DISSERTATION

Crossing Modeling Paradigms in System Models

Submitted at the Faculty of Electrical Engineering and Information Technology, Vienna University of Technology in partial fulfillment of the requirements for the degree of Doktor der technischen Wissenschaften (equals Ph.D.)

under supervision of

Univ. Prof. Dr. habil. Christoph Grimm
Institute number: 384
Institute of Computer Technology

and

Prof. Dr.-Ing. Martin Radetzki
Institute of Computer Architecture and Computer Engineering
University of Stuttgart

by

Markus Damm
Matr.Nr. 0628003

February 2nd 2015

Kurzfassung

Bei der Modellierung und Simulation von eingebetteten und "cyber-physical" Systemen ist es oft vorteilhaft, wenn nicht gar notwendig, unterschiedliche Berechnungsmodelle für die verschiedenen Teile des Systems zu verwenden. Sei es weil diese Subsysteme zu verschiedenen Domänen gehören (wie z.B. Analog oder Digital), die nach verschiedenen Methoden zur Modellierung verlangen, oder weil verschiedene Abstraktionsebenen im gleichen Modell verwendet werden.
Was auch immer die Motivation dafür ist, stets werden wohldefinierte Methoden benötigt um diese unterschiedlich modellierten Systemteile miteinander zu verbinden. In dieser Arbeit werden solchen Konvertierungsmethoden auf der Basis einer formalen Sichtweise auf Prozesssteuerung diskutiert, entwickelt, und in einer SystemC-basierten Simulationsumgebung implementiert.

Abstract

In modeling and simulation of embedded and cyber-physical systems, it is often favorable, if not necessary, to use different Models of Computation for different parts of the system. These system parts might belong to different domains (e.g. analog or digital) which call for different means of modeling, or possibly different abstraction levels within the same model are used.
Whatever the motivation, well-defined methods to connect these differently modeled systems parts are needed. In this thesis, such conversion means are discussed and developed using a formal framework for process control, and are implemented in a SystemC-based simulation environment.

Acknowledgements

I want to thank my supervisor Prof. Dr. Christoph Grimm for giving me this opportunity and his support, and my external advisor Prof. Dr. Martin Radetzki. I also want to express my gratitude to all the contributors of the ANDRES project, especially Dr. Fernando Herrera and Prof. Dr. Axel Jantsch for the helpful debating of conversion semantics, Dipl.-Inform. Philipp A. Hartmann for discussing C++ coding, and Dipl.-Ing. Joseph Wenninger for his support with the Converterchannel data type conversion.
Finally I want to thank all my former colleagues at the Institute of Computer Technology, especially Dr. Jan Haase and Dr. Fritz Bauer. It was a great time.

Table of Contents

1 Introduction

For several years now, the electronic design community follows an agenda to automatize the design process as much as possible, driven by Moore's Law [Sch97]. Elaborate tools (or tool chains) can transform an abstract model of an electronic system, for example coded in a hardware description language like VHDL [LSU89, LG88], into a real chip; generally via several transformation steps. Some target architectures, like FPGA, offer complete automated design flows already today. For others, manual transformation or manual intrusion in semi-automated transformation processes is still necessary. In any case, the means to describe a system will in general be different after each transformation step: A different tool, a different language, a different abstraction level. Consequently, the modeling paradigms for these descriptions will differ as well.

Embedded systems [Zav82, GVNG94] are electronic systems which are specifically designed to fulfill dedicated functions within certain devices or machines. They control, for example, the motor of a car, or white goods like washing machines. They are typically comprised of a microprocessor together with analog sensors and actors, i.e. specialized analog electronics. To model such a system, at least two modeling paradigms are needed for the analog and the digital domain, respectively; but often it might be even more.

Also, there is the need to capture the tight interaction of embedded systems with their analog physical environment already early in the design process. One motivation is that most embedded systems form some kind of feedback loop with their environment. Consider for example an embedded system implementing the controller in a control system by means of software. An important part of a model of such a system is the control algorithm. If the model is used to test the control algorithm by means of simulation, we obviously need to provide an input stream. But since the output of the control algorithm influences the subsequent input to it, we get this input stream only by also capturing the physical process (commonly called the *plant*) which is to be controlled. This motivated approaches to connect simulation models to real physical plants, like hardware-in-the-loop [ISS99, Bac05] and software- or model-in-the-loop [YWL02, Plu06]. However, modeling and simulating the physical processes involved (if possible) is less costly, and definitely preferable in the early stages of the design process.

This is an important difference between embedded software and software in desktop applications like spreadsheet software or compilers. The latter can be evaluated using benchmark inputs, e.g. sets to be sorted or C++ code to be mapped to machine code, with evaluation criteria like runtime or size of the output. With embedded software, such a simple input - output analysis is usually not sufficient. This is even more true for *cyber-physical systems* [Lee08, RLSS10], i.e.

systems which are networked and where the evaluation criteria might concern behavior which emerges from the network interaction, e.g. when considering wireless sensor/actor networks.

The consequence of these developments is that system models tend to become more and more heterogeneous. The modeling paradigm for a certain (sub-)system depends on its domain and the abstraction level, but also on the specific goals behind the model. Figure 1.1 depicts this situation (With no claim of completeness). Note however that the three axes are not completely independent. The goals behind a model often heavily influence its abstraction level, e.g. fast simulation often implies high abstraction. Also, on higher abstraction levels the borders between the domains often blur, since a certain functionality can often be implemented in several domains, e.g. as an application specific integrated circuit (ASIC) or in software.

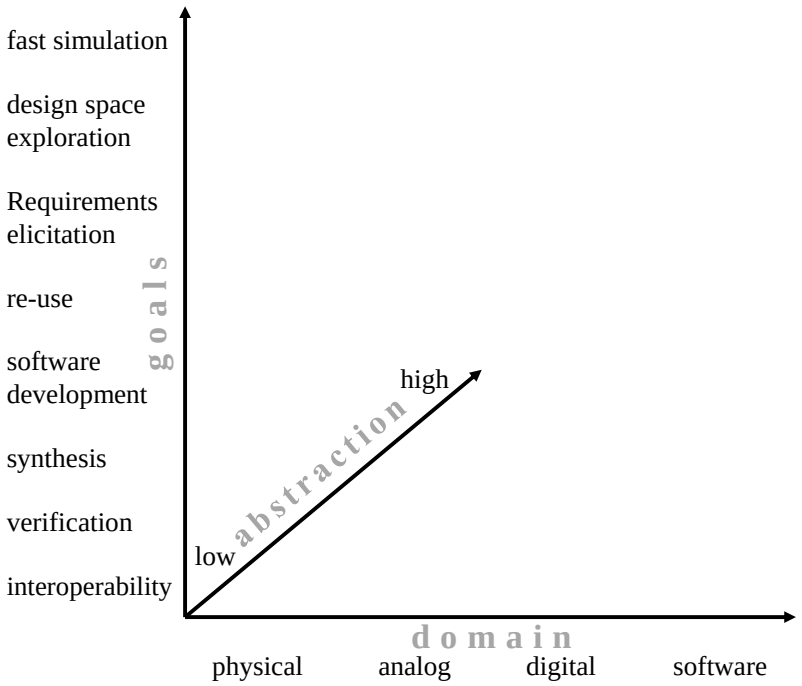


Figure 1.1: criteria for choosing modeling paradigms

Some examples of system models are

- A list of requirements the system has to fulfill
- A block diagram
- Mathematical descriptions like equation systems or transfer functions
- Algorithms for certain functions implemented by the system, e.g. using C code or any other programming language
- Models using a modeling language like VHDL, Verilog or SystemC

In this thesis, we are only interested in models that can be executed, i.e. simulated. The means used to describe a system and the semantics to execute this description make up a *Model of Computation*. For now, we will give an informal definition of a Model of Computation before giving a formal one in Chapter 3:

Definition 1.1 (Informal MoC Definition)**.** *A **Model of Computation** (MoC[1]) is a specification of computational and communicational means for the description of system models.*

The modeling goals, the abstraction level and the domain of a system model determine the Model of Computation to use; maybe directly but mostly indirectly by the choice of the modeling language or modeling tool. In fact, users will often be oblivious about the underlying MoC of the simulation tool they use. With heterogeneous system models the MoCs used for certain sub-systems will in general be different; they might even vary over time with the change of modeling goals in the course of the system development. Handling the interaction of sub-systems modeled with different MoCs is the main topic of this thesis.

The *general* goal behind a system model is mostly to draw conclusions about the (correct) operation of the implemented system in reality. The ideal case would be a system model representing an exhaustive virtual image of the system (and its environment) such that it can be tested on a computer before producing (costly) hardware prototypes. At the same time, *computational feasibility* is an issue. The higher the detail of the model the more computational resources it needs. Therefore it is important to find a suitable level of abstraction: It should be low enough to capture the phenomena of interest (like functional correctness, timing or power consumption), but high enough to keep the computational complexity at bay. Here it can also make sense to mix more than one abstraction level within one system model.

As an example how abstraction levels can vary for the same system, consider a digital circuit which at the lowest abstraction level can be described as a network of transistors. This network can be simulated, for example by using SPICE [NP73], but this can get very costly when the digital circuit is large. The next highest abstraction level is the gate level, where the digital circuit is described by means of elementary logic blocks and registers. For purposes like simulation, the logic blocks can be annotated with meta-information like delay times or power consumption. This meta-information can be obtained by modeling each type of logic block (like a NAND-gate) at the transistor level and analyzing it (e.g. timing analysis) on this lower abstraction level.

A next possible (higher) abstraction level would be the Register Transfer Level (RTL). Here the system is described by means of registers and larger functional units (like an ALU), whose functionality can be simply described by algorithms. Again, information like timing can be included based on models of sub-components on lower abstraction levels. Above that is the CPU itself, which in principle can also be modeled in a structural manner like RTL, especially when designing the CPU itself. For special modeling goals like determining the worst case execution time of software [FHL+01], it is necessary to have a CPU model with key structural elements like the pipeline and the cache hierarchy. However, for many applications it is sufficient to use something like an Instruction Set Simulator (ISS)[LLSV98, ZG99], especially if the modeling goal is to put the CPU in the context of a larger system.

This abstraction level, which finally encompasses the system as a whole, is commonly called the *System Level*, or more specifically, the electronic system level (ESL) [GHP+09, MBP10]. Here, techniques like Transaction Level Modeling [CG03] became more and more important in recent years. The goal on the system level these day is often to provide an early system model where software can be tested (even as a cross-compiled binary) before any real hardware prototypes of the system are available. Therefore, these models are called *virtual prototypes*.

Figure 1.2 shows a chain of these abstraction levels, together with the tools/languages (on top in blue) and the general means (below in green) used on each level. Note that Figure 1.2 is

[1] In this thesis, the term *MoC* is used for the singular case, and the term *MoCs* for the plural case

not intended to be complete, since there are many tools and techniques available and the borders between abstraction levels are blurry sometimes. In fact, as Figure 1.2 also indicates, it is possible to use a lower level model of a sub-system in a higher level model of the overall system, for example if for this sub-system the higher precision of the lower abstraction level is needed, or the modeling goal is to analyze the sub-system in the context of a larger system for which the higher abstraction is sufficient.

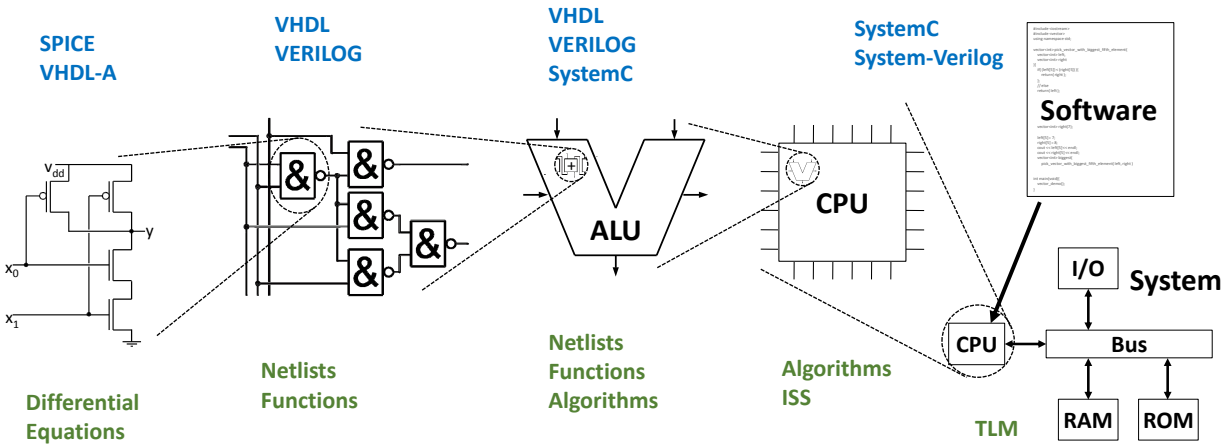


Figure 1.2: A chain of abstraction levels

Not shown in Figure 1.2[2] is another key element, especially for embedded systems: the environment. It is represented indirectly by the I/O block on the system level to the right, since this is the interface for the *direct* interaction of the system with the environment. To be able to test software of an embedded system which interacts with the environment via the I/O (e.g. by reading sensor input and providing control signals for actors), it might be necessary to also provide an environment model. In the case of control loops mentioned above, the environment model would be a model of the plant. User models to simulate user interaction are other examples of possible environment models. Also, there might be *indirect* environmental factors (e.g. temperature or humidity) influencing the system behavior.

The MoCs between abstraction levels don't *need* to change. The RTL and gate level can be modeled very similar (e.g. as discrete event models), the main difference here is the granularity of the functional blocks. From gate level to the transistor level, however, there is a fundamental step from non-conservative to conservative systems. Transistor level models (and analog circuit models in general) give rise to equation systems emerging from conservation theorems like Kirchhoff's circuit laws, and any simulator for such models needs to solve these equations.

MoCs arise also in other areas like computability theory, and also in the modeling and analysis of systems other than electronic ones. But in this thesis, the subject will be treated with respect to electronic system design, concentrating on a special issue: Connecting subsystems modeled using different MoCs.

Let's assume we have two models A and B modeled with two different MoCs, and we want to connect them. There are basically three possibilities:

[2] because it is not an abstraction level or part of the system

1. Find a conversion semantic for the connection, i.e. adapt the way information is exchanged in MoC A such that it can be interpreted in a meaningful way in MoC B (and/or vice versa, depending on the direction of the information flow).

2. Convert A into the MoC of B (or vice versa).

3. Convert both models into a third MoC

It is obvious that only the first option is viable. Transforming the model of a whole subsystem is in general too time consuming and error-prone. Also, the MoC might not be suitable, e.g. when transforming an RTL level model of a digital circuit into a transistor-level model (which can be done automatically since this is a standard synthesis step) in order to simulate it together with an analog circuit. The resulting simulation overhead would mostly be not acceptable. Also, nothing really relevant can be learned about a digital circuit when simulating it as an analog circuit.

The core of this work was done in the course of the ANDRES project[3] [HOH+07], which was an EC-funded project (FP6, IST-5-033511) running from 2006 to 2009. The goal of this project was to provide a (mainly SystemC-based) design flow for adaptive heterogeneous systems, and since the heterogeneity implied the usage of different MoCs there was the need for appropriate conversion means. As an additional requirement, the conversion should be *automatic*; i.e. if two subsystem-models are connected by a converter, and one subsystem-model is replaced by another one using a different MoC, the converter has to adapt automatically such that there is no need to also replace the converter. This concept was called *Converterchannel* and builds on the idea of *polymorphic signals* [Sch07]. In fact, as a black box a Converterchannel behaves very similar to a polymorphic signal, but the conversion approaches are very different.

The main publications which stemmed from this work are

- "Using converter channels within a top-down design flow in SystemC" (conference paper, Austrochip 2007 [DHHV07]) presents the general concept of the Converterchannels.
- "Bridging MoCs in SystemC specifications of heterogeneous systems" (journal paper, EURASIP Journal for Embedded Systems [DHG+08b])contains the approach of connecting process networks to timed data flow clusters in SystemC
- "Connecting SystemC-AMS models with OSCI TLM 2.0 models using temporal decoupling" (conference paper, FDL 2008 [DGH+08]) presents the first approach for converters to connect SystemC TLM2 models to TDF models.

The main contributions of this thesis besides the implementation of the Converterchannel concept are:

- *Conversion semantics for certain MoC conversion cases*: While the Converterchannels also use already existing conversion means like the converter-ports provided by SystemC AMS to connect discrete event signals to timed data flow modules, also new conversion semantics where developed and integrated into the Converterchannels in case they where missing; most notably the timed data flow $\leftrightarrow$ process network conversion semantics. This includes also the TLM$\leftrightarrow$TDF conversion semantics which are not part of the Converterchannels.

[3] ANalysis and Design of run-time REconfigurable, heterogeneous Systems

- *Conversion corner cases*: For most conversion means (even those already available) there are certain corner cases whose handling depends largely on the modeling goals and therefore can't be treated automatically per se. These cases are identified and discussed to derive appropriate conversion semantics.
- *MoC formalism*: A formalism for MoCs which is still abstract yet close enough to frameworks like SystemC such that the conversion semantics derived can be easily implemented.
- An approach to describe transaction level modeling formally as a MoC.

The rest of this thesis is organized as follows: In Chapter 2 we discuss existing work on MoCs as well as frameworks for system modeling (most notably SystemC and its extensions) and the MoCs they use. In Chapter 3, a formalism for the description of MoCs is defined and used to specify the MoCs relevant to this thesis. Chapter 4 describes several MoC conversion approaches for different pairs of MoCs using the formalism of the previous chapter. In Chapter 5 the SystemC implementation of the Converterchannels is described, and Chapter 6 covers the implementation of the converters between transaction level models and timed data flow models; as these converters are not part of the converter channels, and also provide an application example. After that, we conclude.

2 Models of Computation in System Design

As indicated in the introduction, we treat the subject of MoCs (and the conversion between them) in the context of modeling and simulation of physical systems, with a focus on electronic system design. But in general, a MoC is just an abstract model of a computational system. One of the simplest MoCs is the well-known Finite State Machine (FSM). This MoC is restricted to a finite set of states S and a finite set of state-transitions T successively triggered by input symbols out of a finite alphabet Σ; possibly producing an output using a finite output alphabet in the process. Finite State Machines are the backbones of many MoCs. For example, by giving an FSM access to an unbounded stack (by using it as additional input and output), we derive the more elaborate pushdown automaton (PDA); and by using an unbounded tape we get the Turing Machine.

In theoretical computer science, the study of the languages accepted by these computational systems when considering certain acceptance conditions (e.g. reaching an acceptance state) gave rise to a rich theory: formal languages [Har78], formal grammars [GL70], and classifications like the Chomsky hierarchy [Cho56], to name just a few. In computer engineering, these abstract models are the blue prints for the implementation of computational systems. States and transitions manifest in registers and clock-cycles, stacks and tapes in various (possibly big but yet finite) memory forms like RAMs and hard-drives.

In system modeling, MoCs are used to describe how to *model* and eventually *analyze* systems; the latter often by means of *simulation*. We model existing systems which we want to study, like simulation models for weather forecasts or economic predictions, or systems which we intend to build, like vehicles or electronic devices. The analysis means provided by a MoC often include semantics to simulate system models on some computational system.

In electronic system design, modeling general concurrent systems and simulating them is part of the design process. In fact, with a hardware description language like VHDL or Verilog it is possible to define a model of a semiconductor device, simulate it, and then synthesize a physical implementation which then (ideally) behaves like the model in the simulation. This synthesis process is largely automated, depending on the target architecture (FPGA, ASIC, standard cells, ...). Note that the target architecture itself inherently defines a MoC, since it specifies computational (e.g. CLBs, CMOS-transistors) and communicational (e.g. longlines, voltage levels,...) means.

It is important to *choose the suitable MoC* for the task at hand, e.g. regarding the complexity or nature of the system. Simple systems like traffic lights or elevators can well be adequately modeled as FSMs. A system like a desktop computer could *in principle* also be modeled as an FSM, since

it always will be bound to possess finite memory with the different memory configurations forming the states. However, since the state space explodes exponentially with the available memory in the system it is neither feasible nor desirable to do so. Here, a MoC with an FSM with added memory, like a Finite State Machine with Data path (FSMD) [GR94], would be more suitable; and even more so a higher-level view like the von Neumann architecture [VN93]. And if the system at hand involves analog phenomena, means like differential equations and respective solvers come into play.

Also to be considered by a MoC is the treatment of time, especially regarding modeling and simulation. Again, the means should be suitable to the problem. For example, basically every digital system today is clocked synchronous; at every tick of the clock the system transitions into a new state. As a first approach, we can assume that the hardware which computes the state transitions is fast enough to compute the correct next state before the next clock tick, since the clock can always be slowed down. To model such a system, it is sufficient to view time as a series of discrete steps, corresponding to the clock ticks. This MoC is commonly called the *Synchronous Model of Computation*.

In this chapter, we look at related work regarding the use of MoCs in system design, as well as conversion means between different MoCs. Most notably, we introduce SystemC and its extensions and the MoCs used there. For the purpose of the discussion in this chapter, we now give some informal definitions of important MoCs. These will be (re-)introduced in a formal manner in Chapter 3.

- **Kahn Process Networks (KPN)** [Kah74, LP95] consist of processes communicating via unbounded FIFOs by writing and/or reading data token. If a process reads from an empty FIFO, it is blocked until another process writes to that FIFO. This MoC can be varied by bounding the FIFOS (bounded KPN, or B-KPN), with blocking write. While timing *can* be considered, this MoC is usually untimed.

- **Synchronous Data Flow (SDF)** models [LM87b, LM87a] are like KPN models with the restriction that the processes produce and/or consume the same amount of token every time they are executed. The processes of an SDF model can be statically scheduled (if the different data rates in the model are consistent) such that there is no need for runtime scheduling, and processes are never blocked by a FIFO access. SDF models are also untimed.

- The **Timed Data Flow (TDF)** MoC [GBVE08a] is a timed version of SDF; to each process execution a certain (fixed) time span is associated. This implies also a certain time span elapsing for each token passed via a FIFO and, in turn, imposes additional consistency constraints as not only the different data rates have to fit, but also the different process execution time spans relative to the data rates.

- The **Discrete Event (DE)** MoC [Fis73, Zei84, ZPK00, CL08] is probably the most common MoC in digital system design. Time is modeled as a discrete series of events, which are managed by a simulation kernel. There is no standard communication model, although often *discrete event signals* are used where writing to a signal triggers an event which effectively notifies the processes reading this signal.

2.1 SystemC

SystemC [IEE05, LMSG02a] is a C++ class library which implements a system modeling language. The development of SystemC started 1999, following a decade where the idea of *C-based design* became popular. As systems grew more and more complex, describing their functionality in a high-level language like C was more convenient than using dedicated hardware description languages like VHDL. The main purpose here was the creation of *golden models* which could be used to verify hardware prototypes, e.g. by providing test benches.

This development called for a more formal approach and motivated the development of SystemC and other formal C or C++-based modeling languages at the end of the nineties, most notably SpecC [GZD+00, FN01, CGO01], which shares similar ideas with SystemC (like channels, events and discrete event simulation semantics[MDG02]), but is based on ANSI-C. Unlike SystemC, it provides explicit constructs to model finite state machines.

