA Website. `http://www.nvidia.com/object/cuda_home_new.html`, 2013.

[Ope13] OpenArchitectureWare. Xpand / Xtend / Check Reference. `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html`, September 2013.

[PDT13] PDTi. SpectaReg Product Website. `http://www.productive-eda.com/register-management/`, April 2013.

[PMG+10] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty, and A. Herkersdorf. High-level timing analysis of concurrent applications on MPSoC platforms using memory-aware trace-driven simulations. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 229 –234, September 2010.

[PVSL11] J. Peeters, N. Ventroux, T. Sassolas, and L. Lacassagne. A systemc TLM framework for distributed simulation of complex systems with unpredictable communication. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1 –8, November 2011.

[PWB11] Thomas P. Perry, Richard L. Walke, and Khaled Benkrid. An Extensible Code Generation Framework for Heterogeneous Architectures based on IP-XACT. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 81–86, May 2011.

[RG12] Parisa Razaghi and Andreas Gerstlauer. Automatic timing granularity adjustment for host-compiled software simulation. In *ASP-DAC*, pages 567 – 572. IEEE, 2012.

[RK08] M. Radetzki and R.S. Khaligh. Accuracy-Adaptive Simulation of Transaction Level Models. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 788 – 791, 2008.

[Rob05] Robert Bosch GmbH. Night Vision looks far ahead. `http://www.bosch-presse.de/presseforum/details.htm?txtID=2513&locale=en`, November 2005.

[RRO+12] Stephan Radke, Steffen Rülke, Marcio Oliviera, Christoph Kuznik, Wolfgang Müller, Wolfgang Ecker, Volkan Esen, Simon Hufnagel, Nico Bannow, Helmut Brazdrum, Peter Janssen, Hoang Le, Daniel Große, Rolf Drechsler, Erhard Fehlauer, Gernot Koch, Andreas Burger, Oliver Bringmann, Wolfgang Rosenstiel, Finn Haedicke, Ralph Görgen, and Jan-Hendrik Oetjens. Compilation of Methodologies to Speed up the Verification Process at System Level. In *Proceedings of edaWorkshop 2012*, pages 57 – 62, May 2012.

[SBR11] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate resource conflict simulation for performance analysis of multi-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –6, March 2011.

[SD07] G. Schirner and R. Dömer. Result-Oriented Modeling – A Novel Technique for Fast and Accurate TLM. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(9):1688 –1699, September 2007.

[SD09] Gunar Schirner and Rainer Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *ACM Trans. Embed. Comput. Syst.*, 8(1):1–29, January 2009.

[SHRE09] Simon Schliecker, Arne Hamann, Razvan Racu, and Rolf Ernst. Formal Methods for System Level Performance Analysis and Optimization. Technical report, Technical University Braunschweig, June 2009.

[SKR09] Rauf Salimi Khaligh and Martin Radetzki. Efficient Parallel Transaction Level Simulation by Exploiting Temporal Decoupling. In Achim Rettberg, Mauro Zanella, Michael Amann, Michael Keckeisen, and Franz Rammig, editors, *Analysis, Architectures and Modelling of Embedded Systems*, volume 310 of *IFIP Advances in Information and Communication Technology*, pages 149–158. Springer Boston, 2009. 10.1007/978-3-642-04284-3_14.

[SLPH10] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 241–246, October 2010.

[Syn11] Synopsys Inc. SystemC Modeling Library 2. , December 2011.

[Syn13] Synopsys Inc. Synopsys Website. `http://www.synopsys.com/systems/virtualprototyping/Pages/default.aspx`, April 2013.

[Tan09] Andrew Tanenbaum. *Modern Operating Systems*. third edition, 2009.

[TLM14] TLM Central. `http://www.dr-embedded.com/tlmcentral/`, 2014.

[TUV13] SystemC-AMS Building Block Library Website. `http://www.systemc-ams.org/BB_library.html`, May 2013.

[Ulr68] Ernst G. Ulrich. Serial/parallel event scheduling for the simulation of large systems. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, pages 279 – 287, New York, NY, USA, 1968. ACM.