Another approach in this category is the C-based Handel-C [Pag96, LWFK02], which targets a lower abstraction level than SystemC and SpecC and sees a lot of use in FPGA programming. Also targeting FPGA development is the Java-based JHDL[1] [BH98, HBH+99], which was also developed during this time, but didn't see any updates now for almost 10 years.

Apart from the description of functionality, these approaches enable to model timing, and (as the compiled code is executable) come with a straightforward way to *simulate* system models. Therefore, such system models are often called *executable specifications*. In the following, we have a closer look at how SystemC works.

2.1.1 Basic Syntax

SystemC provides several facilities for modeling and simulation of discrete event models, like signals, ports or modules. The most important class is the class `sc_module`. It can be used to either describe atomic entities of a model by means of processes, or to encapsulate a structural description of a sub-model[2].

To define a module, a class (or a struct) must be declared which inherits from `sc_module`, e.g. `class counter:sc_module {...};`. For convenience, there is a macro `SC_MODULE` available; if using it the previous class declaration would read as `SC_MODULE(counter) {...};`.If a module does not encapsulate a structural description, its behavior is described by processes, i.e. functions of the class. There are essentially[3] two kinds of processes in SystemC:

- `SC_THREAD`s are processes which are executed only once at the beginning of the simulation. They can yield to the simulation kernel by calling a function `wait()`. If they call `wait()` with a timespan as argument, they are continued after that timespan (of simulated time) has passed. By using an event as argument to `wait()`, they are continued when the event is triggered. With no argument, they are continued when an event is triggered which they are sensitive to.

[1] Just-Another Hardware Description Language

[2] It therefore corresponds to a combination of ENTITY and ARCHITECTURE in VHDL.

[3] There is a special kind of thread called `SC_CTHREAD` (i.e. clocked thread) for threads which are triggered by clock signals. Since they behave like `SC_THREAD`s in principal, we will not discuss them in detail.

- `SC_METHOD`s are processes which are sensitive to certain events. If one of these events is triggered, the method is executed. Unlike an `SC_THREAD`, an `SC_METHOD` is not allowed to call `wait()` (directly or indirectly).

```
SC_MODULE(clockgen)   // module declaration using the SC_MODULE macro
{
  sc_out<bool> out;   // output port

  SC_CTOR(clockgen)   // module constructor using the SC_CTOR macro
  {
    SC_THREAD(tick);  // thread which does everything
    state = false;    // state of the clock generator
  }

  void tick()           // is started once when the simulation starts
  {
    while(true)         // do everything in an endless loop
    {
      out.write(state); // write state to the output port
      wait(1,SC_US);    // continue in 1 microsecond
      state=!state;     // invert the state
    }
  }

  private:
  bool state;

};


SC_MODULE(counter)
{
  sc_in<bool>  clk_in;  // input port for clock signal
  sc_out<int>  cnt_out; // output port for counter value

  SC_CTOR(counter)
  {
    SC_METHOD(do_count);        // the method which increases the counter value...
    sensitive << clk_in.pos();  // is sensitive to the positive flanks of the clock
    state = 0;
  }

  void do_count()           // is started every time
  {
    cnt_out.write(state);   // write the counter value to the output
    state++;                // increase the counter value
  }

  private:
  int state;
};
```

Listing 2.1: Two example SystemC modules

For the events mentioned above, there is a dedicated SystemC class `sc_event` to represent them. To trigger them, they provide a method `sc_event.notify()`. The `SC_THREAD`s and `SC_METHOD`s of a module have to be declared in the module constructor, as well as the sensitivity of the processes to events. For the constructor, there is also a macro `SC_CTOR` available. See Listing 2.1

for two example SystemC modules: a clock-generator with an `SC_THREAD`, and a counter with an `SC_METHOD`.

Listing 2.1 contains two SystemC (template) classes not yet introduced: `sc_in<>` and `sc_out<>`. These are ports which have to be connected to signals. In line 35 of Listing 2.1, the `SC_METHOD` `do_count` is made sensitive to changes of the signal which the port `clk` is connected to. In this case, since the data type of the port `clk` is Boolean, it is even possible to restrict sensitivity to the $0 \rightarrow 1$ transitions (the positive edges) with `sensitive << clk_in.pos();`. In general, after declaring a class function as a thread or a method with `SC_THREAD(`*`<name>`*`)` or `SC_METHOD(`*`<name>`*`)`, it can be made sensitive to arbitrary many signal changes by using

`sensitive << `$port_1$` << `$port_2$` << ... << `$port_n$`;`

The in- and out-ports have access methods `read()` and `write()`, respectively, to access the signals they are connected to. Listing 2.2 shows how the two modules from Listing 2.1 can be connected at the top-level with a Boolean signal for the clock. The syntax is *`module.port(signal)`*, which is syntactic sugar for *`module.port`*`.bind(`*`signal`*`)` achieved by overloading the `()`-operator.

```
sc_main(int argv, char* argc[])
{
  sc_signal<bool> clk_sig;  // clock-signal
  sc_signal<int>  cnt_sig;  // signal for the counter value

  clockgen clk("clk");      // instantiate clock generator
    clk.out(clk_sig);       // connect its out-port to the clock-signal

  counter cnt("cnt");       // instantiate counter
    cnt.clk_in(clk_sig);    // connect its in-port to the clock-signal
    cnt.cnt_out(cnt_sig);   // connect its out-port to the counter value signal

  sc_start(100, SC_US);     // run the simulation for 100 microseconds
}
```

Listing 2.2: Connecting two SystemC modules

Another SystemC communication channel is the `sc_fifo<>` which allows for blocking as well as non-blocking access via specialized `sc_fifo_in` and `sc_fifo_out` port classes. In general, SystemC allows for the definition of custom communication channels and ports, which is crucial for extending SystemC.

2.1.2 Simulation Semantics

The SystemC DE simulation kernel governs the execution of `SC_THREAD`s and `SC_METHOD`s by maintaining a list of runnable processes. At the start of the simulation, all `SC_THREAD`s as well as all `SC_METHOD`s are put in this list[4]. There is no particular order, as the simulation time for all these executions is 0; in practice (at least with the Accellera reference implementation) it turns out that the order of the instantiation of the modules containing the processes as well as the order of the process declaration within the modules impose an order on the processes in this initial runnable list. However, the SystemC standard imposes no rules here; e.g. it would be perfectly valid to randomize the order of this initial runnable list. To distinguish processes executed at the same point of simulated time in the DE MoC, the notion of δ-time (or δ-step) is

[4]For `SC_METHOD`s, this can be prohibited by calling `dont_initialize()` when declaring the method.

used. If $p_0, p_1, p_2, ...$ are processes executed consecutively at the same simulation time, then p_0 is executed at δ-time 0, p_1 is executed at δ-time 1 and so on.

The processes executed now produce new events, either indirectly by writing to an port, or directly by calling the `notify()` method of an event. If `notify(t)` is called with an `sc_time` value `notify(t)` (*timed notification*), the processes waiting for this event will be made runnable after a span `t` of simulated time has passed. If `t`= 0 (δ-*notification*), they will be added directly to the runnable list and executed at the same simulation time. And if `notify()` is called without any argument (*immediate notification*), the processes in question will also be executed at the same simulation time, but from the next δ-step on; i.e. they will be put directly to the front of the runnable list.

2.1.3 MoCs in SystemC

A fundamental advantage of SystemC is that basically every MoC can be realized; either by *restricting* the computational means used by the system model, or by implementing new facilities using the underlying C++ language, thereby *expanding* the computational means - or both. For example, by using only FIFOs for communication between modules, several variants of the Process Network MoC can be implemented, depending on if...

- FIFO probing is allowed (i.e. blocking vs. non-blocking access)
- the FIFOs are bound in size
- or if timing facilities are used, e.g. the `wait()` command.

This freedom bears also some risk in the sense that it might be hard to say what MoC is actually used in a given SystemC model, especially since the full power of the C++ language is available. For example, a SystemC module can implement a finite state machine, a pushdown automaton or a Turing machine depending on the C++ facilities used. However, the elements of these MoCs are not well defined in the context of SystemC. E.g. any class variable `x` used in a SystemC module is a potential state variable which enlarges the potential state space by a factor of the size of the datatype of `x`. While this is not a problem regarding simulation, it can become an issue when the model is to be used also for other means, e.g. for synthesis.

One way to handle this situation is to use guidelines like in [LMSG02b], where it is described how to use the standard SystemC facilities to effectively model in different MoCs. The downside of such an approach is that not every MoC might be handled computationally effective. A different kind of guideline is the Accellera standard *SystemC Synthesizable Subset* [Ope04] which describes a subset of the SystemC standard which can be used for synthesis, together with synthesis semantics.

2.1.4 SystemC Extensions

Another way is to extend SystemC with new facilities and/or MoCs, e.g. by implementing new SystemC channels or even providing additional simulation kernels. There are several examples of such approaches:

- [PS04] added additional MoCs to SystemC, namely for finite state machines, communicating sequential processes and synchronous data flow. To this end, the SystemC kernel of the Accellera reference implementation[5] is extended directly, which is possible as it is Open Source. E.g. for SDF, facilities are added to describe SDF graphs, which then are analyzed automatically to provide a static schedule which is executed by a static scheduler, i.e. a dedicated SDF simulation kernel.
- HetSC [HV06] provides a SystemC-library for heterogeneous modeling in SystemC with additional MoCs. For HetSC, no SystemC kernel manipulations were necessary, but it also does not provide new simulation kernels. E.g. for SDF, dynamic scheduling is used. In addition, HetSC comes also with a synthesis approach for embedded software [Her08]. As HetSC was also part of the overall framework in the ANDRES project [HOH+07], there has been some work on its interoperability with SystemC AMS [HVG+07, HVG+08].
- In [RRG03] it is described how to use SystemC to implement Petri Nets. The only new facilities provided are dedicated communication channels for moving token from and to places, i.e. it partly has a guideline character. The author's ultimate goal is to synthesize petri nets.

Apart from these research works, there exist extensions to SystemC which are "official" in the sense that they are Accellera-standards. They provide new modeling and simulation facilities, together with rules how to use them, thereby explicitly or inherently defining new MoCs:

- The **SystemC AMS extensions** where developed by an OSCI working group to complement SystemC by means for modeling and simulation of **A**nalog and **M**ixed **S**ignal systems. (see Section 2.2).
- **Transaction Level Modeling** (TLM) provides means to abstract communication in models of digital systems. While TLM 1.0 and later TLM 2.0 (TLM2 for short) were originally extensions to SystemC, TLM2 is now an integral part of SystemC since version 2.3. (see Section 2.3).

Providing facilities to connect sub-systems modeled in a MoC provided by SystemC, SystemC AMS and TLM2 is an essential part of this thesis and is covered in Chapter 5.

[5] At the time this was done it was still the OSCI reference implementation

2.2 SystemC AMS

The idea of SystemC AMS is to provide an extension to SystemC with appropriate facilities to model analog mixed-signal systems, with a certain focus on signals processing [VGE03a, VGE03b, VGE05, GBVE08b]. SystemC AMS essentially targets the same or similar domains as VHDL-AMS [IEE07, SMG05, Rua01] and Verilog-AMS [Acc04, FO00], and partly also Simulink [Hof99], but aims at a higher abstraction level and better simulation performance.

The development of SystemC AMS started a few years after the development of SystemC [ECN+01, EGV+02, ES03]. SystemC AMS 1.0 became an OSCI standard in 2010 [Ope10], followed by SystemC AMS 2.0 in 2013 [Acc13], now as a standard of the Accellera SystemC Initiative.

The SystemC AMS extensions provides several new MoCs to support modeling and simulation of systems which contain analog subsystems and/or systems which are data-flow oriented (signal processing). It provides three new MoCs:

- **Electrical Linear Networks (ELN)**, i.e. electrical networks consisting of the linear elements like resistors or capacitors.
- **Linear Signal Flow (LSF)**, i.e. signal-flow graphs to form general linear differential equations.
- **Timed Data Flow (TDF)**, the timed variant of the SDF MoC.

In this thesis we will only consider the TDF MoC since it is the main MoC of SystemC AMS in the sense that it also "drives" the other MoCs: ELN and LSF essentially describe linear differential equations, but these equation systems are evaluated with the beat of TDF. That is, these two MoCs contain conversion means from and to TDF like TDF-controlled voltage sources or current sinks converting currents to TDF signals, and writing to or reading from these converters trigger the evaluation of the equation systems.

2.2.1 Basic Syntax

SystemC AMS provides its own module class `sca_core::sca_module` which is structured very differently form an `sc_module`. For one, there are no threads or methods to declare, but a fixed set of class methods with standard names which have to be overloaded in order to implement the functionality of the module. The most important method here is the method `processing()`, since this is actually the process called when the SystemC AMS simulation kernel executes the static schedule. Another important method is `set_attributes()` which is a method which is called before the simulation starts and where the data rates of the module ports, as well as the time steps for a port or the whole module are specified. Listing 2.3 shows a sine-wave source as an example.

```
SCA_TDF_MODULE(sine_source)
{
  sca_tdf::sca_out<double> out; // output port
  sc_core::sc_time timestep;    // the timestep of the port
  int rate;                     // the data rate of the port
  double frequency;             // the frequency of the sine wave

  sine_source(sc_core::sc_module_name nm,
```

```
        sc_core::sc_time _timestep,
        double _frequency,
        int _rate
  ): out("out"), timestep(_timestep), rate(_rate), frequency(_frequency)
  {}

  void set_attributes() // setting data rate and time step
  {
    out.set_rate(rate);
    out.set_timestep( timestep );
  }

  void processing()     // the actual process
  {
    double val;
    for(unsigned int i=0; i < rate; i++)
    {
      val =(sin( ( get_time().to_seconds() + i*timestep.to_seconds() ) * frequency * 2 * 3.14159 );
      out.write(val,i); // write the output values
    }
  }
};
```

Listing 2.3: Example SystemC AMS TDF module

As Listing 2.3 shows, SystemC AMS provides also a dedicated TDF port class. Consequently, a dedicated signal class `sca_tdf::sca_signal<T>` is provided as well. Instantiating and connecting TDF modules via TDF signals works exactly the same way like with DE modules and DE signals, but their semantics differ, since a `sca_tdf::sca_signal<T>` is essentially a FIFO - but it is not accessed like a FIFO. As Listing 2.3 shows in line 27, the access to the signal works more like accessing an array. Instead of pushing the output values, the command `out.write(val,i)` explicitly refers to the `i`th position in the FIFO the port `out` is attached to, i.e. the values could be written in any order.

This is possible because of the fixed data rates at each port. Similarly, the input token of an input port `in` is accessed with the command `in.read(i)`. That is, the FIFO access is not destructive as in usual FIFO access via a `pop()` command. Therefore, it is possible that one `sca_tdf::sca_signal<T>` can be connected to several readers.

This might look confusing at first since this does not seem compatible to a MoC communicating with FIFOs, which have always exactly one writer and one reader [6]. However, a `sca_tdf::sca_signal<T>` with more than one reader can be conceptually replaced with a module that reads from this signal as the *only* reader, an then writes the data token read to several new instances of `sca_tdf::sca_signal<T>`, one for each reader of the original signal. Therefore we don't have to adapt the SDF concept for the SystemC AMS MoC, and can use the usual FIFO semantics when looking at the TDF MoC from a theoretical point of view.

[6] Or *at most* one reader if we consider such FIFOs without a reader as an output stream.

2.2.2 Simulation Semantics

After starting the simulation with `sc_start()`, the SystemC AMS simulation kernel analyzes the structure of the TDF modules and their connections and identifies the connected components (commonly called *TDF clusters*). If possible, it computes a static schedule for every TDF cluster, if not, it halts the execution with a runtime error. More information on the schedulability of a TDF cluster is given in Section 3.4. The SystemC AMS simulation kernel also analyzes the time steps set at the ports and determines if they are consistent, halting execution with a runtime error if this is not the case. In fact, the time step only has to be set for at least one module port in a TDF cluster. The time steps of all other ports in the cluster then can be computed as a consequence of the data rates involved.

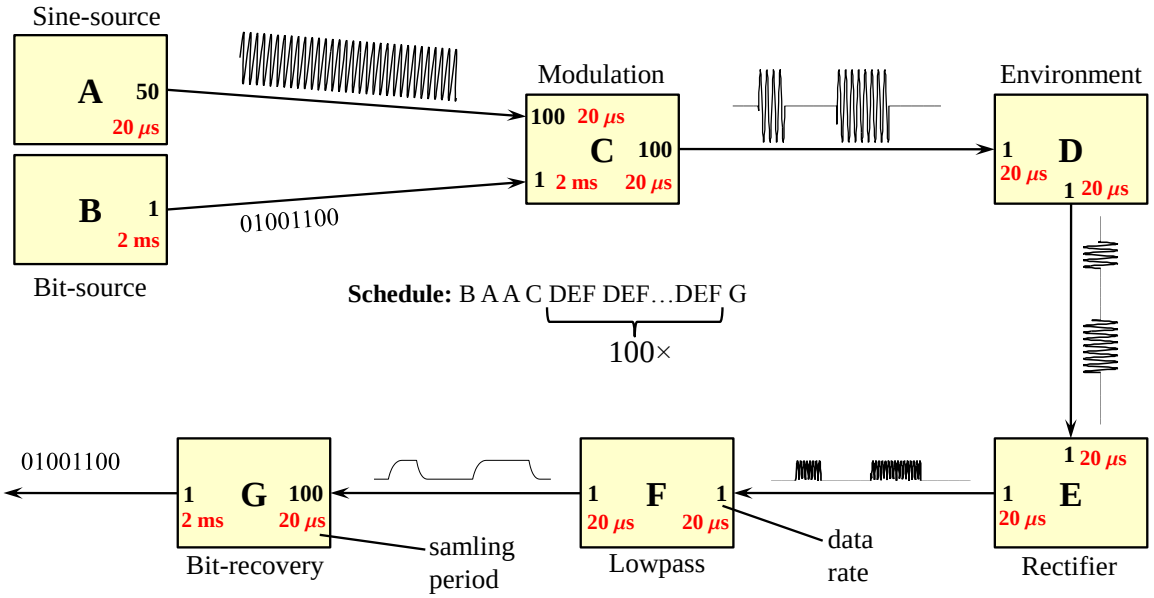


Figure 2.1: Example TDF cluster with static schedule, taken from [DHG08a]

Figure 2.1 shows an example of a consistent TDF cluster. The module C, for example, has a time step of $2ms$ at the binary input port with data rate 1. Since the data rate of the other two ports is 100, respectively, these ports have to have a time step of $2ms/100 = 20\mu s$. Note that the example in Figure 2.1 doesn't contain a closed loop. If a TDF cluster contains a closed loop, a delay (in token) has to be set at one port of a module which closes the loop (using the port method `set_delay()` within the `set_attributes()` method. This port can then also be provided with initial values for the delayed port.

With SystemC AMS version 2.0, the concept of *dynamic* TDF was introduced [BEG+11, RE12]. It allows for re-defining data rates and time steps during simulation time. However, in this thesis this new feature won't be considered. While it is of great practical use, from a theoretical point of view it is just a new TDF cluster setup during runtime resulting in another static schedule. Therefore, it is of no consequence regarding MoC conversion.

SystemC AMS comes already with conversion facilities to be able to connect to DE-signals to TDF modules, namely the converter ports `sca_tdf::sc_in` and `sca_tdf::sc_out`. Since a TDF module can't be sensitive to value changes, a `sca_tdf::sc_in` must be written actively every time the `processing()` method is executed. Also, to an `sca_tdf::sc_out` can only be written

to during the execution of `processing()`. The actual conversion is really just a manipulation of the static schedule in order to allow the TDF module to read from or write to the DE side at the most accurate time as possible. However, some inaccuracies are not avoidable. For example, during two consecutive reads from a DE signal by a TDF module, the value of the DE signal can in theory change an arbitrary number of times, i.e. there is loss of information. More on this in Section 3.4

2.3 TLM 2.0

The concept of Transaction Level Modeling is largely motivated by enhancing the simulation performance for typical digital systems, i.e. containing a microprocessor connected to a set of additional components (Memory, I/O) via a bus. The transportation of a series of bytes over a bus or a similar digital communication channel involves a series of events, possibly affecting several communication lines. Commands, addresses and data are converted into streams of Boolean values and back, accounting for one event per bit-flip. For example, transmitting 1 kilobyte via an SPI bus interface causes about 20000 events. If the modeling goal, for example, is the synthesis of a bus controller, then such low level events have to be considered. But if the goal is a functional model of a complex bus-based system, this is not necessary. It is enough to model the transfers of byte arrays as a whole, for example with a method call in a C-based simulation (see Figure 2.2), possibly assigning a certain time-span to the whole transfer for coarse time modeling.

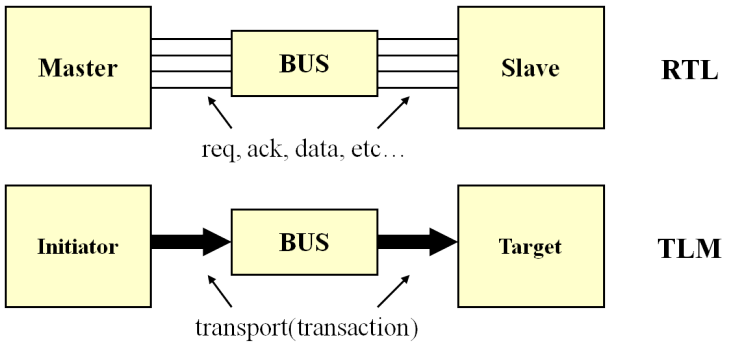


Figure 2.2: TLM vs. RTL

The idea is that the details of the communication can be neglected since they are straightforward (bus systems in general are well understood) and independent of the application. Even timing might not be important at first; we simply trust that we can build the communication facilities well enough such that all data will be delivered in time. Timing estimations, e.g. based on projections of latencies and data throughput of components and bus, should in general be enough even when designing for real time constraints. This degree of abstraction regarding communication and (in part) timing is probably the core of what is known today as Transaction Level Modeling.

There have been many proposals of extending and enhancing the concept of transaction level modeling, for example transaction events [HR13] or advanced temporal decoupling [Huf14]; sometimes targeting certain use cases like bridging TLM models with bus cycle accurate models [PDBR04], multi-accuracy modeling and power estimation [BSS07], and even wireless communication [DMHG10, MDH+12]. However, in this thesis we mainly restrict our view of TLM on the way it is specified in the SystemC standard.

2.3.1 TLM2 Transactions

TLM 2.0 [Ayn09] (TLM2 for short) was originally a SystemC extension library and an OSCI standard, but is now an integral part of SystemC as of SystemC 2.3. The first approach of defining a TLM methodology was TLM 1.0 [RSP+05], with the release of a SystemC extension library in 2005. However, TLM 1.0 is very different from TLM 2.0 in that it concentrates on FIFO-based message passing without defining standard transactions. TLM 1.0 is now also a part of the SystemC 2.3 standard.

TLM2 implements the concept of a transaction with a standard transaction class called the *generic payload*. The most important attributes of the generic payload are the command (read or write), the target address, the data length and a pointer to a piece of memory holding as many bytes as defined in the data length. Table 2.1 shows all parameters of the generic payload.

Table 2.1: Parameters of the TLM 2.0 generic payload

Attribute		Data-Type	Modifiable by Target	Modifiable by Interconnect
Command		`tlm_command`	✗	✗
Address		`sc_dt::uint64`	✗	✓
Data -	pointer	`unsigned char *`	✗	✗
	array	`unsigned char[n]`	✓	✗
	length	`unsigned int`	✗	✗
Byte enable -	pointer	`unsigned char *`	✗	✗
	array	`unsigned char[n]`	✗	✗
	length	`unsigned int`	✗	✗
Streaming width		`unsigned int`	✗	✗
DMI hint		`bool`	✓	✓
Response status		`tlm_response_status`	✓	✓
Extensions pointers		`tlm_array <tlm_extension_base *>`	✓	✓
Option		`tlm_gp_option`	✓	✗

In a TLM 2.0 model, every component assumes one of three roles (see also Figure2.3):

- *Initiators* are abstractions of bus masters (e.g. CPUs) and *initiate* transactions.
- *Interconnects* are abstractions of buses or similar means like crossbars and *forward* transactions. They possibly modify transactions in the process; for example they might change the target address from the virtual address in the virtual address space of the initiator to the corresponding physical address within the target.
- *Targets* are abstractions of bus slaves (e.g. memories or I/O components), and ultimately *process* transactions.

Table 2.1 indicates the "access rights" of interconnects and targets to the various parameters of the generic payload; initiators have full access, but have to obey to certain rules when setting up a transaction (for details see [IEE05]).

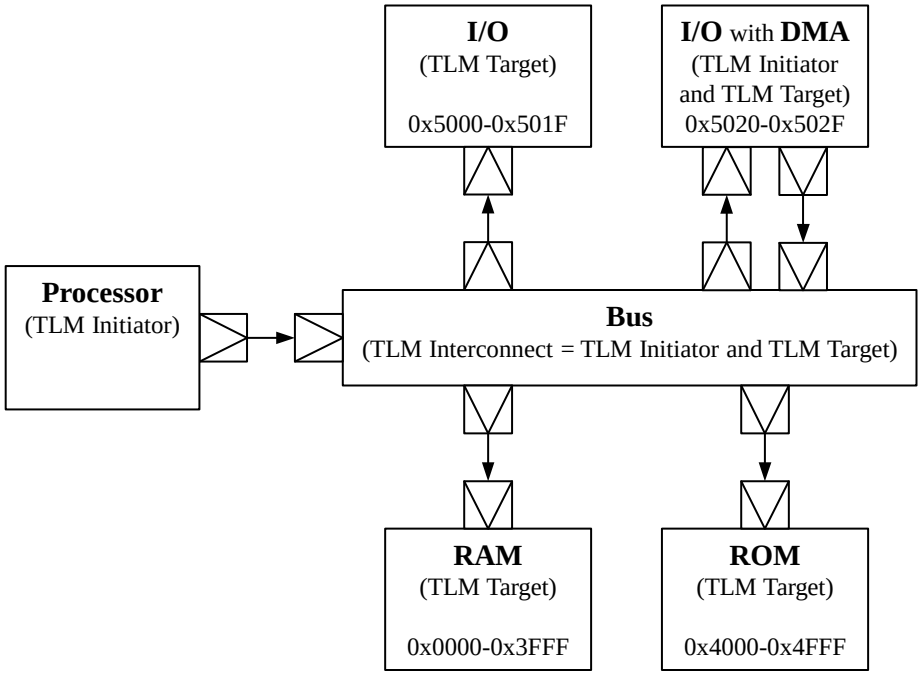


Figure 2.3: Example memory mapped bus system

Standard accesses in TLM 2.0 are always *memory-mapped*. When a TLM initiator wants to access a certain TLM target, it does so by accessing a certain piece of memory by its (virtual) address. This piece of memory can be literal memory (e.g. RAM), but it can also be, for example, a (number of) control registers of an I/O device like an A/D converter or a PWM driver. Every TLM target in such a model has an address range within the virtual memory space of the TLM initiator(s).

To set up a generic payload for such an access, the initiator first allocates memory of adequate size in the system which hosts the simulation. If it's a write access, the initiator also fills this memory space with the data needed, e.g. certain values for the targeted control registers. The initiator then instantiates a generic payload object and sets the command, the target address and the data length as needed, and also sets the dedicated data pointer parameter of the generic payload to the address of the allocated memory. After that, a reference to the generic payload object is sent to the target (possibly passing one or more interconnects) with a method-call (more on that later). The target then processes the transaction (e.g. for a read command copy data from it's memory to the memory specified by the data pointer, after which the initiator inspects the result (e.g. the response status and the data read).

Apart from command, address, data and data length, two more parameters are relevant for the considerations in this thesis: The response status and the streaming width. The *response status* is an enumeration type with which a target or an interconnect can specify if the transaction was processed without problems (`TLM_OK_RESPONSE`), or if there was some error (e.g. `TLM_ADDRESS_ERROR_RESPONSE` or `TLM_COMMAND_ERROR_RESPONSE`).

The *streaming width* can be used to write to or read from the same address several times with

just one transaction. If d is the data length, and s is the streaming width, there will be $\lceil d/s \rceil$ reads or writes, respectively. The purpose of this mechanism is to support access to targets which implement some kind of buffer. For example consider a transaction writing 6 bytes `ABCDEF` (i.e. the data length is 6). If the transaction is intended to write to a FIFO which is two bytes wide (e.g. having 16-bit integers as a data type), the streaming width has to be 2. The target would then first write `AB` to the FIFO, then `CD`, followed by `EF`.

While it is not relevant for this thesis, it is worth mentioning that the generic payload contains an extension mechanism which makes it possible to attach *arbitrary* data to a transaction on the fly. E.g. an extension might add additional possible commands, which would be an example of a *mandatory extension* that every component (or at least all targets) must know. In this case, the model is leaving the TLM2 standard.

But an extension can also be just meta-data attached by a component for it's own use, if only for the sake of simplifying implementation. E.g. an interconnect might attach routing information to a transaction only relevant to itself. This would be an example of an *ignorable extension* which would be still standard conformal.

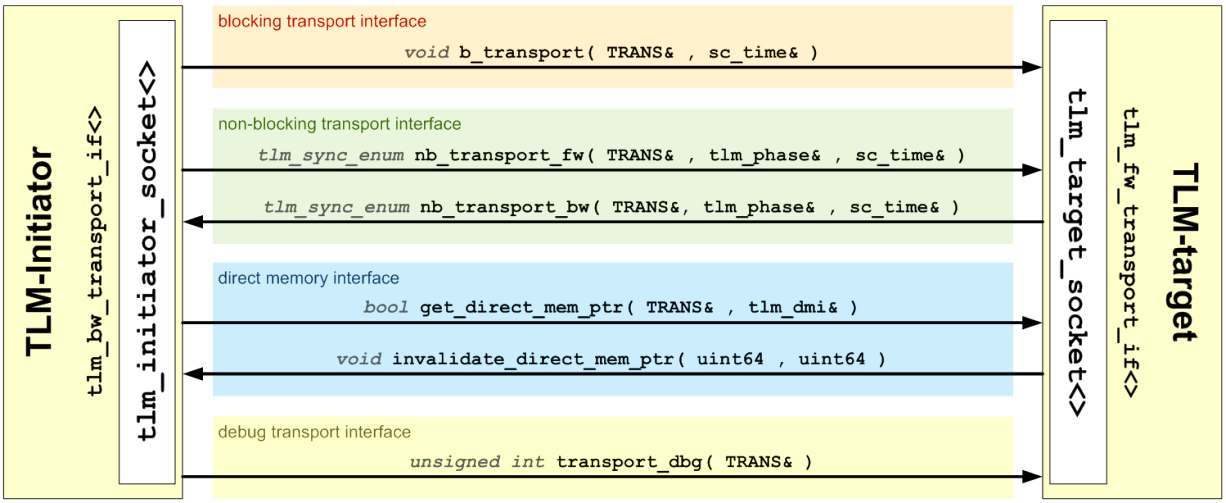


Figure 2.4: The method interfaces of TLM 2.0

2.3.2 TLM2 interfaces and Coding Styles

A TLM2 generic payload is passed by reference via method calls. To this end, TLM2 provides different method interfaces for initiators and targets, as shown in Figure 2.4. Initiators and targets have to implements these interface methods, and then export them via a *socket*[7]. By connecting their sockets, initiators, interconnects and targets can access their interface methods for communication via transactions. Which interface is used essentially depends on the *coding style* chosen:

- The **loosely timed coding style** typically uses the *blocking transport interface*, which only foresees one method on the target side, namely `b_transport(`*`transaction, delay`*`)`. A target which receives a transaction via the blocking interface is allowed to call `wait()`

[7] which is similar to a port

before returning, therefore blocking the initiator for that time span. Therefore, the lifetime of a transaction in this coding style has two significant points in time: when `b_transport()` is called, and when it returns.

- The **approximately timed coding style** typically uses the *non-blocking transport interface*. Here, we have a method `nb_transport_fw()` on the target side (the forward path), as well as a method `nb_transport_bw()` on the initiator side (the backward path). A target receiving a transaction via `nb_transport_fw()` is not allowed to call `wait()`. However, the lifetime of a transaction can now be subdivided even further, by sending it back and forth several times while changing the value of the additional *phase* parameter.

The semantic of the delay parameter in both interfaces is such that if a transaction arrives at simulation time t with a delay d, it has to be treated as if it would arrive at time $t + d$. How this information is handled depends on the coding style, but also on the modeling goals and the semantics of the component receiving the transaction.

In the loosely timed coding style, targets can simply wait for the time of the delay; e.g. to avoid inconsistencies caused by transactions arriving out of order. For example a read-transaction arriving at time t_1 with delay d_1 and a write-transaction arriving at time $t_2 > t_1$ with delay d_2 such that $t_1 + d_1 > t_2 + d_2$, with both transactions accessing the same data.

If such inconsistencies can not occur, or can be neglected w.r.t. the modeling goals, the target can also process the transaction right away. It would then return the transaction, possibly adding to the delay to model latencies. This avoids calling `wait()`, which speeds up the simulation, and essentially puts the initiator in charge of synchronizing with the global simulation time. It is possible for an initiator to decouple from the global simulation time even further by maintaining a local time offset which adds up these delays over several transactions (together with local latencies), and only synchronize when necessary or based on certain time quanta. This technique is known as *temporal decoupling*.

Approximately timed targets can in principle also directly return a transaction that way, but since the goal of the approximately timed coding style is usually to model transactions more fine-grained, they are mostly not implemented that way. Since they can't call `wait()` they typically employ a *payload event queue*, which stores a transaction and calls a callback function when the delay has passed; only then the transaction is processed. This also resolves eventual out-of-order transactions.

In any case, the target returns the transaction right away, and indicates via the `tlm_sync_enum` return value of `nb_transport_fw()` to the initiator how the transaction has been handled by the callee. `tlm_sync_enum` is an enumeration data type with the following possible values:

- `TLM_ACCEPTED` indicates that the transaction is unchanged and that the target will respond later via the backward path. This is a typical response when a transaction has been stored in a payload event queue.
- `TLM_UPDATED` is used when some parts of the transaction (including the delay or the yet to be explained phase parameter) have changed, but more is yet to come via the backward path.
- `TLM_COMPLETED` signals that the target is finished with the transaction.

Note that also `nb_transport_bw()` uses these return types; i.e. this is also relevant for initiators. With different values of the phase parameter (also an enumeration type), the transaction lifetime is now subdivided into several time-spans:

- `BEGIN_REQ` marks the start of the transaction by the initiator.
- `END_REQ` is set by targets at the (simulated) time when they received the transaction.
- With `BEGIN_RESP`, targets mark the point in time when they start to respond.
- `END_RESP` is used by initiators when they finally received the finished transaction.

The list above already indicates that usage of certain phases also depends on the component's role (initiator or target). Moreover, the phases must also be consistent with the `tlm_sync_enum` value returned. The SystemC LRM gives an extensive set of rules for this, which is shown in Figure 2.5. As can be seen, Using all phases is not mandatory. For example, a target can complete a transaction after the first call to `nb_transport_fw()`, which would correspond to the typical behavior of a loosely timed target. It is also possible to add additional phases, or use a custom set of phases altogether, which would sacrifice conformity to the standard.

The direct memory interface now can be used to allow initiators to access the memory of TLM targets *directly* for even faster simulation. This technique has also be assigned to the loosely timed coding style as it partially removes the need to model transactions altogether. The debug interfaces allows simple direct access for debug purposes, i.e. it is not really used for modeling. These two interfaces won't be considered any further in this thesis.

This concludes the introduction to the aspects of TLM2 which are relevant to this thesis; for a complete picture including all relevant coding rules we refer to the LRM [IEE05] which gives a good introduction to TLM2 going beyond what is usually found in LRMs.

The TLM2 standard looks very complicated (and indeed it is) especially when looking at the approximately timed coding style. But this complex set-up provides several benefits:

- **Fast Simulation:** By avoiding the modeling of low-level events, the simulation runs considerably faster.
- **Abstractness:** The transaction level covers not only bus communication, but also other approaches like Networks on Chip (NoCs).
- **Communication Requirements:** With a TLM model of a system, requirements for the communication like data throughput can be determined, e.g. by simulating typical applications and workloads.
- **Model interoperability** If only standard transactions (possibly with ignorable extensions) and standard phases are used, TLM modules can usually be connected in a plug-and-play manner.

However, some of the TLM2 concepts also stem from practical C++ coding considerations. For example, while the phase and the time delay are passed as separate arguments together with the transaction, they might as well be considered as part of the transaction itself. But keeping them separate has certain advantages regarding the coding. Therefore, in Section 3.5 we attempt to describe TLM in a more abstract manner, while also concentrating on the core concepts. Before that, however, we will ask a crucial question:

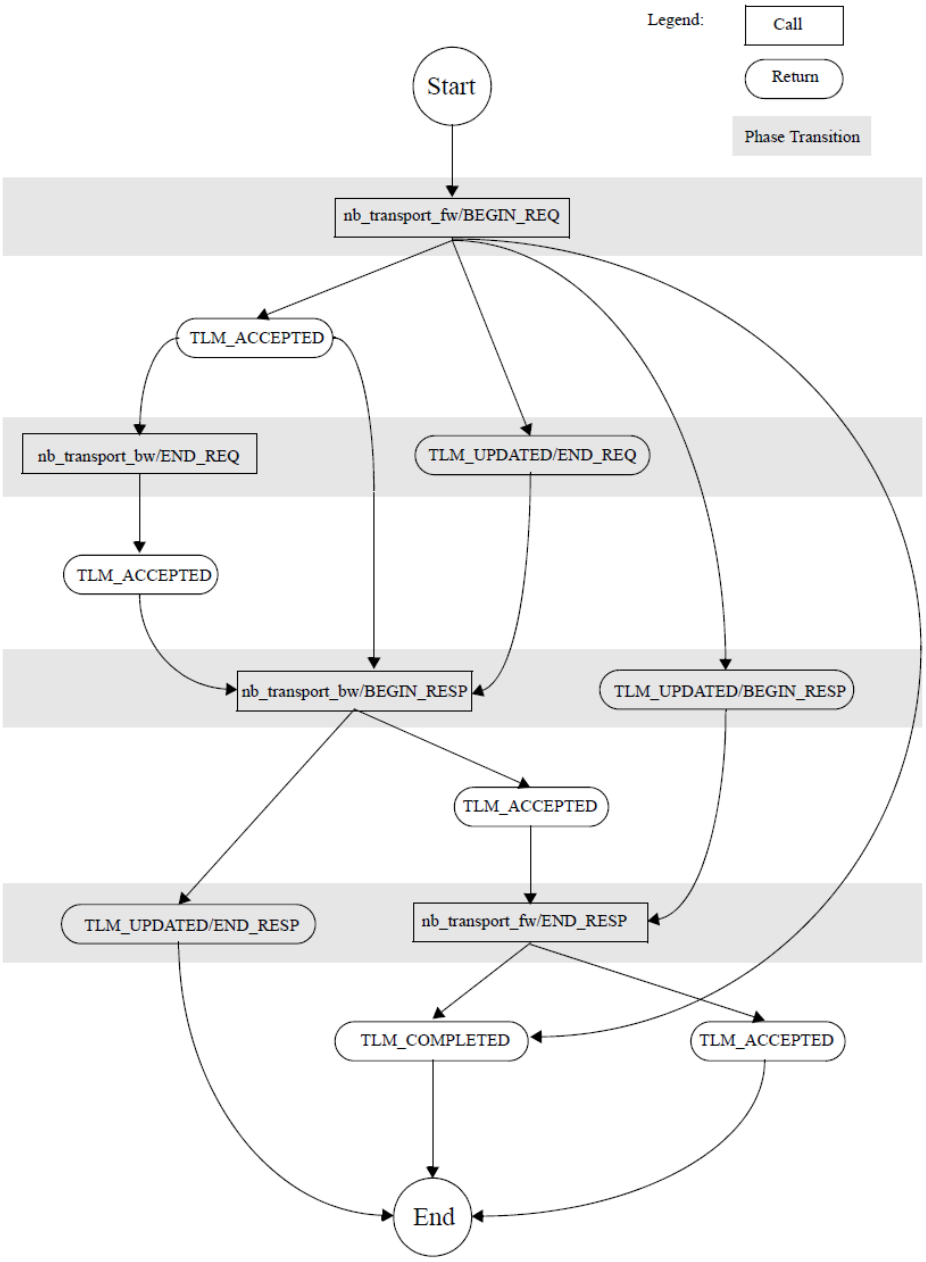


Figure 2.5: The phase rules of the TLM 2.0 base protocol, taken from [IEE05]

2.3.3 What kind of a MoC is TLM?

This is not easy to answer, as the question "What is transaction level modeling?" itself is not easily answered. As the overview of SystemC TLM2 above already shows, there are several ways to employ TLM in SystemC, depending on which interface/coding style is used, and if techniques like temporal decoupling or direct memory access are applied. In the SystemC LRM itself [IEE05], it is suggested that the loosely timed coding style should be used for software development, and the approximately-timed coding style for hardware/architectural analysis and verification. The precision needed regarding the timing for the task at hand indicates, for example, if temporal decoupling can be used, or how many phases a transaction has. However, the borders are blurry.

For example, there is no rule prohibiting the use of temporal decoupling together with the non-blocking interface, although it is suggested that initiators in the approximately timed coding style typically run in lock-step with the global simulation.

So transaction level modeling is really a family of approaches, and these approaches are often classified depending on the modeling goals. Popular terms used are *Programmer's View* (PV) for models which are exact enough to support software development; this is often associated to untimed models which are sufficient with respect to functional correctness. This view is extended to the *Programmer's View + Timing* (PV+T) if some timing is needed. And the term *Architectural View* (AV) is used for models employed in hardware design and verification (with *Verification View* (VV) being another variant here), which is often associated to fine-grained (even cycle accurate) modeling of timing.

In [BAGK07], this use-case based abstraction is criticized. While the authors agree that a Programmer's View is reasonably well defined, they argue that certain architectural and verification activities can be performed just fine with PV models, while others might need cycle accurate models. In general, they don't see a viable classification of TLM abstraction levels.

Regarding transactions themselves, the authors propose a simple definition as "containers for information that is communicated between multiple components of a system", and classify the type of information contained into 12 categories along two dimensions:

- *Ownership and stability* captures lifetime and mutability of transaction data (the two rightmost columns of Table 2.1 are a flavor of this); here they identify 4 classes.
- For the *type of data* they identify 3 classes: functional data, performance data and meta data.

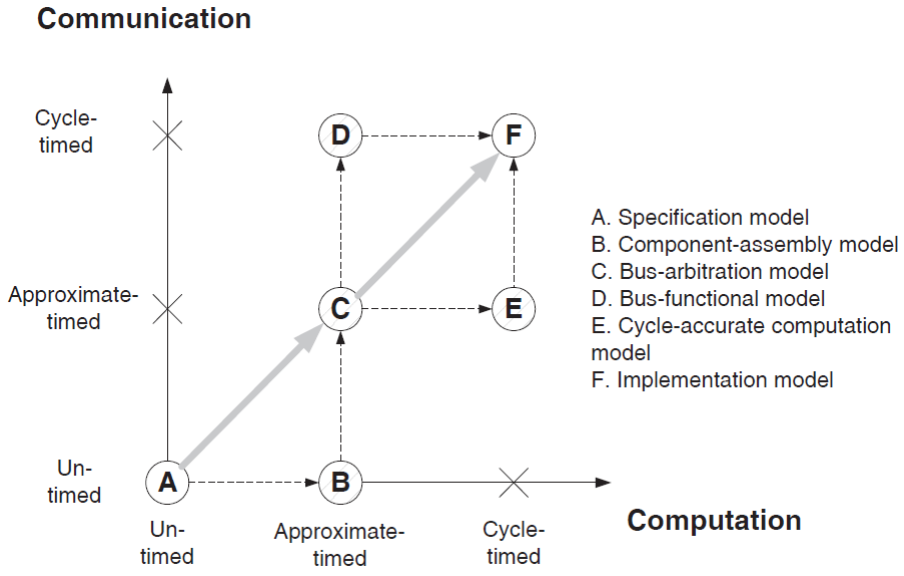


Figure 2.6: TLM classification, taken from [CG03]

In [CG03], transaction level models are classified according to the granularity of timing used for communication one the one hand, and the timing granularity of the computation on the other

hand. The authors consider 3 levels of granularity: untimed, approximate-timed, and cycle-timed. Within this two-dimensional scheme they identify 6 different classes of TLM models from specification models to implementation models (see Figure 2.6). That is, they also classify according to the modeling goals (or use cases), but base this classification on timing granularity considerations.

In [SKR11], a TLM Meta-model is proposed where the communication and communication within a TLM model is described by an abstract language, given by a grammar, called the *Transaction-level Modeling Language* (TLL). TLM models are composed of components with dedicated connectors and transaction handlers, which correspond to the sockets and interface methods in TLM2. The two TLM2 coding styles are then described by TLL subsets and certain restrictions on the state of components.

So what does all this mean for a TLM-MoC? While the notion of the information exchanged in TLM models is relatively clear (and captured well by the TLM2 generic payload), this is not the case for the treatment of time. The latter ranges from being untimed to being cycle accurate, with many shades in between. And because of temporal decoupling, transactions might not even arrive in the right order. In any case, the DE-MoC seems to be an appropriate base for a TLM MoC, at least when considering the concepts of TLM2.

Since the ultimate goal regarding TLM in this thesis is to define conversion semantics between TLM2 models and SystemC AMS TDF models, we will base our definition of TLM MoCs in Section 3.5 on the TLM2 standard in that we will define two MoCs: a loosely timed TLM MoC (LT-TLM) and an approximately-timed TLM MoC (AT-TLM).

2.4 Other approaches and related work

There are a many modeling and simulation frameworks available, both commercial and Open Source. Especially those frameworks with a focus on mixed-signal (or mixed-domain) systems modeling using different MoCs are related to this thesis.

Probably the best known framework in this category is Ptolemy II [EJL+03, LJ99], which is a Java-based software framework for modeling and simulation of heterogeneous systems. The basic concept of Ptolemy II is that a set of components called *actors* are executed concurrently under the governance of so-called *directors*, which are software components themselves. The director now defines the MoC used by the actors under its care.