[VCBF12] Sara Vinco, Debapriya Chatterjee, Valeria Bertacco, and Franco Fummi. SAGA: SystemC Acceleration on GPU Architectures. In *DAC '12 Proceedings of the 49th Annual Design Automation Conference*, pages 115 – 120, June 2012.

[vHdKV11] Gino van Hauwermeiren, Erwin de Kock, and Jos Verhaegh. IP-XACT-based register abstraction and software test generation. DATE 2011 Special Interest Workshop 7, March 2011.

[XDM10] Tao Xie, Gilles B. Defo, and Wolfgang Mueller. An Eclipse-based Framework for the IP-XACT enabled Assembly of Mixed-Level IPs. In *First Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems (HoPES 2010)*, June 2010.

[ZBBR09] J. Zimmermann, O. Bringmann, A. Braun, and W. Rosenstiel. Integration of high-level synthesis in ESL platform modeling by automated generation of protocol adapters. In *Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on*, pages 1149 – 1154, 2009.

[ZM08] Henning Zabel and Wolfgang Müller. Präzises Interrupt Scheduling in abstrakten RTOS Modellen in SystemC. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 31 – 39, MBMV08, Freiburg, Germany, 2008. Shaker Verlag.

[ZMG09] Henning Zabel, Wolfgang Müller, and Andreas Gerstlauer. Accurate RTOS Modeling and Analysis with SystemC. In *Hardware-dependent Software - Principles and Practice*, pages 233 – 260. Springer, 2009.

# Glossary

**checkpoint** A point in a computer program at which program state, status, or results are checked or recorded [IEE90]

**delta cycle** simulation mechanism used in DES to simulate interdependent coincident simulation events

**delta event** event that is to occur at the same time as the notification happens. Delta events are processed in the following delta cycle

**design flow** Decomposition of the design process into a number of phases, e.g. requirements definition, specification, architecture design, mapping, software design, hardware design, and integration [Kro03]

**formal specification** a specification written in a formal notation, often for use in proof of correctness [IEE90].

**global quantum** maximum amount of simulation time after which a synchronization is forced when using quantum keeper

**host-compiled execution** Technique to execute software upon a virtual prototype. In contrast to execute the software using an ISS, the source code of the software is integrated into the virtual prototype and compiled for the host architecture.

**inbound data dependency** data dependency towards the currently executed thread inhibiting its further execution as data provided by an external resource is required.

**initiator** unit initiating a transaction in transaction level modeling

**IP-XACT** standard structure for packaging, integrating, and reusing IP within tool flows [IEE10]

**local time warp** timing effect where the local simulation time of a simulation thread is greater than the global simulation time. This might occur in temporal decoupled transaction level simulations [IEE11]

**non-atomic transaction** transaction carrying data spanning multiple atomic time units

**quantum keeper** mechanism provided by SystemC standard to limit the offset between global simulation time and a thread's local simulation time to reduce the effect of simulation errors occurring in temporal decoupled simulations [IEE11]

**resume event** Event exclusively associated to one thread which is used to implement synchronization in ATD

**rollback** restore the state of a program to a state previously recorded at a checkpoint

**SRA matrix** data structure used for resource contention detection and handling. Each row of this matrix is associated to exactly one TDSem and each column to one thread. Each cell contains an ordered set of SRAs registered at the corresponding TDSem by the associated thread denoted as $\mathbf{SRA}_{Th_i,TDSem_j}$.

**SRA tree** rooted tree data structure used for maintaining inter SRA dependencies. There is one SRA tree per thread and the root element of each SRA tree corresponds to one thread.

**synchronization point** class of thread activities comprising data dependencies. To avoid data dependency violations, temporal decoupling has to stop at synchronization points

- **influencing synchronization point** A synchronization point is denoted as an influencing synchronization point after the resume event of the corresponding thread has already been notified and therefore influences the value returned by `sc_time_to_pending_activity()`. The notification of the resume event takes place during the transaction processing phase after all pending SRAs of that thread have been processed.
- **non-influencing synchronization point** A synchronization point is denoted as a non-influencing synchronization point before the resume event of the corresponding thread is notified and therefore does not influence the value returned by `sc_time_to_pending_activity()`.