Therefore, the Ptolemy II framework is extensible by definition, especially as it is an Open Source framework; i.e. custom directors can be added to support additional MoCs. In particular, there are directors available for all the MoCs mentioned so far and more, e.g. continuous-timed MoCs or MoCs similar to the concept of Communicating Sequential Processes as introduced in [Hoa78]. To combine different MoCs, different directors are combined under the control of finite state machines; a concept which is called *model models* [GBA+07, LT10].

Most modeling frameworks have, in a way, a "black box" view on their system components. While the means of communication, modeling of time and process schedules are well-defined, these frameworks usually do not strictly define the mode of computation that happens within a process when it is executed. For example, a SystemC thread can work like a stateless function, a finite state machine or even (at least conceptually) like a Turing machine by using an unbounded amount of storage. The facilities of SystemC don't capture this "inner" behavior; it is described

by arbitrary constructs of the underlying programming language, in this case C++. The same is true for example, for SystemC AMS, SpecC, Ptolemy II (although the last two have dedicated FSM facilities [FN01, Lee09]), or standard modeling languages like VHDL. Also the formalism used in this thesis, which will be introduced in the next chapter, follows this path.

The modeling framework ForSyDe (FOrmal SYstem DEsign)[San03, Jan06], which is based on the functional programming language Haskell [HPJW+92] is different in this aspect, as it defines not only the communication (which is also modeled by means of signals), but also the computation completely. It is based on the concept of process constructors [LSV98, Jan04].

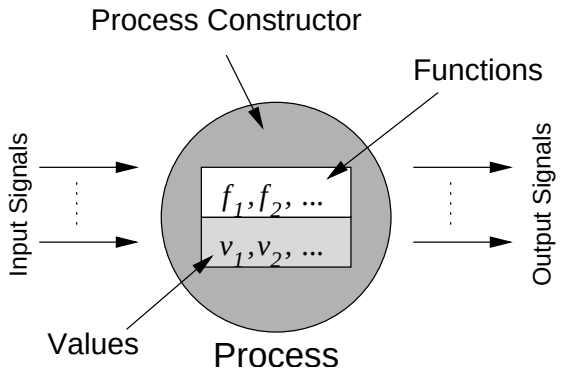


Figure 2.7: General concept of a process constructor, taken from [AND09]

A process constructor (see Figure 2.7) describes a process, e.g. the inputs, outputs, and states, as well as the functions used to compute specific states and/or outputs. For example an SDF process constructor

$$mooreSDF(N, M, C_{in} = (c_{in}^1, ..., c_{in}^N), C_{out} = (c_{out}^1, ..., c_{out}^M), g, f = (f_1, ..., f_M), w_0)$$

describes an SDF process which reads from N inputs with data rates $c_{in}^1, ..., c_{in}^N$ and writes to M outputs with data rates $(c_{out}^1, ..., c_{out}^M)$. It has an internal state w with start value w_0, and the state transitions are computed with g, which takes the N input signals and w as an input. The function $(f_1, ..., f_M)$ compute the M output values from the state. Such a complete description allows not only to describe and simulate a model, but also to treat the transformation of a model in a formal manner[SJ04].

Polymorphic signals [Sch07, SGW05, GSWB07] are a predecessor to this work in that they also provide automatic MoC conversion means for SystemC and SystemC AMS. Some of the C++ programming techniques used in the polymorphic signals to enable them to adapt automatically to different port classes have also been used in the Converterchannels. However, the conversion scope of the Converterchannels is broader as it offers more MoC conversions, and also dedicated options for certain MoC conversions.

Also the internal conversion approach is different. Polymorphic signals transform a signal of MoC M_1 with data type D_1 first into a generic internal representation, after which it is transformed to the target MoC M_2 with data type D_2. In particular, the values of data type $D_1$are first converted into `double` values, and then transformed to data type D_2.

In contrast, the Converterchannels have no internal generic representation, but have dedicated converters for every possible pair of MoCs, as well as every possible pair of data types. Therefore, the converter channels are more of an convenience layer to automatically choose converters out of a library, with means to customize them.

3 A formalism for MoCs and computational system models

In this chapter, we introduce a simple formalism to consider MoCs in an *operational* approach. This point of view is much closer to the SystemC environment where we ultimately want to implement these concepts as opposed to the functional view of the denotational approaches in [LSV98, Jan04]. The basic idea is to just provide a framework for process control *without* an explicit communication concept. Communication between processes will exclusively be implemented with variable access, often supported by helper processes. We start with the following

Definition 3.1 (processes and process control operations)**.** *A process p is a sequence of instructions which can read and manipulate variables using arbitrary mathematical operations, conditional execution and loops. With the following operations it can control its own execution and that of other processes:*

1. ***End**: With this operation p ends its execution.*
2. ***Freeze**: With this operation, p can halt its execution (direct freeze) until it is unfrozen by another process.*
3. ***Call(q)**: This operation starts another process q. If q freezes, p freezes as well (indirect freeze). p is continued when q ends. If q is frozen, this operation is inconsequential.*
4. ***Invoke(q)**: This operation starts another process q or, if q is frozen directly, unfreezes it. p is continued when q ends or freezes. If q is frozen indirectly, this operation is inconsequential.*

If a process q was called by a process p and ends, p is continued before any other process.

No time is associated to the execution of a process.

The idea of Definition 3.1 is to provide a framework which is powerful enough to describe all Models of Computation of interest. By postulating that the execution of processes does not consume time (therefore making the whole framework inherently untimed), we avoid confusing simulated time and simulation time. If a MoC is timed, time will be represented by means of processes and variables. Figure 3.1 shows an illustration that includes examples for all the rules given in Definition 3.1.

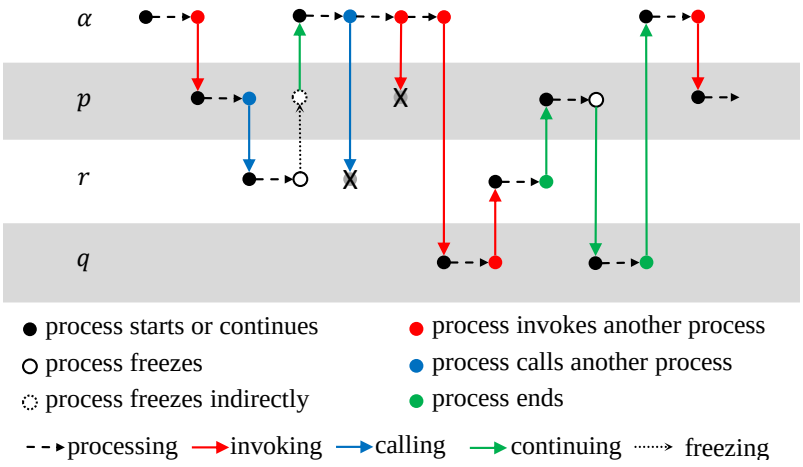


Figure 3.1: An example illustrating the rules of the formalism

With the Freeze/Invoke concept we can model process control concepts like the event-based control in SystemC, or the FIFO blocking mechanisms in process networks. How processes continue after freezing themselves depends on the MoC and how it is implemented in our formalism. For example, before a process p freezes itself, it can call another process q to pass a reference to itself (or similar information) such that q (or a third process) can wake it up again later with $Invoke(p)$. Or there might be a process governing the overall model execution which wakes p up later.

The restriction that frozen processes can't be called again until they are unfrozen and end makes sense regarding hardware modeling: Since a process represents the functionality of a physical object which can't be accessed or used by two or more objects during the same time period, we also don't need to do so with the process. In fact, for almost all the MoCs considered in this thesis, the process control concept of Definition 3.1 is sufficient.

There is only one MoC where it isn't sufficient: The TLM2 loosely timed coding style. Consider the following example: A TLM-target receives a transaction via a call to `b_transport()`, on which it calls `wait(t)` (which corresponds to a Freeze) for some timespan `t`. Since there can be more than one initiator in the model, it is possible that during this timespan, a second transaction arrives via `b_transport()`. In SystemC this works since in C++ this results in the execution of a new copy of `b_transport()` with a new call stack.

Therefore we yet have to capture this concept. However, we still need means to express the concept of a call stack. In general, we haven't specified any inputs or outputs of processes yet. In fact, we will sacrifice the common concept of input parameters and return values for a paradigm where processes access variables in an open computational space. The relation of a variable v to a specific process p is defined implicitly by the way v is accessed by p as well as other processes:

Definition 3.2 (variables of a process, state variables, input, output, variable and process references)**.** *For a process p we define the following:*

1. $Var(p)$ *denotes the set of variables of a model which are accessed by* p.
2. $Var_R(p) \subset Var(p)$ *is the set of variables which are read by* p.
3. $Var_R^{EX}(p) \subset Var_R(p)$ *is the set of variables which are read exclusively by* p.
4. $Var_W(p) \subset Var(p)$ *is the set of variables which are written by* p.
5. $Var_W^{EX}(p) \subset Var_W(p)$ *is the set of variables which are written exclusively by* p.
6. $In(p) := Var_R(p) - Var_{RW}(p)$ *is the* ***input*** *of* p.
7. $In^{EX}(p) := Var_R^{EX}(p) - Var_{RW}^{EX}(p)$ *is the* ***exclusive input*** *of* p.
8. $Out(p) := Var_W(p) - Var_{RW}(p)$ *is the* ***output*** *of* p.
9. $Out^{EX}(p) := Var_W^{EX}(p) - Var_{RW}^{EX}(p)$ *is the* ***exclusive output*** *of* p.
10. $State(p) := Var_R(p) \cap Var_W(p)$ *is the set of* ***state-variables*** *of* p.
11. $State^{EX}(p) := Var_R^{EX}(p) \cap Var_W^{EX}(p)$ *denotes the* ***exclusive state variables*** *of* p.
12. $Context(p) := Var_R^{EX}(p) \cup Var_W^{EX}(p)$ *denotes the* ***context*** *of* p.

A reference to a variable v will be denoted as ref_v.

Process references are represented by the process name.

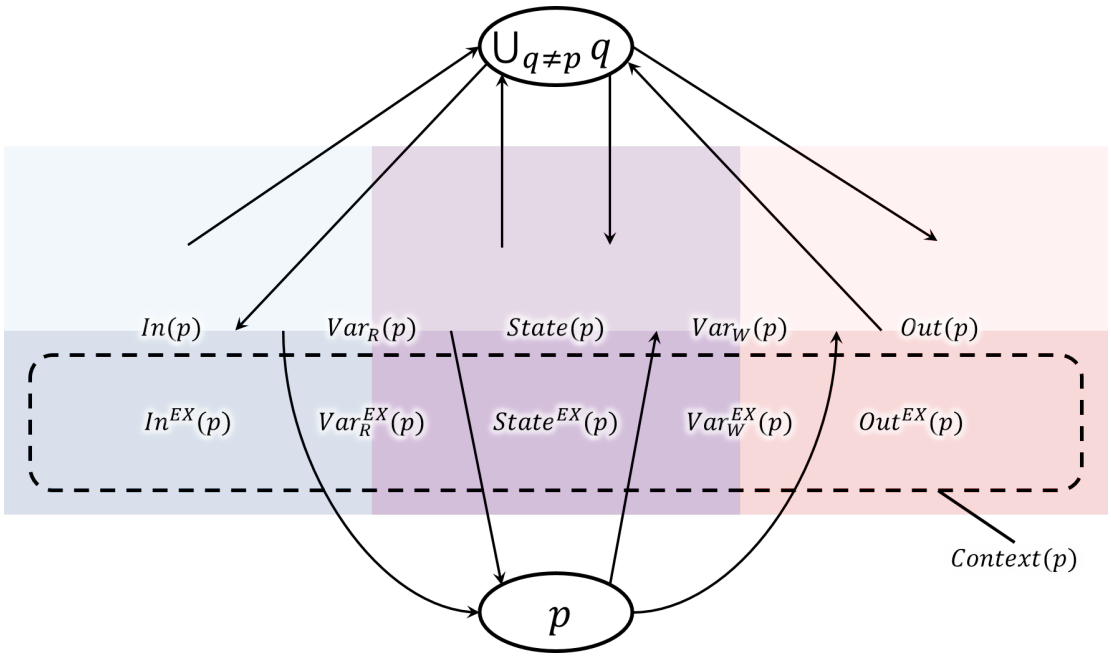


Figure 3.2: Overview of the subsets of $Var(p)$.

Figure 3.2 visualizes Definition 3.2. Except $Context(p)$, which is shown explicitly with a dashed box, the other sets encompass all the areas that they touch. The arrows indicate the sets that are accessed by p, and those accessed by other processes. Incoming arrows indicate reading access, and outgoing arrows indicate writing access. The access encompass all the areas touched by the start- or endpoint of the arrow within $Var(p)$.

Dynamic creation of variables is possible, i.e. p can add new variables to the computational space. For example, if p maintains a list of variables $\{v_0, v_1, ..., v_n\}$, it can add a variable v_{n+1} to the list. If p adds a new variable v, then $v \in Var_W(p)$ first, and by passing references to v to other processes, p enables $v \in Var(p) - Context(p)$.

Although important in practice, data types will not be considered here. For the purpose of this thesis, all variables can be considered as unspecified data token, or references to variables or processes, respectively. Occasionally we might specify variable domains explicitly for clarity or convenience, especially when talking about TLM. With respect to practical applications, we can assume that processes perform data type transformations (if necessary) before writing to inputs of other processes. The converter channel presented in Chapter 5 also deals with data type conversion[1], but this conversion is implemented in a way independent from (or, in some sense, orthogonal to) the MoC conversion.

Variables in a model can have initial values. In that sense, variables which are never written can be considered as *constants*. These constants can be though of as fixed values which are part of a model, for example containing (a part of) the structure of a model by means of references to processes as we will see in the case of discrete event signals.

For convenience, we use for a process p with $In^{EX}(p) = \{v_1, v_2, ..., v_n\}$ the notation

$$Call(p(w_1, w_2, ..., w_n))$$

to express first writing the parameters $w_1, w_2, ..., w_n$ to $In^{EX}(p)$ (i.e. setting $v_1 = w_1, v_2 = w_2, ..., v_n = w_n$) and then calling p.

Finally, we define how to handle calling the same process with a different call stack:

Definition 3.3 (Copy, CopyCall, CopyInvoke)**.** *With the operation $p' =$**Copy(p)**, a process q can create a copy p' of a process p.*

- ***Copy(p)** also creates a copy $Context(p')$ of $Context(p)$.*
- *All other variables in $Var(p')$ are the same as in $Var(p)$, i.e.*

$$Var(p') - Context(p') = Var(p) - Context(p)$$

- *When a copied process p' ends, the content of $Out^{EX}(p')$ is copied to $Out^{EX}(p)$, after which $Context(p')$ is discarded.*

For convenience, we define two derived operations:

- *The operation **CopyCall(p)** is equivalent to $p' =$Copy(p) followed by Call(p').*

[1] In fact, most of the code in the implementation deals with data type conversion

- *The operation* ***CopyInvoke(p)*** *is equivalent to* $p' =$*Copy(p) followed by Invoke(*p'*). In this case,* p' *will always start at the beginning so it makes sense to write CopyInvoke(*$p(In^{EX}(p))$*).*

While we don't explicitly forbid it, the idea is that processes which are copied are never called or invoked themselves, but rather serve as a template. Therefore the copying the values of $State^{EX}(p) \subset Context(p)$ to $State^{EX}(p') \subset Context(p')$ is to be understood as instantiating $State^{EX}(p')$ with initial values. As mentioned above, we will use copied processes only in the context of the loosely-timed TLM models.

Another way of thinking about processes and variables would be to consider a process p implemented as Turing machine M_p which accesses several tapes.

After reading from the tape, the Automaton goes into a new state, producing as output a symbol to be written at the current tape position and a direction (left, right or stay) to go next on the tape. The Turing machines in our model would operate on several tapes, some of them shared with the other Turing machines in the model. The operations from Definition 3.1 can be implemented by means of states:

- **End**: This corresponds to a dedicated end-state.
- **Freeze**: This corresponds to dedicated freeze-states. Upon unfreezing, the machine continues from the state it froze. Note that while being frozen, other machines might have written on a shared tape which might influence the next state transition and output.
- **Call(q)**: On the side of Turing machine M_q implementing q this simply corresponds to starting M_q in the start state. The Turing machine M_p implementing the caller process p transitions into a dedicated call-state from where it continues once M_q has reached its end-state.
- **Invoke(q)**: This corresponds to starting the Turing machine implementing q in the start state or the state where it last froze. The caller switches into an Invoke state from where it continues after the callee goes into an End- or Freeze-state.

3.1 Computational System Models

Definition 3.4 (computational system model, sub-model, component)**.** *A* ***computational system model*** *(CSM) is a tuple* $\mathcal{M} := (\mathcal{P}, \mathcal{V})$*, with* $\mathcal{P}$ *being the set of processes and* $\mathcal{V}$ *the set of variables of the model.* $In(\mathcal{M}) := \bigcup_{p\in\mathcal{P}} In(p) - \bigcup_{p\in\mathcal{P}} Out(p)$ *are the inputs of* $\mathcal{M}$ *and* $Out(\mathcal{M}) := \bigcup_{p\in\mathcal{P}} Out(p) - \bigcup_{p\in\mathcal{P}} In(p)$ *are the outputs of* $\mathcal{M}$*.*

A ***sub-model*** $\mathcal{M}'$ *of a CSM* $\mathcal{M}$ *is a tuple* $(\mathcal{P}', \mathcal{V}')$ *with* $\mathcal{P}' \subset \mathcal{P}$ *and* $\mathcal{V}' \subset \mathcal{V}$ *such that*

$$\bigcup_{p'\in\mathcal{P}'} Var(p') \subset \mathcal{V}'$$

.

We define $State(\mathcal{M}') := \bigcup_{p'\in\mathcal{P}'} Var_R(p') \cap \bigcup_{p'\in\mathcal{P}'} Var_W(p')$ *as the state variables of* $\mathcal{M}'$*, and* $State^{EX}(\mathcal{M}') := \{v \in State(\mathcal{M}') | v \notin \mathcal{V} - \mathcal{V}'\}$

*A **Component** C of a CSM $\mathcal{M}$ is a sub-model $(\mathcal{P}', \mathcal{V}')$ such that*

$$\exists v \in State^{EX}(C) \text{ such that } v \notin \bigcup_{p' \in \mathcal{P}'} State^{EX}(p')$$

That is, a sub-model $\mathcal{M}'$ has to (naturally) contain all the variables its process access. A component is a sub-model that has exclusive state variables which are not exclusive state variables of one of its processes; i.e. a component cannot be (reasonably) sub-divided into further sub-models.

For simplicity, we will for the most part assume CSMs to be closed in the sense that there are no dedicated inputs or outputs. In this setting, a test bench which provides inputs to a system model and checks the results would be a sub-model of the whole model, with the system model under test being another sub-model. However, any process p in such a closed CSM which acts as a pure data source ($In(p) = \emptyset$) or data drain ($Out(p) = \emptyset$) could be considered as an input or output to the model, respectively.

Definition 3.5 (execution semantics, executable system model)**.** *An **execution semantic** for a computational system model $\mathcal{M}$ is a set of execution variables $\mathcal{V}_{EX}(\mathcal{M})$ and execution processes $\mathcal{P}_{EX}(\mathcal{M})$ with a dedicated start process $\alpha \in \mathcal{P}_{EX}(\mathcal{M})$. An **executable system model** is the union of execution semantic and model, and its execution starts by executing α.*

For example, the execution processes can constitute the kernel of a simulator, which generates an execution order for the set of processes in the model. In that case a typical execution variable would the simulation time or a priority queue to organize dynamic process schedules. An execution process can also implement other analysis means than simulation. For example, the model could be a netlist of logic gates, the latter attributed with delays, and the execution process performs a longest path analysis. Therefore, while an execution semantic is comparable to a director in Ptolemy [EJL+03], it is broader in scope as it covers also applications beyond simulation.

It is important to note that the (parts of the) models that we investigate in this thesis using this formalism are considered to be *static* in the sense that there is no initial analysis or setup phase performed on them (e.g. like the elaboration phase in SystemC). In other words, the models we consider here are runnable, the α process has every information it needs (like process-lists or schedules), as does every process in the model. And if certain variables need initial values (like initial values in feedback buffers or initial states of processes), this is also been done already. When it comes to conversion between MoCs, however, we might on occasion we might describe how a certain item (like a schedule) has to be changed to make conversion work.

We can now finally define what we mean by a MoC in this setting:

Definition 3.6 (Model of Computation for computational system modeling)**.** *A MoC for computational system modeling is a restriction on the processes and variables used in a system model, together with an execution semantic.*

Examples:

- Formalizing the data transfer between processes, e.g. by means of FIFOs or signals
- Requiring that (certain) processes can only be called or invoked by an execution semantics process.

- Require that certain processes are only invoked, or only called
- Restricting the use of the $Freeze$ operation.

In the rest of this chapter, the MoCs of interest are described using the introduced formalism.

3.2 The Discrete Event Model of Computation

We start off by discussing discrete event (DE) models first, since this MoC is the base MoC of SystemC and will therefore be also a basis of our considerations. However, we will use a different, simpler version of an event for our purposes:

Definition 3.7 (Event)**.** *An **event** in a CSM $\mathcal{M} = (\mathcal{P}, \mathcal{V})$ is a pair $(t, p) \in \mathcal{T} \times \mathcal{P}$ where $\mathcal{T}$ is the time-base. $\mathcal{T}$ is usually identified with the positive real numbers $\mathbb{R}_{\geq 0}$.*

An event (t, p) as in Definition 3.7 states that a process p is to be called (or invoked) at simulation time t. This version of an event is more true to what we colloquially understand when talking about an event: something that occurs at a specific point in time and then passes, i.e. an event only happens once. In SystemC, an event is a data structure where processes can subscribe to (via sensitivity or `wait()` commands), such that when the event is notified, all the subscribers are executed. In contrast, our version of a discrete event kernel works as follows:

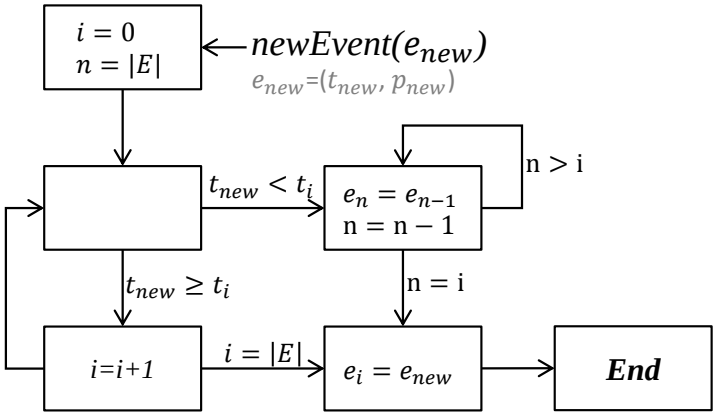


Figure 3.3: *newEvent* process of the DE execution semantic

Definition 3.8 (Discrete Event Execution Semantics)**.** *The Discrete Event (DE) execution semantics consists of*

- *a time variable t_{DE} which can be read by every process.*
- *a list $E = \{e_0, e_1, ..., e_{n-1}\}$ of events $e_i = (t_i, p_i)$*
- *a list $F = \{f_0, f_1, ..., f_{k-1}\}$ of processes.*
- *a start process α_{DE}*
- *a process newEvent with an input event $In(newEvent) = e_{new} = (t_{new}, p_{new})$*

such that

1. *E is always ordered according to time, i.e. $i < j \Rightarrow t_i \leq t_j$.*
2. *If newEvent is called, it inserts e_{new} into E with the largest possible index. (see Figure 3.3)*
3. *If α_{DE} is started, it sets $t_{DE} = 0$ and invokes all processes in F. After that, α_{DE} goes into a loop where it repeatedly removes (t_0, p_0) from E, sets $t_{DE} = t_0$ and checks if $p_0 \in F$. If this is the case, it invokes p_0, otherwise it calls p_0. If E is empty, α_{DE} ends (see Figure 3.4). Only α_{DE} can write to t_{DE}.*

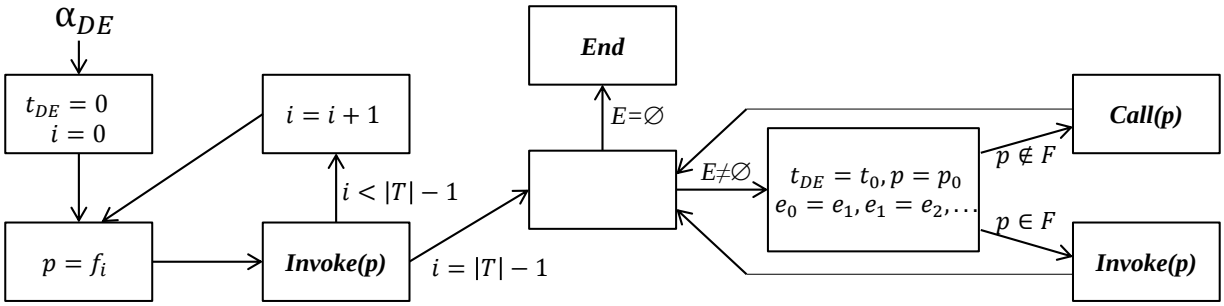


Figure 3.4: The α_{DE} process of the DE execution semantic

Figure 3.4 shows a diagram visualizing α_{DE}. Throughout this thesis, we will use these diagrams, which are a mixture of state-chart and flow-chart, to describe processes.

The essential idea of our version of the DE MoC is that the simulations semantics holds a universal simulation time and a list of events ordered by time. The simulation semantics takes the first (i.e. earliest) event from the list, sets the simulation time to the time of the event, and calls the process of the event. Upon finishing (after possibly generating more events for the simulation kernel's event list), this process yields again to the simulations semantics, which then repeats this procedure with the next event from the list.

The processes in the list F correspond to `SC_THREAD`s in SystemC. Like `SC_THREAD`s, the processes in F are started at the beginning of the execution of the model, eventually freeze and unfreeze several times before ending, after which they are never invoked or called again. All other methods are called, and are not allowed to freeze. We sum this condition up in the following

Definition 3.9 (Discrete Event Models)**.** *A CSM $\mathcal{M} = (\mathcal{P}, \mathcal{V})$ is modeled in the Discrete Event (DE) MoC if*

1. *No $p \in \mathcal{P}$ invokes another process.*
2. *Each $p \in \mathcal{P}$ which can freeze (i.e. it uses freeze at least once) is never called by another $q \in \mathcal{P}$ but will be included in the list F of the DE execution semantics.*

While the idea of the DE-MoC is that processes invoke/call other processes by generating events that they pass to the *newEvent* process of the execution semantic, we do not forbid *calling* other processes directly if they don't freeze. If a process freezes itself, it can only be unfrozen by α_{DE} if

a corresponding event is contained in the event list E. For example, a process $p \in \mathcal{P}$ can suspend itself for a simulated timespan of $d \in \mathcal{T}$ by passing an event $(t_{DE}+d, p)$ to the $newEvent$ process before it freezes. In SystemC, this is achieved by calling the function wait(sc-time d).

A common form of communication used in the DE MoC is the *discrete event signal*:

Definition 3.10 (discrete event signal)**.** *A discrete event signal S consists of*

- *two variables s and s_{old},*
- *an access process $write_s$ with $In(write_s) = s$, $State^{EX}(write_s) = s_{old}$ and*
- *a constant list of processes $L = \{l_0, ..., l_{k-1}\}$*

For each discrete event signal s, only one process in a DE-model is allowed to write to $In(write_s) = s$ and call $write_s$. When $write_s$ is called and $s \neq s_{old}$ holds, it sets $s_{old} = s$, after which it calls $newEvent(t_{DE}, p)$ for each process $p \in L$. After that it ends.

That is, the processes L are the processes which listen to the value changes of s, and if a process writes on s and signals that by calling $write_s$, events are generated for each process P_s and passed to the execution semantic processes (see also Figure 3.5). Definition 3.10 corresponds to the `sc_signal<>` in SystemC. If we remove the test for $s \neq s_{old}$ in Definition 3.10, we get a definition corresponding to the SystemC `sc_buffer<>`.

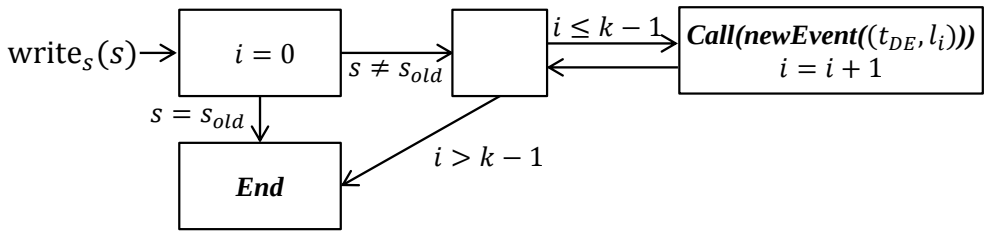


Figure 3.5: The access process of a discrete event signal

Note that we didn't define a dedicated $read_s$ method; the process which is called by the event generated by the write simply reads s (and other variables it is interested in).

3.3 Process Networks

In this work, the term Process Networks (PN) is used to denote the family of MoCs stemming from processes communicating via FIFOs. A well-known form is the *Kahn Process Network (KPN)* [Kah74], where *concurrent* processes are connected with unbounded FIFOs, writing and reading so-called *tokens*, unspecific generic data elements. When a process reads from an empty FIFO, it is blocked (i.e frozen) until another process writes to the FIFO. We describe the PN-style MoCs in our processes formalism after quickly fixing the following

Definition 3.11 (FIFO). *An **FIFO** F of size n_F ($n_F \in \mathbb{N}_{>0} \cup \infty$) is a component which consists of*

- *a list of n_F state variables $S = \{s_0, ..., s_{n_F-1}\}$,*
- *an input variable in_F,*
- *an output variable out_F and*
- *two state variables $l_F \in \mathbb{N}$ and $b_F \in \{0, 1\}$.*

*Each FIFO is assigned two **access processes**, a write process $push_F$ and a read process pop_F, such that*

- *l_F and b_F are initially set to 0 and are only accessed by $push_F$ and pop_F.*
- *If $push_F$ is called then (see also Figure 3.6)*
  1. *if $l_F = n_F$ then set $b_F = 1$ and Freeze.*
  2. *set $s_{l_F} = s_{l_F-1}, s_{l_F-1} = s_{l_F-2}, \ldots s_1 = s_0, s_0 = in_F$ and $l_F = l_F + 1$.*
  3. *if $b_F = 1$, set $b_F = 0$ and Invoke(pop_F).*
  4. *End.*
- *If pop_F is called then (see also Figure 3.7)*
  1. *if $l_F = 0$ then set $b_F = 1$ and Freeze.*
  2. *set $out_F = s_{l_F-1}$ and $l_F = l_F - 1$.*
  3. *if $b_F = 1$, set $b_F = 0$ and Invoke($push_F$).*
  4. *End.*

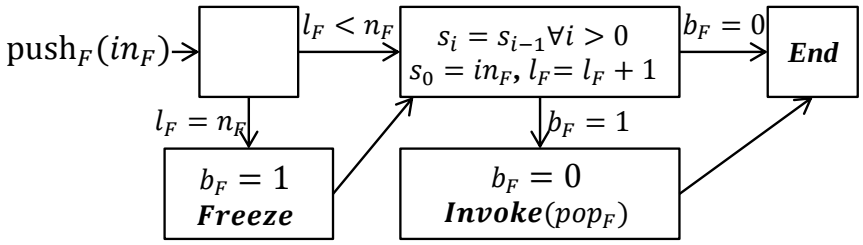


Figure 3.6: The $push_F$ access process

That is, if a process p calls pop_F on an empty FIFO F, it is frozen by the freeze of pop_F. If another process q later calls $push_F$, $push_F$ will invoke (and therefore unfreeze) pop_F which will unfreeze p by ending. According to Definition 3.1 $push_F$ will be unfrozen after p freezes again or ends.

If p calls $push_F$ on a full FIFO F, it will be unfrozen in a similar manner by a subsequent process q call pop_F. Note that we don't allow for PN processes to test if FIFOs are empty or full (i.e. non-blocking access). When we come to conversion between MoCs in Chapter 4, however, we might allow for converting processes to use such tests.

The SystemC implementation of `sc_fifo<T>` is conceptually the same as in definition 3.11. However, the unblocking of the complementary access process is handled via an event and immediate notification. With the Invoke mechanism in our formalism, we don't need to employ a simulation semantics process for this.

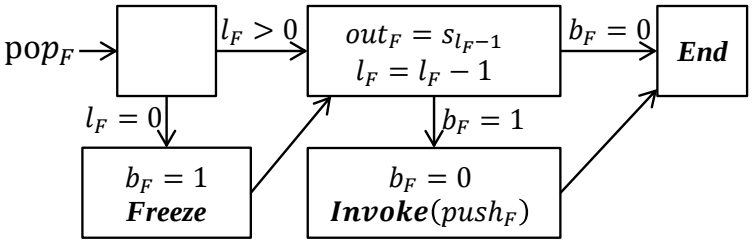


Figure 3.7: The pop_F access process

Definition 3.12 (Process Network MoC)**.** *A CSM $\mathcal{M} = (\mathcal{P}, \mathcal{V})$ is modeled in the Process Network (PN) MoC if for each process $p \in \mathcal{P}$ which is not an access process of a FIFO all of the following holds:*

1. *p only calls access processes of FIFOs. It does not invoke processes.*
2. *In(p) consists only of output variables of FIFOs, Out(p) consists only of input variables of FIFOs.*
3. *For every FIFO $F \in \mathcal{M}$, at most one process $p \in \mathcal{P}$ exists with $in_F \in Out(p)$ that calls $push_F$.*
4. *For every FIFO $F \in \mathcal{M}$, at most one process $p \in \mathcal{P}$ exists with $out_F \in In(p)$ that calls pop_F.*

Note that it is possible that the FIFOs in a PN-model carry initial values.

Definition 3.13 (Kahn Process Network MoC)**.** *The Kahn Process Network (KPN) MoC is a PN-MoC where each FIFO has infinite size.*

In particular, write access is always non-blocking in a KPN model.

Definition 3.14 (Bounded Kahn Process Network MoC)**.** *The Bounded Kahn Process Network (B-KPN) MoC is a PN-MoC where each FIFO has finite size.*

To execute a PN model, the starting process α_{PN} has to execute a schedule on the processes in the model (by invocation in order to not get frozen by a blocking FIFO access). For KPN it is known that the order of execution of the processes does not affect the outcome of the computation. Therefore, α_{PN} could simply invoke processes randomly in an infinite loop; of course, this would not be efficient for practical purposes. Some caution at least is needed for KPN models in case that there are processes in the model which run infinite loops to act as an infinite input streams to the rest of the model. Such a process p might be started while there is no other process frozen from a blocking read that would be unfrozen by the data token p produces. For example, this is the case right at the start of the execution of the model. If p does not freeze, the other processes will never get executed.

This situation could be remedied by imposing a rule on processes that forbids writing infinite token between invocation and freeze or end. However, we will not be concerned about scheduling of process network processes, as there is extensive literature on this (e.g. see [PPP95]). We will assume α_{PN} to provide a valid schedule for the model at hand if such a schedule exists.

While process networks are traditionally considered as an untimed MoC, the original paper by Kahn [Kah74] wasn't concerned about timing; the focus was the communication of processes via FIFOs. In fact, it is perfectly viable to consider processes of a process network operating in a discrete event fashion. This gives rise to the following

Definition 3.15 (Discrete Event Process Network MoC (DE-PN))**.** *A CSM $\mathcal{M} = (\mathcal{P}, \mathcal{V})$ is modeled in the Discrete Event Process Network (DE-PN) MoC if each process $p \in \mathcal{P}$ which is not an access process of a FIFO fulfills the conditions of Definition 3.9 and Definition 3.12. The execution of a DE-PN is governed by a DE execution semantics.*

3.4 Synchronous and Timed Data Flow

The Synchronous Data Flow Model of Computation is a special case of a PN MoC. The restriction here is that every time a process executes, it writes and/or reads a fixed finite number of token to/from each FIFO it accesses. We define the *data rate* of p with respect to F as follows:

$$r_F^p = \begin{cases} n & \text{if } p \text{ writes } n \text{ token to } F \text{ when executed} \\ -n & \text{if } p \text{ reads } n \text{ token from } F \text{ when executed} \\ 0 & \text{if } p \text{ does not access } F \end{cases}$$

For simplicity, we discard here the possibility that a process reads *and* writes to the same FIFO. If a process is in need for a FIFO for it's internal operation the FIFO can just be implemented as a state variable. Also, we are only interested in SDF models where there is a finite schedule for the processes such that all FIFOs can be chosen finite. If there is a process which reads and writes from/to the same FIFO, this is clearly only possible if it reads the same amount of token as it writes, such that the amount of token in the FIFO after each execution of such a process would be always 0, which would justify setting $r_F^p = 0$.

Definition 3.16 (Synchronous Data Flow MoC)**.** *A CSM $\mathcal{M} = (\mathcal{P}, \mathcal{V})$ is modeled in the Synchronous Data Flow (SDF) MoC if it fulfills all the conditions of Definition 3.12, as well as the following restrictions on the processes $p \in \mathcal{P}$ which are not an access processes of some FIFO:*

- *p does not freeze itself.*
- *For every FIFO F, $-\infty < r_F^p < \infty$.*

A connected component of processes $\mathcal{P}' \subset \mathcal{P}$ is called an SDF-cluster*.*

If the data rates of an SDF-model $\mathcal{M}$ are consistent, then a finite schedule S for the processes in $\mathcal{M}$ can be found that can be repeated indefinitely such that all the FIFOs in $\mathcal{M}$ can be chosen of finite size. To specify this formally, we define the *topology matrix* $A_M = (a_{i,j})$ for $\mathcal{M}$ where the rows are indexed by the FIFOs in M and the columns are indexed by the processes which are not FIFO access processes with entries $a_{F,p} = r_F^p$.

Theorem 3.1. *Let A_M be the topology matrix of an SDF model $\mathcal{M} = (\mathcal{P}, \mathcal{V})$. If there is a vector $v \in \mathbb{N}^{|\mathcal{P}|}$ with $A_M \cdot v^T = \mathbf{0}$ then a finite schedule S for the processes in $\mathcal{M}$ exists which allows for finite-sized FIFOs in $\mathcal{M}$. v is called a* repetition vector*.*

Due to Theorem 3.1, the execution semantic α_{SDF} for an SDF-model $\mathcal{M}$ is very simple: It just executes a static schedule of the processes in $\mathcal{M}$, repeatedly for a given number of times. This makes this MoC very favorable to use, since after the initial analysis to determine a static schedule, there is virtually no simulation overhead. Also, the access processes of the FIFOs can be simplified since there is no need for blocking due to an empty or a full FIFO any more.

There is a certain disadvantage in not having the notion of time available in SDF models. However, due to the static nature of the SDF MoC, this can be easily remedied by associating the token production/consumption of a process (which is always fixed) to a certain fixed time period.

Definition 3.17 (Timed Data Flow MoC)**.** *The Timed Data Flow (TDF) MoC is a SDF-MoC where each process p which is not a FIFO-access process has a time period $t_p \in \mathbb{R}$ associated to its execution, i.e. after each execution of p, a time span of t_p has passed. For each FIFO F with $r_F^p \neq 0$ a time span of $t_F^p := t_p / \left|r_F^p\right|$ is passing with each consumption or production of one token.*

Every process p holds a time variable t_{TDF}^p for the current simulation time which is initialized with 0. At the end of each execution, p updates it to $t_{TDF}^p = t_{TDF}^p + t_p$.

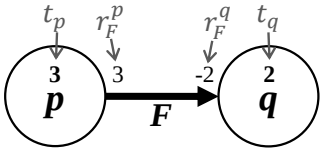
The introduction of time periods for each process gives rise to an additional consistency issue. Theorem 3.1 implies that the data rates r_F^p within an SDF-model have to be consistent such that no token accumulate in certain FIFOs in order to allow for a static finite schedule. For a TDF model, also the different time periods t_p for all $p \in \mathcal{P}$ have to be consistent.

More precisely: Let $\mathcal{M} = (\mathcal{P}, \mathcal{V})$ be a TDF model and $\mathcal{P}' \subset \mathcal{P}$ a TDF-cluster (which is just corresponding to an SDF-cluster). Consider a repetition vector $v = (v_0, v_1, ..., v_n)$ for the processes $\mathcal{P}' = \{p_0, p_1, ..., p_n\}$ in $\mathcal{M}$. Then $v_i t_{p_i}$ must be the same value for each $i = 0, ..., n$, and this value $t_{\mathcal{P}'}^c := v_0 t_{p_0} = ... = v_n t_{p_n}$ is called the *cluster period* of $\mathcal{P}'$.

In Definition 3.17 each process keeps track by itself of the simulation time. Alternatively, we could have defined a global simulation time variable which is updated by the execution semantic α_{TDF} (which behaves exactly like α_{SDF}) before executing a process. In fact, the SystemC AMS extension have implemented a mechanism like that. However, using a time variable which is maintained locally highlights an important difference between the time semantics of the TDF MoC and the DE MoC. In the DE MoC, there is a *global* time variable t_{DE} which either stays the same (during a δ-step) or increases in between execution of processes. If there was a global time variable t_{TDF} which holds a copy of the local t_{TDF}^p time variable of the currently executed process p, the values of t_{TDF} would in general also *decrease* in the course of the execution of the model.

To illustrate this consider for example a small TDF cluster with two processes p and q connected by a FIFO F with $t_p = 3$, $r_F^p = 3$, $t_q = 2$ and $r_F^q = -2$. A possible schedule for this cluster is p, p, q, q, q, and the values of the local time variable (i.e. t_{TDF}^p and t_{TDF}^q, respectively) at the different process executions would be $0, 3, 0, 2, 4$ for the first schedule execution, $6, 9, 6, 8, 10$ for the next schedule execution and so on (see Figure 3.8).

This simple example demonstrates that TDF exhibits a form of temporal *decoupling*, similar to the TLM 2.0 loosely timed coding style. Moreover, there is also a certain temporal dissociation *within* a TDF process p: If, for example, p reads several token $d_1, d_2, d_3, ..., d_n$ from one FIFO F when executed, the time stamps of these token (i.e. the start of the time span associated with the token's consumption) are $t_{TDF}^p, t_{TDF}^p + t_F^p, t_{TDF}^p + 2t_F^p, ..., t_{TDF}^p + nt_F^p$, respectively. If p processes these token in a context where the time when a token is valid matters (e.g. it multiplies the token with the current time), it has to take these offsets relative to t_{TDF}^p into account.



Schedule		p	p	q	q	q	p	p	q	q	q
local time at execution	t^p_{TDF}	0	3				6	9			
	t^q_{TDF}			0	2	4			6	8	10

Figure 3.8: Example TDF schedule and local times.

There also can be causality violations in the sense that output token might depend on input token with a later time stamp. Consider for example a process p with $t_p = 10$ which reads from a FIFO F with rate $r^p_F = -10$ and writes to a FIFO G with $r^p_G = 2$. The two token it writes have values depending on all of the 10 input token, for example their maximum and minimum, respectively. If $t^p_{TDF} = 0$ when p executes, the time span associated to the first output token lasts from 0 to $t^p_G = t_p/r^p_G = 5$, but the value of that token depends on input token with a later time stamp.

It would be possible to make a Definition for the TDF MoC which is more strict than Definition 3.17 in that it would demand that when a process p executes, token are read and written in an order corresponding to their timestamps, with an appropriate update of t^p_{TDF} in between. But since a process will in general read and write from/to different FIFOs with different data rates, this would be overly complicated and impractical to implement in practice. In the SystemC AMS extensions, it is the responsibility of the user to handle these issues correctly, or at least in a way which is correct enough for the problem at hand.

Regarding the connection of TDF models to models using different MoCs, the temporal looseness of the TDF MoC has to be handled with care, but also presents some opportunities as we will see in Section 4.5.

3.5 Transaction Level Modeling

In this section, we define a formal framework for TLM as a MoC. TLM (especially SystemC TLM2) was developed with many practical considerations in mind (like simulation speed, coding effort, and interoperability). The challenge is now to isolate and abstract the elements of TLM which are essential to the idea of Transaction Level Modeling.

3.5.1 Transactions

We start with the transaction itself. In SystemC TLM2, this is mainly a command (read or write), an address, and data. Since only the pointer to the data is passed with the TLM2 generic payload, the data size has to be passed as well, but in principle this is information inherent to the data. The elements byte enable and streaming width are already needed only in special cases (more selective access, modeling of bus lanes, access of buffers); this is basically additional information for the interpretation of the data. The response status can be regarded as a state of the transaction.

The remaining elements of the generic payload will be disregarded for our considerations. The DMI hint is part of mechanism to circumvent the transaction mechanism altogether for even faster simulation, so it does not really add to the concept of a transaction. However, with respect to [BAGK07], the DMI hint can be considered as meta-data.

The extension pointers are a tricky mechanism to add arbitrary additional information to a transaction, like additional commands, transaction states or meta-data. Therefore extensions don't really add anything new, with the exception of potential additional meta-data. The `tlm_gp_option` element was added for compatibility reasons and indicates how to interpret the transaction, so this is also information on how to interpret data.

There are other important elements of the transaction mechanism which are not part of the TLM2 generic payload itself, but are passed as separate elements in the TLM2 method calls: the delay and the phase (the latter only in the approximately-timed coding style). The phase is again a state of the transaction. The delay is essentially a time stamp, as it describes the time offset relative to the SystemC simulation time. If an absolute time stamp would be used instead, most coding efforts would become more complicated; for example targets which just wait for the delay before processing the transaction would have to compute the difference between simulation time and time stamp first. Nevertheless, the information is the same since in the SystemC setting, there is always a global simulation time.

After these considerations, let's collect the essence:

- the **command**: This is essentially read or write, but other commands are possible.
- the **data**: this would often be just an array, but in principle any kind of data is conceivable since there is also
- the **data descriptor**: In the TLM2 generic payload, this includes address, data length, streaming width and the byte enable information. This is all information to instruct the recipient of the transaction how to interpret the data. The command could also be viewed as a data descriptor, but since the command has a certain prominent role, and is also independent of the other descriptors, we choose to regard it as separate entity.
- the **time stamp**
- the **state**: In TLM2 response status and phase, as well as the `tlm_sync_enum`} return type of the `nb_transport_{fw|bw}` methods.
- the **meta-data**: This captures all additional information of a transaction which is not related to the data directly. Examples are meta-information attached to a generic payload with an ignorable extension relevant to only one system component, but also what in [BAGK07] is called *performance data*.