**target** unit receiving and processing a transaction in transaction level modeling

**temporal decoupling** simulation technique that allows simulation threads to run ahead in simulation time. Temporal decoupling can be use in SystemC TLM-2.0 loosely-timed modeling style [IEE11]

**thread** A single path of execution through a program, a dynamic model, or some other representation of control flow [Kro03]

**time budget** amount of simulation time available for the processing of each transaction in Advanced Temporal Decoupling

**TLM** Transaction Level Modeling [IEE11]

- **approximately-timed** SystemC TLM-2.0 modeling style which allows finer grained transaction timing modeling by introducing transaction phases [IEE11]
- **loosely-timed** SystemC TLM-2.0 modeling style which allows temporal decoupling. Each transaction is modeled using a single function call [IEE11]

**trace** A record of the execution of a computer program, showing the sequence of instructions executed [..] [IEE90]

**transaction** logically delimited interaction occurrences between a sending unit called initiator and a receiving unit denoted as target

**virtual prototype** computer-simulation model of a final product, component, or system [Gro01]

**word** A sequence of bits or characters that is stored, addressed, transmitted, and operated on as a unit within a given computer [IEE90].

# Acronyms

**API** Application Programming Interface
**ATD** Advanced Temporal Decoupling

**CPU** Central Processing Unit
**CUDA** Compute Unified Device Architecture

**DES** Discrete Event Simulation
**DMA** Direct Memory Access
**DSL** Domain Specific Language

**ECU** Electronic Control Unit
**EMF** Eclipse Modeling Framework

**FPGA** field programmable gate array
**FQN** fully qualified name

**GPGPU** General Purpose Graphic Processing Unit
**GPU** Graphic Processing Unit

**IP** Intellectual Property
**ISS** Instruction Set Simulator

**MEI** modified-exclusive-invalid
**MPSoC** Multi Processor System-on-Chip
**MWE** Modeling Workflow Engine

**NIR** near-infrared

**OS** operating system

**PCI** peripheral component interconnect

**RTL** register transfer level
**RTOS** Real Rime Operating System

**SCML2** SystemC Modeling Library 2 [Syn11]
**SLoC** Source Lines of Code
**SRA** Shared Resource Access

**TDSem** Temporal Decoupled Semaphore
**TLM-CA** TLM - Cycle Accurate [MCG05]

**TLM-PV** TLM - Programmer's View [MCG05]
**TLM-PVT** TLM - Programmer's View with Timing [MCG05]
**TTLM** Transparent Transaction Level Modeling

**VLNV** Vendor, Library, Name, and Version [IEE10]
**VP** Virtual Prototype

**XLB** 60X Local Bus [Fre05]

# Curriculum Vitae

**Simon Hufnagel**

| | |
|---|---|
| **Since 08/2013** | **Research Engineer**<br>Functional Safety for Powertrain Applications<br>Robert Bosch GmbH, Schwieberdingen<br>(Corporate Sector Research and Advance Engineering) |
| **08/2010 – 07/2013** | **Doctoral Program**<br>Robert Bosch GmbH, Schwieberdingen<br>(Corporate Sector Research and Advance Engineering),<br>Technical University of Kaiserslautern<br>(Department of Computer Science),<br>Technical University of Vienna<br>(Faculty of Electrical Engineering and Information Technology) |
| **10/2006 – 07/2010** | **Studies of Electrical Engineering and Information Technology**<br>Degree: Diplom-Ingenieur (FH)<br>Georg Simon Ohm University of Applied Sciences Nuremberg<br>(Faculty of Electrical Engineering and Information Technology) |
| **09/2005 – 02/2009** | **Vocational Education and Training**<br>Electronics Technician for Automation Technology<br>Robert Bosch GmbH, Plant Ansbach-Brodswinden<br>(Automotive Electronics) |
| **09/1996 – 05/2005** | **Academic High School**<br>Feuchtwangen, Bavaria |

Ludwigsburg, August 23, 2014