We summarize this in the following

Definition 3.18 (transaction)**.** *A* transaction *is a variable* $T = (c_T, d_T, a_T, t_T, s_T, m_T) \in \mathcal{C} \times \mathcal{D} \times \mathcal{A} \times \mathcal{T} \times \mathcal{S} \times \mathcal{M}$ *where*

- $\mathcal{C}$ *is the set of commands*
- $\mathcal{D}$ *is the data-space*
- $\mathcal{A}$ *is the descriptor-space*
- $\mathcal{T}$ *is the time-base*

- $\mathcal{S}$ *is the set of states*
- $\mathcal{M}$ *is the meta-data-space*

of the transaction.

As an example, in the case of the TLM2 generic payload, leaving `tlm_gp_option` and the extension mechanism aside, this would look like this:

- $\mathcal{C} = \{$`TLM_IGNORE_COMMAND`, `TLM_READ_COMMAND`, `TLM_WRITE_COMMAND`$\}$
- $\mathcal{D} = \mathbb{N}$ (where values are interpreted as byte-vectors)
- $\mathcal{A} = \{0, \ldots, 2^{64} - 1\} \times \{0, \ldots, 2^{32} - 1\} \times \{0, \ldots, 2^{32} - 1\} \times \mathbb{N}$ (address, data length, streaming width, and byte enable respectively, the latter interpreted as byte-vector of a size as indicated by the data length)
- $\mathcal{T} = \{0, \ldots, 2^{64} - 1\}$ (as an `sc_core::sc_time` value is essentially an unsigned 64-bit integer which is interpreted relative to the time resolution)
- $\mathcal{S =}$ `tlm_response_status` $\times$ `tlm_phase`$\times$`tlm_sync_enum`
- $\mathcal{M} = \{true, false\}$

In this thesis, we will use this simplified version of a transaction:

- $\mathcal{C} = \{read, write\}$
- $\mathcal{D} = \{$token-vectors of varying size $n\}$ (the type of the token does not matter for our purposes)
- $\mathcal{A} = \mathbb{N}$ (the address)
- $\mathcal{T} = \mathbb{R}$
- $\mathcal{S}$ = {INCOMPLETE, ERROR, OK} $\times$ {begin_req, end_req, begin_resp, end_resp} $\times$ {unchanged, updated, completed}
- $\mathcal{M} = \varnothing$

$\mathcal{S}$ is essentially the same state-space as that of the TLM2 standard, except that we just use one generic ERROR state, and instead of `TLM_ACCEPTED`, "unchanged" is used. Therefore we refer to the three parts of the state as response, phase and sync-state, respectively. We could also choose to add the data length n to the data descriptor by using $\mathcal{A} = \mathbb{N} \times \mathbb{N}$. However, the size of the data vector will always be an indefinite n in our considerations, so this would just be redundant.

3.5.2 The TLM MoCs

We now define MoCs corresponding to the loosely-timed and the approximately-timed coding style, respectively. First, we capture what is common to both MoCs:

Definition 3.19 (General TLM MoC, interface processes, components)**.** *A Transaction Level Modeling MoC is governed by a DE execution semantics where DE processes communicate via passing references to transactions. There are three types of components: initiators, interconnects, and targets. We call a process belonging to such a component an initiator process, interconnect process, or target process, respectively. For each of these components, there are distinguished component processes, called* ***interface processes****:*

- ***Initiator interface processes*** *receives references to transactions that were generated by itself or another process of the same component.*
- ***Interconnect interface processes*** *receive references to transactions, possibly change parts of the referenced transactions, and pass them on to other processes.*
- ***Target interface processes*** *receive references to transactions and possibly change parts of the referenced transactions. These processes execute the transaction in the sense that they act in a certain way depending on the transaction parameters, especially the command.*

Interface processes obey to the following rules:

- *When a reference to a transaction T is passed to an interface process, $t_T \geq t_{DE}$ has to hold.*
- *Only interconnect or target interface processes can change the state to $(ERROR, ", ")$, only target interface processes can change the state to $(OK, ", ")$ (The '"' indicates that the other entries remain unchanged).*

We can now define our version of the two TLM coding styles:

Definition 3.20 (Loosely-Timed TLM MoC (LT-TLM))**.** *The Loosely-Timed TLM MoC is a TLM MoC with the following additional rules:*

1. *There are no initiator interface processes.*
2. *Interface processes are only started with CopyCall().*
3. *Only a reduces state space $\mathcal{S} = \{INCOMPLETE, ERROR, OK\}$ is used.*

That is, transactions are passed in a very simple way: initiator processes (i.e. processes belonging to an initiator component) generate them and pass them by calling interface processes, either of interconnect components or target components. If they arrive at a target interface process, they will be processed, possibly resulting the target interface process to freeze, after calling *newEvent* to schedule a later *Invoke* by α_{DE}. In this case the initiator process also freezes. Later, α_{DE} will invoke the target interface process which eventually ends, such that the initiator process can continue to inspect the result of the transaction. Figure 3.9 shows an example where a target interface process freezes before ending.

For simplicity, no interconnect interface process is shown in Figure 3.9. However, they behave in essence like a target interface process in that they might freeze and block the initiator process.

The reason why interface processes in the LT-TLM MoC are only executed with *CopyCall* was already explained at the beginning of this chapter, shortly after definition 3.1[2]: While an interface

[2]This actually motivated the Definition of *CopyCall* and *CopyInvoke*

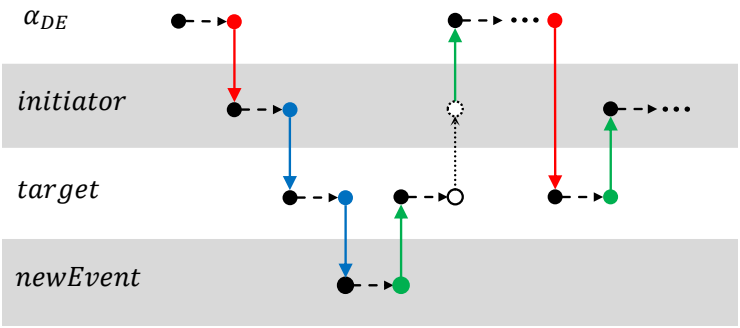


Figure 3.9: Example process sequence in a LT-TLM model

process is frozen, there might be other transactions arriving. These are now handled by a copy of the interface process in question.

If an interface process generally freezes if the timestamp t_T of a transaction T is larger than t_{DE}, such that is invoked by α_{DE} when $t_{DE} = t_T$, this even sorts transactions arriving out-of-order, since a frozen process copy processing the transaction with the smallest timestamp is always invoked first.

We give no formalism or define facilities for temporal decoupling, e.g. unlike [SKR11] where temporal decoupling based on a quantum keeper is part of the author's definition of loosely timed models. While the TLM↔TDF MoC converters in section 4.5 have to deal with transactions which might have been issued by temporally decoupled initiators, the converters themselves don't employ temporal decoupling. However, if an initiator component wants to employ temporal decoupling, it would simply hold a state variable with its local time offset, which its processes would increase accordingly instead of calling *newEvent*. This is straightforward and works like it is described in the SystemC standard [IEE05].

Definition 3.21 (Approximately-Timed TLM MoC (AT-TLM))**.** *The Approximately-Timed TLM MoC is a TLM MoC with the following additional rules:*

1. *Interface processes are only started with CopyInvoke().*
2. *Only initiator processes can change the state of a transaction to* (", *begin_req*,") *or* (", *end_resp*,")
3. *Only target processes can change the state of a transaction to* (", *end_req*,") *or* (", *begin_resp*,")
4. *During the lifetime of a transaction, its state can only change in the order* (", *begin_req*,") → (", *end_req*,") → (", *begin_resp*,") → (", *end_resp*,")*, where omitting* (", *end_req*,") *and/or* (", *begin_resp*,") *is allowed.*

The last rule in definition 3.21 represents a simpler version of the TLM2 phase rules as depicted in Figure 2.5. For our MoC conversion efforts, this will suffice. But in general definition 3.21 can be easily extended with additional phase rules, which might also refer to an extended set of phases.

It might come as a surprise that we don't forbid interface processes to freeze, i.e. like the rule in TLM2 that the non-blocking interface methods are not allowed to call `wait()`. The reason

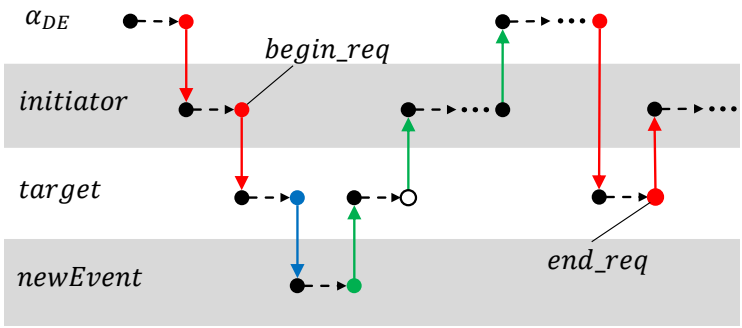


Figure 3.10: Example process sequence in an AT-TLM model, showing the first two transaction phases

is that with the *Invoke*() operation, our formalism provides an alternative approach to achieve non-blocking access.

Figure 3.10 shows an example with the first two phases of our protocol: After an initiator process is invoked, it generates a transaction with phase *begin_req* and calls a target process. Here, the target process can't begin the next phase directly, so it calls *newEvent*, such that it can be invoked later, and then freezes. This continues the initiator process, which eventually yields to α_{DE}. In turn, α_{DE} will invoke the target process, which now can sends the transaction reference to the initiator with an updated phase, in this case *end_req*. Note that the initiator interface process invoked here is in general not the initiator process which initiated the transaction (although it could), but of course it has to be a process of the same initiator component according to the second rule of definition 3.21.

The advantage of this alternative non-blocking access is that we don't have to deal with facilities like *payload event queues*. In the TLM2 approximately-timed coding style, these are common tools used to schedule the continuation of transaction processing at a later point in simulation time without calling `wait()`. They store the transaction reference together with time stamp and a reference to a callback function in a queue which is sorted according to the time stamp. Using an event, they then call the callback function (with the transaction reference as the parameter) at the time given by the time stamp. Then callback function then proceeds to process the transaction.

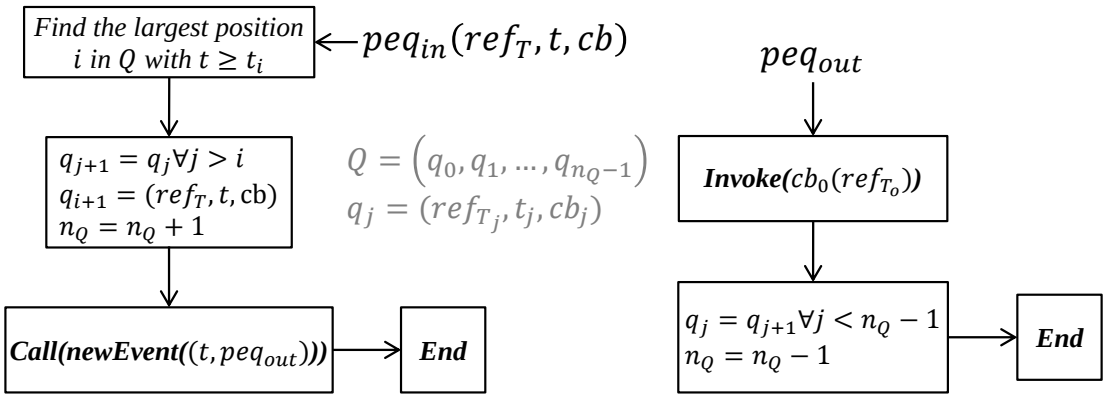


Figure 3.11: Example operation of a payload event queue

Figure 3.11 shows how this could look in our formalism: The peq_{in} process takes transaction reference ref_T, time stamp t and a callback process reference cb as an input, inserts them in the

right place within the queue Q, and then schedules a new event for the process peq_{out} for time t.

When the peq_{out} process is executed, it invokes the callback process of q_0, after which it removes q_0. Since there is exactly one pending (peq_{out}, t) event per queue entry, Q is never empty when called. And since the (peq_{out}, t) events are processed in the same order as the corresponding entries in Q are ordered, q_0 will always be the right entry; in particular $t_0 = t_{DE}$ will always hold.

Therefore we could also use a definition of the AT-TLM MoC which is closer to TLM2 in that we would not allow interface processes to freeze, and employ payload event queues to control transaction processing. While we won't do so, we have to employ techniques similar to the payload event queue for the converters between TLM and TDF models, which will be introduced within the next chapter.

4 Connecting System Models described with different MoCs

We already discussed the main reasons for the need to connect models using different MoCs in Chapter 1: The system of interest might consist of subsystems inherently belonging to different implementation domains, e.g. in a mixed analog/digital system. Also, different subsystems might use different abstraction levels. Another reason might be the desire to (re-)use existing models.

The difficulty of such a connection depends on the MoCs used. In some cases, the conversion semantics are obvious and natural, while in other cases, the semantics might depend on the modeling goals, i.e. are driven by the use case. Certain combinations of MoCs used will make information loss unavoidable in certain cases. And even if the conversion semantics are well-defined in normal conditions, there might be the need to handle certain corner-cases where the conversion semantics are not that obvious.

In this Chapter we analyze several MoC conversion problems in our formalism. For some of them implementations are available, like the SystemC AMS converter ports, which connect TDF modules to DE signals. Some are new, like the conversions between TDF and TLM. In the following, we first make for each pair of MoCs considered general considerations, like in how far the two executions semantics have to be synchronized, after which we look at the two different conversion directions, as this in general makes a substantial difference regarding the conversion semantics. Note the converter processes discussed will in general "bend" the rules of the MoCs involved in some way, e.g. by accessing data like the number of token in a FIFO, which is normally hidden to the outside.

4.1 Discrete Event models and TDF models

We start with the conversion problems between DE models and TDF models since they are already covered in SystemC-AMS, as outlined in Section 2.2. The general problem to solve is to allow TDF-processes to generate and/or react to events; however we mainly look at the harder problem of writing and/or reading from/to discrete event signals step-by-step. We first deduce the minimum conversion setup for a TDF cluster only *writing* to DE-signals, then have a look what we need for reading from DE signals, before putting it together in Section 4.1.3.

4.1.1 TDF writer

The most easy and straightforward case is when processes in a TDF cluster only *generate* events. Every time a TDF process p fires, it can generate events with time $t^p_{TDF} + d$ by calling the *newEvent* process of the DE execution semantics, where $d \geq 0$ denotes a delay. Since the TDF cluster is a pure event source for the DE model, we can even execute the whole cluster first for a certain desired time-span, and afterwards start the DE model with the event queue E of α_{DE} already filled with events from the TDF cluster. The events subsequently generated by the DE model will be sorted adequately into the event queue. Therefore, in this simple case there is no need for any additional synchronization between the TDF execution semantics and the DE execution semantics.

Of course this does not allow for the transmission of information yet, since no data is passed with an event per se. Suppose the TDF process p wants to write to a DE signal S as defined in 3.10. If we still want to execute the TDF cluster before we start the DE model execution, we need to store the values to be written to S during the execution of the TDF model, and then write them to S at the right times when the DE model is executed. The following converter achieves this:

Definition 4.1 (TDF→DE converter)**.** *A* TDF→DE converter *consists of*

- *an unbounded FIFO F*
- *a process $conv_S(v,t)$ which takes a value v and a time-value t as arguments*
- *and a process $exec\_conv_S$*

If a TDF process p wants to write a value v to a DE signal S, it calls $conv_S(v, t^p_{TDF})$. $conv_S$ pushes v to the FIFO F, and schedules $exec\_conv_S$ to be executed by a_{DE} at time t^p_{TDF} by calling $newEvent((t^p_{TDF}, exec\_conv_S))$. If $exec\_conv_S$ is executed at $t_{DE} = t^p_{TDF}$, it removes v from the FIFO and writes it to the Signal S. See also Figure 4.1.

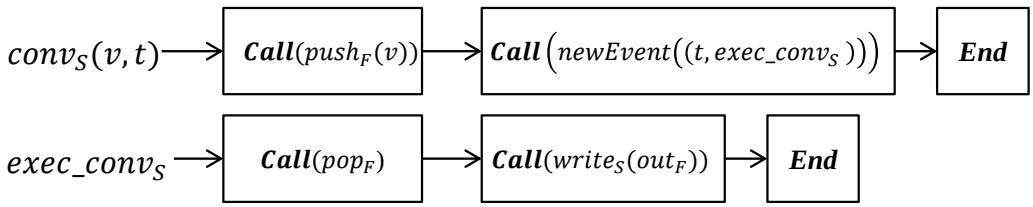


Figure 4.1: TDF to DE converter

In SystemC AMS, the converter ports `sca_tdf::sca_de::sca_out<>` implement a mechanism similar to the one described in Definition 4.1.

To be able to use *bounded* FIFOs is an important part of the concept of TDF (or SDF in general). In order to use bounded FIFOs for the TDF→DE converter as well, the TDF execution semantics and the DE execution semantics have to be synchronized. A simple way to achieve this is to let the TDF execution semantics yield to the DE execution semantics after each round of schedule execution. More precisely, we initialize α_{DE} such that $\alpha_{TDF} \in F$ (Where F is the "thread-list" invoked by α_{DE} at the start of the execution). α_{TDF} contains an infinite loop, and the first command in that loop is a call to $newEvent((t^p_{TDF}, \alpha_{TDF}))$, followed by a *Freeze*.

Here, p is the first process for α_{TDF} to call according to the static schedule which α_{TDF} executes. After unfreezing, α_{TDF} executes the static schedule once, and jumps back to the beginning of the loop. Now the cycle repeats with another call to $newEvent((t^p_{TDF}, \alpha_{TDF}))$, but as described in Definition 3.17, t^p_{TDF} now contains the correct time of the next execution of p.

Effectively, this causes α_{TDF} to be executed by α_{DE} at the times $t_{DE} = 0, t^c_{\mathcal{P}}, 2t^c_{\mathcal{P}}, ...$, where $t^c_{\mathcal{P}}$ is the cluster period of the processes $\mathcal{P}$ of the TDF cluster. We won't describe this strategy more formally, since it has an important drawback: While α_{TDF} is executed such that $t_{DE} = t^p_{TDF}$ for the first process p in the static schedule, it is unclear at what δ-step it is executed. It could be the first δ-step for the time $t_{DE} = t^p_{TDF}$, but it won't be in general. In the SystemC AMS PoC implementation, the SystemC AMS simulation kernel (which essentially corresponds to our α_{TDF}) is always executed during the *first* δ-step; we will see in Section 4.1.3 why this is the case.

Without loss of generality, we will assume from now on that α_{TDF} only executes *one* TDF cluster. If more than one TDF cluster is to be executed, we could either foresee a different α^c_{TDF} for each cluster c, or define a mechanism in α_{TDF} which checks (according to the current DE simulation time t_{DE}) which clusters to execute when α_{TDF} is executed.

4.1.2 TDF reader

In the previous section we saw that if the information only flows from a TDF cluster to a DE model, we can run the two models totally decoupled from each other, although for practical purposes (bounded FIFOs in the converters) some synchronization is advisable. If a TDF process has to read from a DE signal, however, we *need* to synchronize since the DE side has to write on the signal, and the TDF side needs to read it at the appropriate time before it gets overwritten. Moreover, we can't use the listener mechanism of the DE signal for the reading TDF process because the TDF process still needs to be executed in the right order within the static schedule.

Therefore, the TDF execution semantics α_{TDF} has to yield to α_{DE} each time it is about to execute a process which reads from a DE signal. To this end, α_{TDF} holds a list of all processes in the cluster which read from DE signals. If the next process p to execute from the static schedule is in this list, it calls $newEvent(t^p_{TDF}, \alpha_{TDF})$ and freezes in order to give α_{DE} the chance to invoke and/or call DE processes which might update the DE signal p reads from.

While this conversion direction is more complicated, there is an upside: There is no need for an additional converter as in the opposite case. If p is executed at the right time (w.r.t. t_{DE}), it simply reads from the signal variable. The conversion effort here lies in the synchronization between the two execution semantics. In SystemC AMS there is a corresponding converter port `sca_tdf::sca_de::sca_in<>`, but unlike the previous case it is basically a marker that a process reads an `sc_core::sc_signal<>`, and needs to be treated accordingly.

Yet, there is still a complication: When α_{TDF} is finally unfrozen by α_{DE}, there could be still events in the event list E with a time t^p_{TDF}, i.e. the corresponding processes will be executed in subsequent δ-steps after α_{TDF} was executed p. These processes, however, could still update the DE signals p reads from.

We now could make sure that α_{TDF} is executed in the last δ-step, e.g. by giving α_{TDF} access to the event list E. α_{TDF} would then check if there are still such events present and if so, it would yield again to α_{DE}, possibly several times until there are no more events with a time t^p_{TDF}, such that it now can execute p. Indeed, this is a viable, well defined strategy if the cluster *only reads* from DE signals.

In the next section, we will see that we need to execute α_{TDF} in the *first* δ-step though, and this means that a TDF process p which reads from a DE signal will always miss the signal updates happening at $t_{DE} = t^p_{TDF}$. However, a TDF process p reading from a DE signal will *in general* be in danger of missing signal updates, even if α_{DE} would be executed only at the last δ-step: Because in between two executions of p, α_{TDF} yields to α_{DE}, which in turn can trigger arbitrarily many updates of the DE signal. Therefore, this is a general *corner case* to handle even if we can manage to get all updates at time $t_{DE} = t^p_{TDF}$.

If it is a problem for the overall model when a TDF process p is missing the updates of a DE signal now depends on the use case. For example, p could represent an A/D converter, which samples an analog signal in fixed intervals. In this case, missing the signal updates in between is no real problem, since a real A/D converter also will miss the changes of the analog input signal in between samples. However, this is not a likely scenario since one reason of using TDF signals in addition to DE signals is to provide better options for the modeling of analog signals. That is, A DE signal would never be used to model an analog signal in the first place.

In realistic mixed DE-TDF models we will usually see that the DE signal S is a *control signal* to influence the TDF process p in some way (e.g. by changing some parameter like the amplification). And if p misses the updates of S that might indicate that something is not in order. For example, the DE process updating the signal might do this too frequently because a delay was set to low or a clock frequency was set too high.

This is a corner case which is not foreseen by SystemC AMS, and while there is no way to remedy it, we can at least detect it. Therefore the converter channels which will be introduced in Chapter 5 provide an option to detect such missed updates. In our formalism, we can implement this as follows: Instead of an ordinary DE signal we use a modified DE signal s with an additional counter variable u_s which is incremented every time the value of s is updated. u_s is initialized to 0 and is reset to 0 by p every time after reading from s. p can now inspect u_s before reading from s. If $u_s > 1$, there are missed updates and p can e.g. generate a warning.

4.1.3 TDF clusters which read from DE signals an write to DE signals

We now finalize the conversion between DE models and TDF models by considering the general case: A TDF cluster with processes which read from DE signals as well as processes which write to DE signals via TDF→DE converters as in Definition 4.1. This includes processes which do both, although we will see that this does not add additional complexity.

Theorem 4.1. *If a TDF cluster contains processes which read from DE signals and write to DE signals via a TDF→DE converter, respectively, α_{TDF} can't reliably be executed at the last δ-step of a given discrete time point t_{DE}.*

Proof. Consider a TDF process p reading from a DE signal s as in the previous section, and assume α_{TDF} is executed in the last δ-step of $t_{DE} = t^p_{TDF} = T$. Suppose further that the static schedule executed by α_{TDF} is such that there is another TDF process q which is executed before α_{TDF} yields to α_{DE} such that $t^q_{TDF} = T$ (for example p itself), and q writes to a DE signal S via a TDF→DE converter. That is, after α_{TDF} is started at there is a call $conv_S(v, T)$ as described in Definition 4.1.

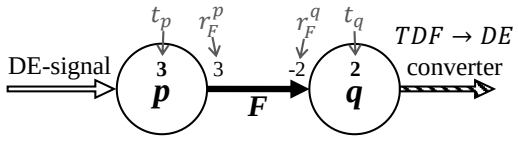
This generates a new event $(T, exec\_conv_S)$, i.e. the process $exec\_conv_S$ will be executed after α_{TDF} yields to α_{DE} in a subsequent δ-step of the time $t_{DE} = T$. Therefore, α_{TDF} wasn't executed at the last δ-step as assumed. q.e.d.

This means we have to execute α_{TDF} in the *first* δ-step of a given discrete time point t_{DE}, since this is the only other well-defined case. To achieve this, we can for example change α_{DE} such that every time it increases t_{DE}, it checks if there is an event (t_{DE}, α_{TDF}) somewhere in the event list E. If this is the case, it removes the event from E and invokes α_{TDF} before executing any other process.

However, there is still one issue remaining: In general, α_{TDF} will execute other TDF processes from the static schedule after executing p and before yielding to α_{DE}, and one of these processes (say q) might write to a DE signal via a TDF→DE converter. As we have seen in Section 3.4, it can now happen that t^q_{TDF} is *smaller* than the value of t^p_{TDF} when p was executed, which means $t^q_{TDF} < t_{DE}$ (see also Figure 4.2).

This obviously constitutes an error, as we can't schedule an event for a time smaller than the current time. In these cases, the process q has to write to the DE signal with a *delay* d such that $t^q_{TDF} + d \geq t_{DE}$. With the TDF→DE converter this can simply be achieved by calling $conv_S(v, t^q_{TDF} + d)$.

The minimum size of d depends on the static schedule for the TDF cluster. Figure 4.2 shows two different schedules for the same cluster. While in schedule 1, $t^q_{TDF} - t_{DE} \in \{-3, -1, 1\}$, in schedule 2 we have $t^q_{TDF} - t_{DE} \in \{-1, 1\}$.



Schedule 1		p	p	q	q	q	p	p	q	q	q
local time at execution	t^p_{TDF}	0	3				6	9			
	t^q_{TDF}			0	2	4			6	8	10
t_{DE}		0	3	3	3	3	6	9	9	9	9
Difference to t_{DE}		0	0	-3	-1	1	0	0	-3	-1	1
Schedule 2		p	q	p	q	q	p	q	p	q	q
local time at execution	t^p_{TDF}	0		3			6		9		
	t^q_{TDF}		0		2	4		6		8	10
t_{DE}		0	0	3	3	3	6	6	9	9	9
Difference to t_{DE}		0	0	0	-1	1	0	0	0	-1	1

Figure 4.2: A TDF cluster reading and writing to DE signals and two schedules for it.

It would be possible to use a different delay d every time, i.e. just large enough such that $t^q_{TDF} + d = t_{DE}$if t^q_{TDF} is not larger than t_{DE} already, in which case $d = 0$ would be set. But in the interest of a sound, consistent model, the delay should be constant, i.e. $d = 3$ for schedule 1, and $d = 1$ for schedule 2. And in the interest of precision this constant delay should be as small as possible, i.e. among the possible static schedules for a TDF cluster, the one which yields the smallest delay values necessary should be chosen.

We won't discuss here how to compute such a schedule, the important observation here is that a it is needed in general. Our converter in definition 4.1 does not to be changed for this, as the TDF process using the converter can just add the delay on its own by calling $conv_S(v, t^p_{TDF} + d)$. The converter ports in SystemC AMS offer a method `set_delay()` where this can be done explicitly. The SystemC simulation kernel also detects during the simulation if the SystemC AMS time is smaller than the SystemC time, in which case it throws an error and also suggest a minimum delay value for the culprit converter port.

4.2 Untimed Process Networks and SDF/TDF models

We will now look at the problem to connect a PN model with a SDF model. This conversion promises to be easy to handle since both MoCs are based on processes communicating via FIFOs. The main difference is that SDF processes never freeze, since the schedule and the size of the FIFOs are chosen accordingly. Now when an SDF process accesses a FIFO which connects to a PN process, it can happen that it freezes, e.g. when reading from an empty FIFO that is written to by a PN process. This has to be handled. However, it turns out that in our formalism, the PN-SDF connection comes almost for free.

We describe a simple conversion strategy to connect a (bounded or unbounded) PN model $\mathcal{M}_{\mathcal{PN}}$ to an SDF model $\mathcal{M}_{\mathcal{SDF}}$. The idea is that α_{PN} invokes α_{SDF} first before invoking the processes in $\mathcal{M}_{\mathcal{PN}}$. Now some of the processes in $\mathcal{M}_{\mathcal{SDF}}$ are connected to PN processes via FIFOs, and if a such a TDF process p_{SDF} accesses one of these FIFOs F connecting to a PN process p_{PN}, it will eventually freeze. Since α_{SDF} only *calls* the TDF processes, it is then also frozen which in turn continues α_{PN}.

Now α_{PN} will at some point invoke the PN process p_{PN}, whose activity will eventually (i.e. after several invokes or after several freezes and unfreezes) fill F enough to unfreeze p_{SDF}. p_{SDF} now can end eventually, which continues α_{SDF}. α_{SDF} continues to execute the static schedule until it freezes again because of some other TDF process freezing due to access of a FIFO which connects to a PN process. This now continues p_{PN} and, at some point α_{PN}. That way, α_{PN} and α_{SDF} are running alternately (see Figure 4.3).

Of course it can also happen that a PN process p_{PN} freezes because of accessing a FIFO connecting to an SDF process (for example when attempt to read from an empty FIFO). But this means that α_{SDF} is frozen, which implies that some SDF process p_{SDF} is frozen. So as above, by continuing to execute PN processes, p_{SDF} will be unfrozen, which unfreezes α_{SDF} and eventually p_{PN}.

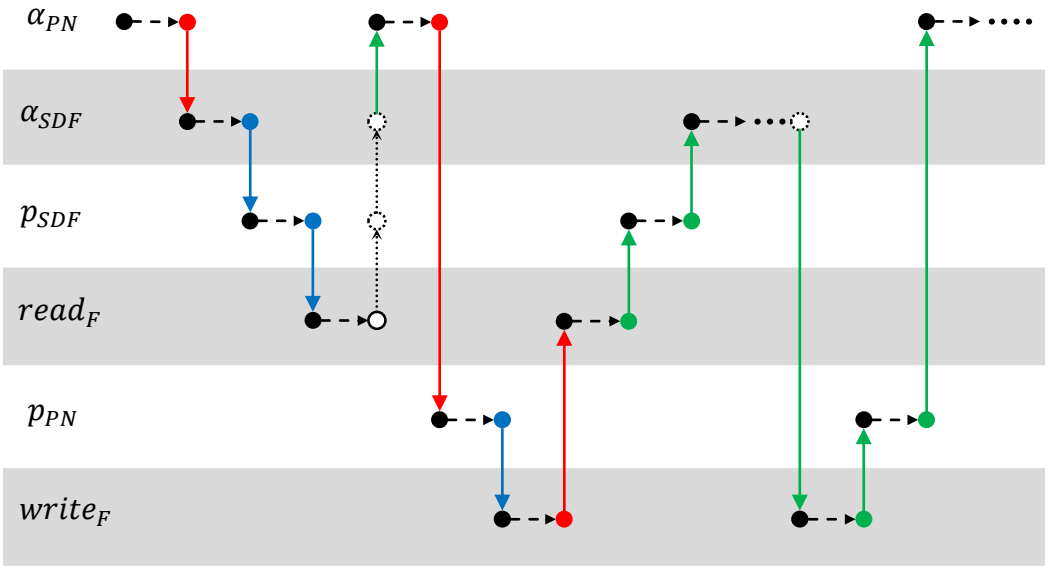


Figure 4.3: Alternating execution of α_{PN} and α_{SDF} when executing a combined PN-SDF model

For this strategy to work, the connecting FIFOs between writing SDF processes and reading PN processes have to be finite; otherwise it could happen that α_{SDF} never freezes, e.g. when

the SDF cluster is a pure token producer for the PN model. In this case, if the FIFOs to the PN processes are all unbounded, the SDF cluster could just keep on going as no SDF process would ever (indirectly) freeze. But this situation can occur also in PN models in general with PN-processes which are pure token producers. If α_{PN} has an appropriate scheduling strategy for this situation, the connection to an SDF cluster poses no further challenges.

The above remarks prove the following

Theorem 4.2. *Let $\mathcal{M}_{\mathcal{PN}}$ be a bounded or unbounded PN model which connects to an SDF model $\mathcal{M}_{\mathcal{SDF}}$ with an execution semantic α_{SDF}. Then α_{PN} can schedule the execution of $\mathcal{M}_{\mathcal{SDF}}$ together with the execution of $\mathcal{M}_{\mathcal{PN}}$ by treating α_{SDF} like a PN process.*

The exact same strategy can now be used to connect Process Networks to TDF models, since there is no time adoption necessary as the PN network is untimed. Interestingly, this MoC conversion could entirely treated globally by invoking α_{SDF} from α_{PN}; there was no need for any kind of local converter in between a PN process and a SDF process apart from demanding a bounded FIFO. When the PN network is timed, however, this is not the case.

4.3 Discrete Event Process Networks and TDF models

According to Definition 3.15, a DE-PN model is basically a DE model where processes communicate via FIFOs. Since FIFOs are the only means of communication, every process has to be in the list F of the DE execution semantics to be invoked by α_{DE}.

We will see that conversion between DE-PN models and TDF models is mainly a local problem handled by TDF processes. Therefore we could also allow for non-blocking (i.e. non-freezing) access of DE-PN processes to FIFOs, e.g. by allowing them to access n_F and l_F, as this doesn't change the conversion problem.

4.3.1 TDF reader

When a TDF process p_{TDF} reads from a FIFO F_{DE} written by a DE process p_{DE} , it can in general happen that the FIFO contains not enough token. In the previous Section we had p_{TDF} freeze itself, effectively yielding control to α_{PN} which would eventually result in the FIFO to be filled and and p_{TDF} to be unfrozen.

This strategy does not work here, since it can take a certain period of *simulated time* before p_{DE} writes on F_{DE}. And when p_{TDF} finally unfreezes, t_{DE} might be (much) larger than t^p_{TDF}, i.e. the token cannot be really used if the model is supposed to be sound.

As a first consequence, we follow the strategy from Section 4.1 and synchronize α_{DE} and α_{TDF} the same way. And we also synchronize every time before a TDF process is executed which reads from a FIFO written by a DE-process. That way we give the DE side a maximum chance to fill the FIFO sufficiently.

However, it can of course still happen that the FIFO does not contain enough token when it is read by p_{TDF}. This constitutes a corner case, and there are essentially three ways to handle it:

1. We could throw an error
2. We could fill the missing token with the last valid value
3. We could fill the missing token with a constant c

Which option is appropriate now depends on the use case: When the combined model should not exhibit this behavior, option 1 is the right choice. If this communication channel has a "sample and hold" style semantics, option 2 should be preferred[1]. And option 3 could be used if there is an appropriate default value.

We now construct a converter process $fetch^p_{PN}$ which has to be scheduled by α_{TDF} before every occurrence of p, which is the TDF process supposed to read from the FIFO F_{DE}. Instead of p, $fetch^p_{PN}$ reads from F_{DE} and copies the token to a FIFO F which is read by p.

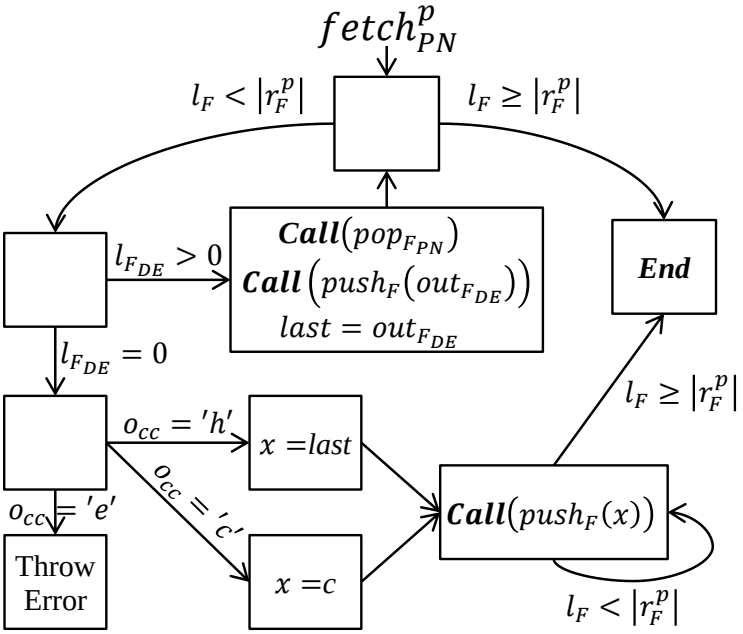


Figure 4.4: The $fetch^p_{PN}$ converter process

More precisely: When $fetch^p_{PN}$ is executed, it checks if F already contains enough token for p to read (i.e. checks if $l_F \geq |r^p_F|$). If not, it takes token from F_{DE} and pushes them to F until there are enough token. If this is not possible because F_{DE} does not contain enough token, then $fetch^p_{PN}$ handles this corner case depending on the value of the option variable o_{cc}:

- If o_{cc} ='e' (for "error"), an error is thrown and the execution of the model is stopped. We won't specify in detail what throwing an error exactly means here, for example it could mean calling a dedicated error process.
- If o_{cc} ='h' (for "hold"), the missing token in F are filled with a copy of the last token that could be copied from F_{DE} to F.

[1] Although one might argue that in this case a DE signal would have been the better choice to use for this channel in the first place

- If o_{cc} ='c' (for "constant"), the missing token in F are filled with a constant c.

Figure 4.4 shows the diagram of $fetch_{PN}^{p}$.

4.3.2 TDF writer

When a TDF process p_{TDF} writes to an unbounded FIFO, there is never a problem regardless of what kind of process read from F. Since p_{TDF} does not need to get blocked in order to yield to the simulation semantics, and there is obviously always enough space in F. When F is bounded, however, and read by a DE-PN process p_{DE-PN} it can happen that F is full.

Now we are in a corner case similar to the one in the previous section, but now we have to decide what to do with the surplus token. Storing it in an internal queue within p_{TDF} to write it to F later is not a real option, as this is essentially equivalent to enlarge the size of F. There are three possibilities which are viable:

1. We could throw an error
2. We could discard the oldest token in F and then write the new token to F
3. We could discard the new token, i.e. don't write to F

As before, thee is no apparent "correct" way which of these options to choose; it depends on the use case at hand. So we make all three options available to our converter process $write_{F}^{PN}$, which is called by p_{TDF} with the token v as an input of calling $push_F$. $write_{F}^{PN}$ is an additional access process of F which simply calls $push_F(v)$ if there is enough space in F. If not, it proceeds according to the option chosen (see also Figure 4.5:

- If o_{cc} ='e' (for "error"), an error is thrown and the execution of the model is stopped.
- If o_{cc} ='o' (for "oldest token discard"), pop_F is called, followed by $push_F(v)$.
- If o_{cc} ='c' (for "current token discard"), no action is taken (process ends).

This converter is very simple, as it's only task is corner case handling. There is also no need to schedule it in a special way, as it is called by the TDF process directly.

4.4 Discrete Event Models and Process Networks

This is an awkward pair of MoCs to combine, as one of the conversion directions is basically trivial, while the other direction is very artificial. Therefore we will only sketch this case. There is an implementation of these conversion within the Converterchannels in Chapter 5.

The trivial direction is if the writing process is a DE process p_{DE}. There is obviously no problem when the FIFO F is unbounded; p_{DE} just writes to F like an ordinary PN process. If the FIFO is bounded, p_{DE} might eventually block, which in general should be OK for most use cases. So there is no need for a converter, unless the blocking behavior is unwanted for some reason. In

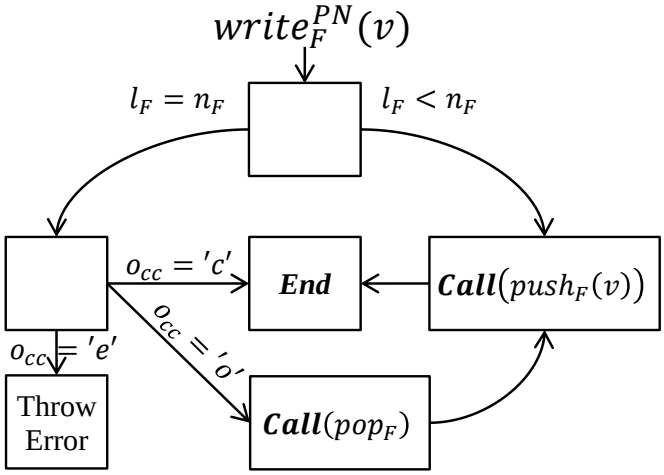


Figure 4.5: $write_F^{PN}$ converter process

that case we would need a converter $conv_{DE \to PN}$ for the sole purpose of handling the full FIFO with corner-case options like in section 4.3.2, i.e. throw an error or discard one token. In fact, this was done for the Converterchannels for technical reasons (see section 5.2.3).

If the writing process is a PN process p_{PN}, the situation is very different. For example, p_{PN} might require to be blocked by a bounded FIFO it writes to. If we replace that FIFO with a DE-signal s, p_{PN} would just keep on writing to s forever without ever blocking. And even if p_{PN} blocks by itself, e.g. because it's a DE-PN process, it might write big chunks of data to s at the same δ-cycle, which effectively constitutes a data loss.

The only remedy to the last two problems is to let p_{PN} write to a (possibly bounded) FIFO F as usual, and employ a DE converter process $conv_{PN \to DE}$ which reads the token from F one by one at a certain pace and then writes their values to s. The time between consecutive reads would usually be constant, but it could also change in some defined manner, e.g. according to the number of token in F. If F is empty when $conv_{PN \to DE}$ reads from it, we just would not write to s, which is essentially a "hold" semantic as in section 4.3.1.

That is, in contrast to the previous MoC conversion problems, there is no real "natural" MoC conversion semantic for the PN→DE conversion direction. While the Converterchannels in Chapter 5 also contain converters between both MoCs, they were mostly added for the sake of completeness, as the applications seem limited.

4.5 TDF models and TLM models

The problem of connecting TLM models to TDF models is essentially how to stream transactions to TDF processes, and how to transform TDF streams into transaction data. We use the simplified transaction as defined in Section 3.5, i.e. a transaction $T = (c_T, d_T, a_T, t_T, s_T)$ of the form

$$(read|write, (d_0, ...d_{n-1}), address, timestamp, state)$$

with the state-space depending on the TLM-MoC (LT-TLM or AT-TLM). In the following we will consider first the parts of the conversion which are common to both TLM-MoCs before elaborating on the details with respect to the two variants.

4.5.1 TLM writer

The basic idea of writing a transaction $(write, (d_0, ...d_{n-1}), address, timestamp, state)$ to a FIFO F which is read by a TDF process p in a TDF model is to to push the data token $(d_0, ...d_{n-1})$ successively to F. Note that we simplified the situation compared to SystemC TLM2 in that there is no data descriptor which states that the data token are written consecutively to the same address. In the TLM2 generic payload, this is established by choosing a streaming width sw smaller than the data length dl. In fact, if the size of the data type `T` of the TDF signal to be written to is s_T, the converter implementation in Chapter 6 has to check if $sw = s_T$ holds, in which case it streams dl/sw token to the TDF side. Otherwise the transaction cannot be processed.

The converter we need is a TLM Target as in Definition 3.19. We will for now assume that F is unbounded, and deal with the bounded situation later. The first problem to consider is if F contains enough token for p to consume when p is executed. Since we don't make any assumptions on the arrival rate of transactions, which can arrive asynchronously in general, we can't ensure that.

Therefore, we again are in the corner case situation of section 4.3.1, where we have to decide what to do if we can't provide an TDF process with enough token. We therefore use the same option $o_{cc} \in \{$`'e'`, `'c'`, `'h'`$\}$ for throwing an error, filling in a constant or holding the last value, respectively. The use case again drives the choice of the option. If the converter represents a D/A converter, filling the missing token with zeros is probably fine as an equivalent to having the D/A converter switched off. However if it represents D/A conversion based on pulse-width modulation, re-using the last available token might be a better option. And if the initiator was intended to deliver enough transactions, the converter should raise an error.

However, we can maximize the chances of DE processes to produce sufficient write-transactions by synchronizing with α_{DE} in the same manner as in section 4.1.2. We can do this every time before executing p, like in Section 4.3.1 as if p was reading from a DE signal, or implement a mechanism which does this *only* if there are not enough token in F. In any case, we can assume $t_{DE} = t^p_{TDF}$ from now on.

The next problem is more subtle: A transaction T can in general arrive with a timestamp $t_T > t_{DE} = t^p_{TDF}$. So if the token $(d_0, ...d_{n-1})$ from T are written to F right away, it might happen (depending on how many token were already written to F) that when p consumes (some of) these token, $t > t^p_{TDF}$. This causality issue should be avoided. While we saw in Section 3.4 that TDF itself has some ambiguity regarding causality, it is at least always bound by the cluster period. In this case, however, the difference between t_T and t^p_{TDF} can in theory grow arbitrary large.

We will handle this issue in a way which also minds another potential timing anomaly: Out-of-order arrival of transactions, i.e. when a transaction T' with timestamp $t_{T'}$ arrives after a transaction T with timestamp t_T, such that $t > t'$.

We will therefore use the strategy of storing the transactions first in a queue $Q = (T_0, T_1, ..., T_{n_Q-1})$ where they are ordered by time-stamp, i.e. if t_{T_i} is the timestamp of T_i, then $t_{T_i} \leq t_{T_j}$ iff $i < j$. The idea is now to write the token from these transactions to the FIFO F as late as possible. Whenever F needs to be filled, we look at the transaction T_0 at the top of Q. If $t_{T_0} \leq t^p_{TDF}$, we remove T_0 from Q and write the token of T_0 to F, and repeat this process until there are enough

token in F for p to read. If $t_{T_0} \leq t^p_{TDF}$, we are in the corner case situation as described above, and handle it accordingly.

This strategy achieves the following: It can never happen that token from transactions with timestamp $t_T \leq t^p_{TDF}$ are consumed by p. At the same time, it maximizes the chances that transactions arriving out-of-order get "re-ordered" in the queue. It can still happen that after a transaction T with timestamp t_T was removed from the queue and streamed to F, a transaction T' arrives with timestamp $t_{T'} < t_T$.

Now it depends again on the nature of the overall model if the modeler would consider this as an error or not. If so, we can include a check for this case by storing the timestamp t_{last} of the last transaction which was streamed to F, compare it with the timestamp t_T of the next transaction T to be streamed, and raise an error (or a warning) if $t_T < t_{last}$.

Now we look at the parts where we have to treat LT-TLM and AT-TLM differently.

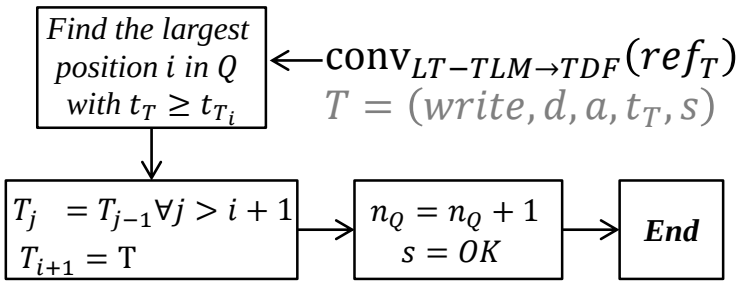


Figure 4.6: $conv_{LT-TLM \to TDF}$ process

4.5.1.1 LT-TLM converter

In LT-TLM, the converter (being a TLM Target) has to process the transaction before ending. If it just stores the reference to the transaction in Q and ends (to ultimately process it later), the initiator which issued it might re-use the transaction, such that its data might have changed when it is streamed to F. Therefore, we actually have to copy the transaction[2]. After copying the transaction T to Q, the converter sets the response of T to OK and ends. This describes the first converter process $conv_{LT-TLM \to TDF}$ (See Figure 4.6).

The process which finally writes the token from the "oldest" transactions in the queue is a TDF process $fetch^p_{TLM}$, which has to be inserted in the static schedule executed by α_{TDF} in front of every occurrence of p. Also, $fetch^p_{TLM}$ is treated like a process which reads from a DE-signal: Before executing $fetch^p_{TLM}$, α_{TDF} yields to α_{DE}. That way, we can make sure that DE processes get the opportunity to provide p with enough token. Figure 4.7 shows $fetch^p_{TLM}$ in detail. For simplicity, all the corner-case handling when there are not enough token available is represented by one box, since it is handled exactly like in DE-PN to TDF conversion in Section 4.3.1. If the case $t_{T_0} < t_{last}$ occurs, the behavior depends on the option variable o_{ord}. If o_{ord} = 'e', an error is raised, if o_{ord} = 'w', a warning is raised and the process continues.

Note that it would also possible that instead of synchronizing with α_{DE} every time before $fetch^p_{TLM}$ is executed to only synchronize with α_{DE} when necessary; i.e. when F contains not

[2]However, since only the data and the timestamp is needed at this point, in real-world implementations (where efficiency matters) only those two parts of the transaction are stored in the queue.

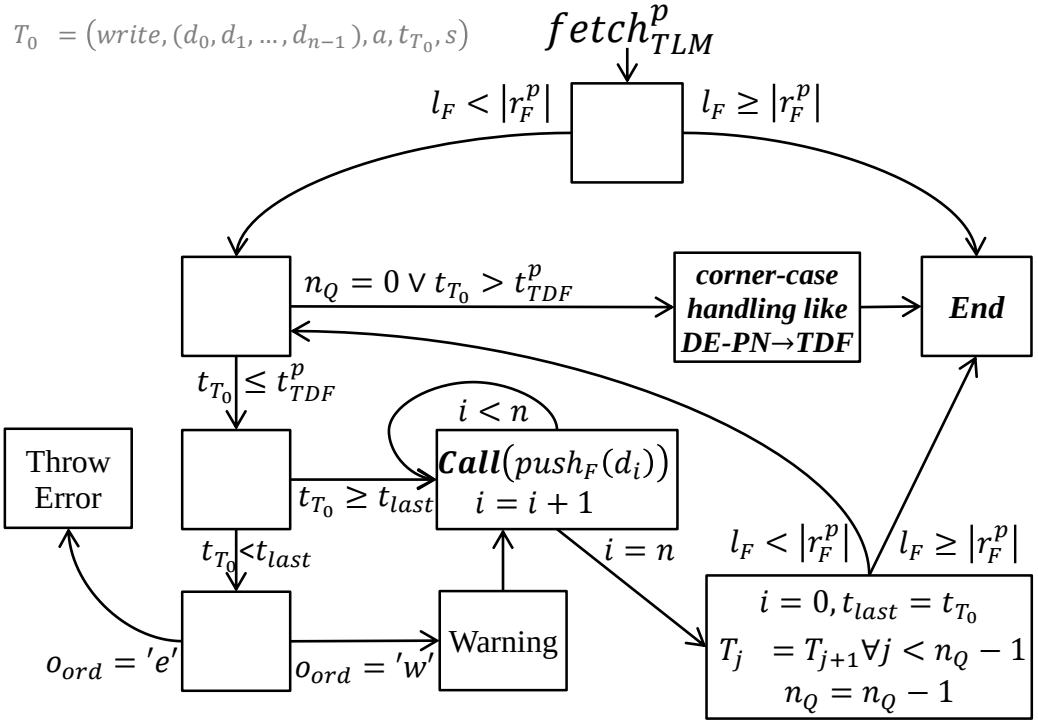


Figure 4.7: The $fetch^p_{TLM}$ process.

enough token *and* the transactions in Q also can't provide enough token. To this end, $fetch^p_{TLM}$ would schedule its wake-up by calling $newEvent((t^p_{TDF}, fetch^p_{TLM}))$ before freezing. However, according to the SystemC AMS extensions standard, this can't be implemented as calling `wait()` from a TDF module is not well-defined. And since this strategy does not change the results (while improving performance due to potentially less context switches), we opt for the simpler alternative.

The $fetch^p_{TLM}$ process writes all the token of a transaction to F at once; since we assumed F to be unbounded, this always works. To choose for F a fixed finite size, we would have to know an upper bound N of the number of tokens of transactions in the model. Then F could be chosen of size $N + r^p_F$.

Another option would be to take only as many token from the transaction T_0 as needed, and then take the next token from T_0 at the next execution of $conv_{LT-TLM \to TDF}$. However, it is possible that a transaction with a lower timestamp than t_{T_0} might arrive in between. To avoid this kind of disarray, using an unbounded FIFO F is the better option, especially when considering that the transaction queue is unbounded as well.

However, in this setting it is possible that token gradually pile up in the transactions within the queue Q. This would indicate an imbalance in the overall model as the initiator(s) produce more token than can be consumed. A way to detect such an imbalance would be to bound number of token in the queue relative to the token consumption of p in the time covered by the queue. More precisely, let n^Q_{token} be the number of token of all the transactions in the queue Q, and let $C > 1$ be a constant. By checking if the relation

$$n^Q_{token} < C r^p_F \frac{t_{T_{n_Q-1}} - t_{T_0}}{t_p} \tag{4.1}$$

holds (e.g. within the $conv_{LT-TLM\to TDF}$ process), and throwing an error or a warning if it does not, it can be ensured that the number of token in Q does not grow to large. E.g. with $C = 2$, there would be no more than twice as many token as p can consume within the queue. Having this kind of wiggle room is important in order to deal with temporal decoupled initiators in LT-TLM.

The $conv_{LT-TLM\to TDF}$ converter basically copies the token from incoming transactions and then completes the transaction. The token from the transaction get consumed later, so in a real-life implementation of the model there has to be some kind of buffer where the converter was. How big has this buffer to be?

From 4.1 we can derive that the maximum value of

$$\frac{n^{Q}_{token}}{r^{p}_{F}\frac{t_{T_{n_Q-1}}-t_{T_0}}{t_p}} \tag{4.2}$$

in the course of the simulation would be an upper bound on the size of the buffer in the implementation. This is an example on how the converter can provide additional information useful at a later refinement or implementation step.

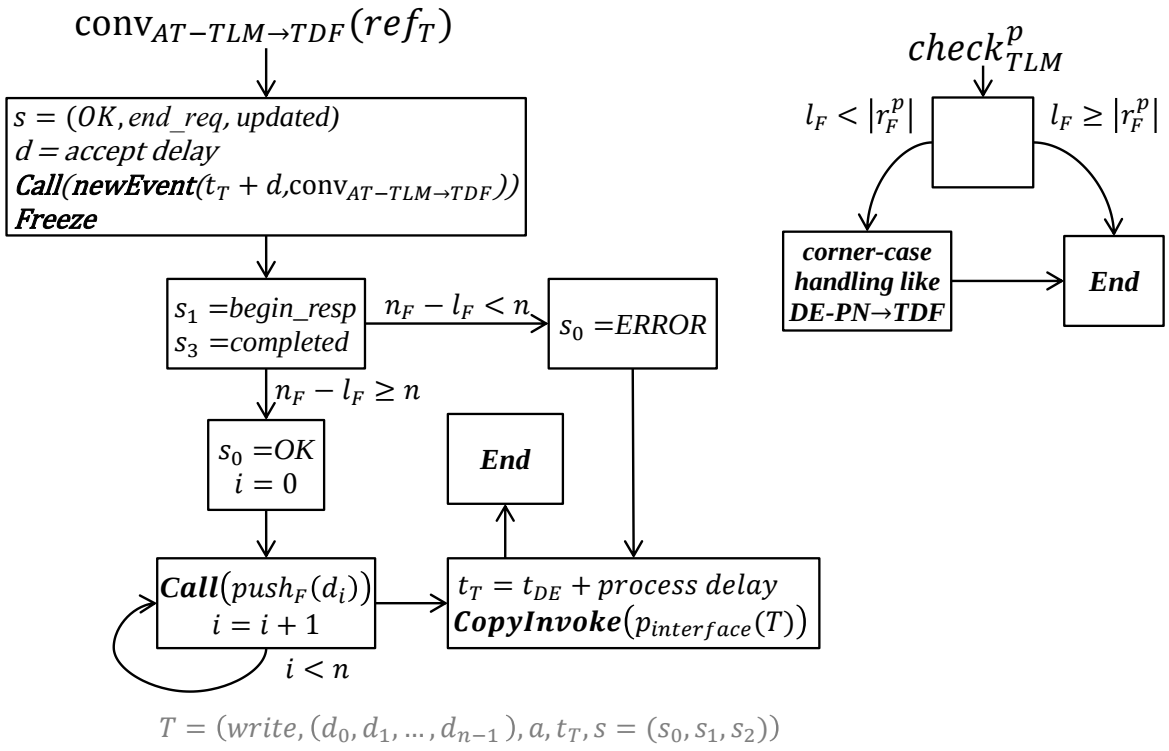


Figure 4.8: $conv_{AT-TLM\to TDF}$ and $check^{p}_{TLM}$ processes

4.5.1.2 AT-TLM converter

In AT-TLM, the streaming to an TDF process can be handled very differently as the target can respond to an incoming transaction T at a later time if $t_T > t_{DE}$, so it is possible to process the transaction actually at time $t_T = t_{DE}$ and write the token from T *directly* to F. Now if F is

finite, and the transaction causes an overflow to F, it is now possible to reject the transaction by setting the state to (ERROR,",").

Figure 4.8 shows the $conv_{AT-TLM \to TDF}$ process. In this case, we actually model an accept delay d[3], which is a common use case for AT-TLM models. Therefore the transaction is never processed right away. Instead, the $conv_{AT-TLM \to TDF}$ schedules its continuation at time $t_T + d$ and freezes, after also updating the transaction state accordingly, in particular by setting the phase to `end_req`.

After the process is invoked again by $\alpha_D E$, T can be processed directly by pushing its token to F; i.e. there is no need for a process like $fetch^p_{TLM}$. Now the transaction state is updated again by setting the phase to `begin_resp`, and setting the sync-state to `complete`.

In Figure 4.8 it can be seen that $conv_{AT-TLM \to TDF}$ also contains a test to check if F has enough space for the token of T. IF not, T is rejected with an ERROR state. This was not possible with LT-TLM, as the converter could not know if the token could be written to F at the time T was processed. However, in the implementation of the TLM→ TDF converters in chapter 6, we will use an unbounded FIFO.

After the transaction has been processed, the timestamp of the transaction is set to the current simulation time with an additional delay added to model the time to process the transaction. Then an interface process $p_{interface}$ of the component which passed the transaction to the converter is invoked. Since we set the sync-state to `complete`, the converter is done with the transaction. In general, the initiator could call the converter once more with phase `end_resp`, but we leave out this case for simplicity, as it also does not change the core conversion approach.

Not discussed so far is what to do if F contains not enough token when it is read by a TDF process p. A simple solution is to use a process similar to $fetch^p_{TLM}$ in section 4.5.1.1, which is scheduled by α_{TDF} just before p. However, this process now has just to check if there are enough token in F for the execution of p, therefore we call this process $check^p_{TLM}$. If there are not enough token, $check^p_{TLM}$ initiates the usual use-case driven corner-case handling as $fetch^p_{TLM}$ (see right side of Figure 4.8).

As a final note we should point out that with the AT-TLM converter, we can get a better picture than in the LT-TLM case how big a buffer in an implementation has to be if we start off with an unbounded FIFO F. It now suffices to check what the maximum number of token in F was in the course of the simulation.

4.5.2 TDF writer

When sending a $(read, (d_0, ...d_{n-1}), address, timestamp, state)$ transaction to a converter which reads from a FIFO F written by a TDF process p, the obvious approach is to remove n token from F and copy them to $(d_0, ...d_{n-1})$. The obvious corner case is now that there are not enough token in F, i.e. $l_F < n$.

We now have the same use-case driven options as in section 4.3.1: Throwing an error, repeating the last valid value, or fill in the missing token with a constant value. However, for the TDF→TLM conversion direction, two more options make sense:

[3]How this delay is determined is not important here, but most of the time it would just be a constant value, i.e. a fixed property of the converter

- Dismissing the transaction with an ERROR response. In this case, the initiator can decide the further actions. It could now throw an error itself, or it could try again after some time. The latter might be a viable option in certain use cases.
- The converter waits until there are enough token present in F.

In the following we consider only the *wait* option, as handling the other cases works essentially like in the conversion problems in section 4.3.1 (error, hold, constant) and section 4.5.1.2 (error response).

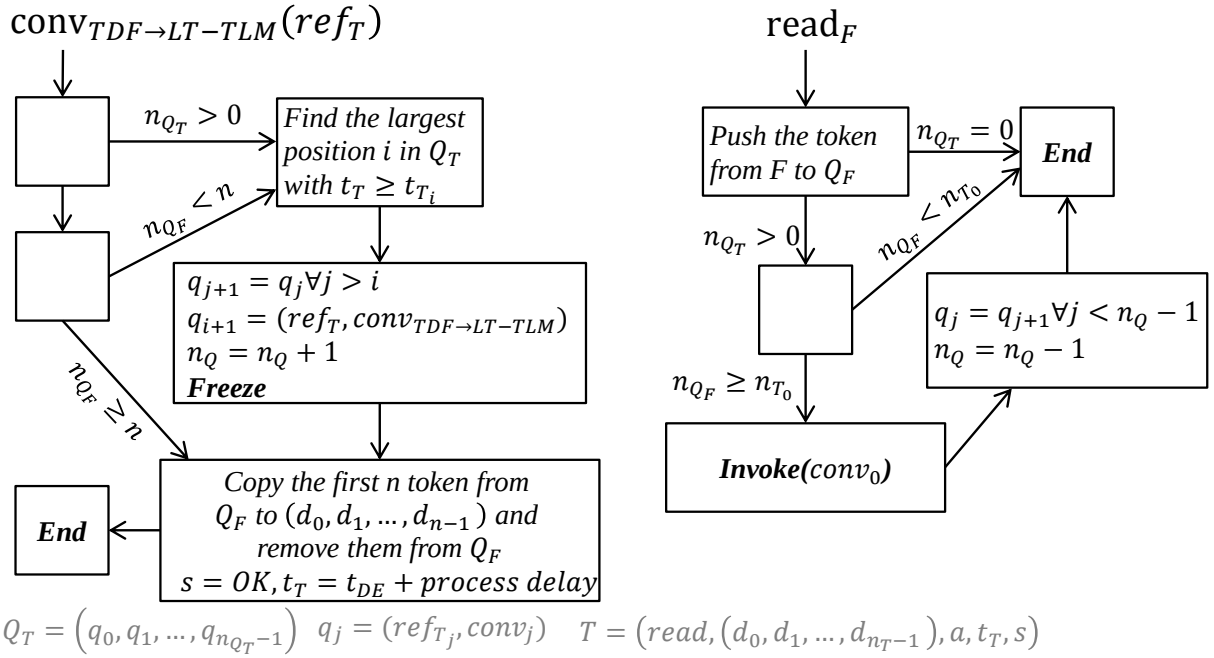


Figure 4.9: $conv_{TDF\to LT-TLM}$ and $read_F$

4.5.2.1 LT-TLM converter

The general idea is to have a TDF process $read_F$ which reads from the FIFO F with the same data rate as p writes to it. $read_F$ now stores the token in an internal queue Q_F, from which the actual converter process $conv_{TDF\to LT-TLM}$ receiving the transactions now can take its token.

If $conv_{TDF\to LT-TLM}$ receives a read-transaction T, but there are not enough token to read in Q_F, it stores a reference to itself and Tin a queue Q_T similar to a payload event queue as described at the end of section 3.5.2 (see Figure 3.11 there). After that, it freezes. Note that it is necessary to store the reference to $conv_{TDF\to LT-TLM}$, since it is actually a copy of $conv_{TDF\to LT-TLM}$ according to rule 2 of Definition 3.20.

Q_T is ordered according to the transaction time stamps (like the transaction queue we used for the LT-TLM→TDF converter in section 4.5.1.1), and is accessed by $read_F$. If $read_F$ is executed, it checks (after copying the token from F to Q_F) if Q_F has enough token the first element in Q_T to be processed. If so, it invokes the corresponding copy of $conv_{TDF\to LT-TLM}$, which then finishes its transaction.

This approach makes it necessary that $conv_{TDF \to LT-TLM}$ has also to check if there are already frozen copies of itself waiting in Q_F. If so, it also has to be enqueued in Q_F, even if there are enough token available for its transaction T. Figure 4.9 shows diagrams for the two converter processes.

4.5.2.2 AT-TLM converter

For the corresponding conversion in the AT-TLM MoC, the same TDF process $read_F$ can is used. The converter $conv_{TDF \to AT-TLM}$ also works very similar to $conv_{TDF \to LT-TLM}$, with some added steps for the updates of the transaction state. Figure 4.10 shows the diagram of the $conv_{TDF \to AT-TLM}$ process.

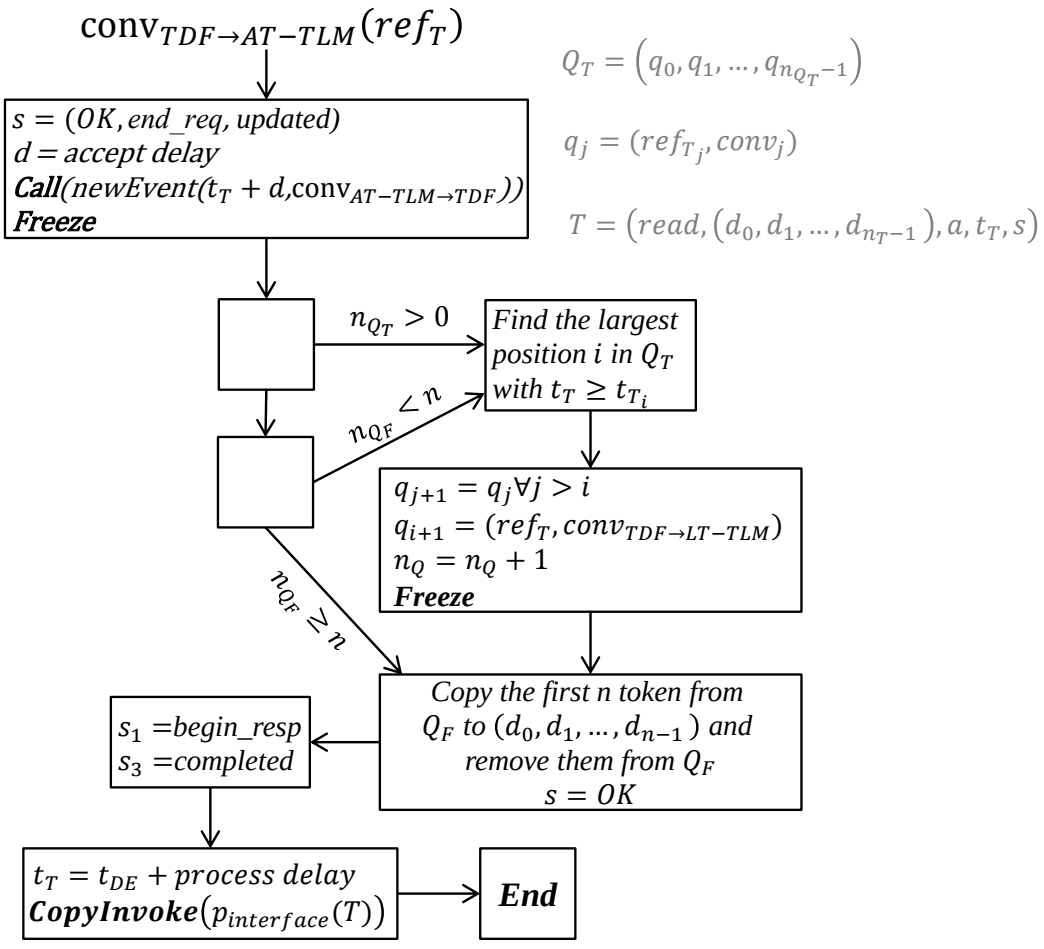


Figure 4.10: $conv_{TDF \to AT-TLM}$ converter process

5 Automatic MoC conversion in SystemC: Converter Channels

The Converter Channel library is an extension of SystemC and SystemC AMS which allows for automatic conversion between model parts modeled with different MoCs. It is part of the ANDRES modeling framework, which was the main technical result of the ANDRES project [HOH+07], and was documented in the Deliverables D1.5b and D1.6b of the ANDRES project [DW08, AND09]. Most of the information of this chapter can also be found in these reports.

From a C++ programming perspective, a converter channel is a C++ template-class named `converterchannel<>`, which derives from the SystemC class `sc_core::sc_channel`, which is the class all signal-like classes in SystemC have to inherit from. A converter channel is instantiated as follows:

```
converterchannel<MOC_WRITE, T_W, T_R1, T_R2> conv;
```

where

- `MOC_WRITE` denotes the MoC of the writing side (mandatory)
- `T_W` denotes the data type of the writing side (mandatory)
- `T_R1` and `T_R2` denote the possible data types of the reading sides of this converter channel (optional)

If only the mandatory writing side data type `T_W` is set, the converter channel doesn't perform data type conversion; all ports bound to it have to be of data type `T_W`. In general, the writing port bound to the converter channel has to be of data type `T_W`, and any reading port bound to it has to be of one of the data types `T_W`, `T_R1` or `T_R2`. The MoC `MOC_WRITE` is specified with an enum type `MoC`, which has the following members:

- `SC` denotes the DE MoC of SystemC
- `TDF` denotes the Timed Synchronous Data Flow (TDF) MoC of SystemC AMS.
- `FIFO` denotes all Process Network (PN) MoCs. For the conversion, it is not important if the modules connecting to the converter channel are timed or untimed, or if they use blocking or non-blocking access.

- `ELEC_VOLTAGE` and `ELEC_CURRENT` denote the continuous time electrical network (CT-NET) MoC of SystemC AMS, depending on the electrical quantity of interest

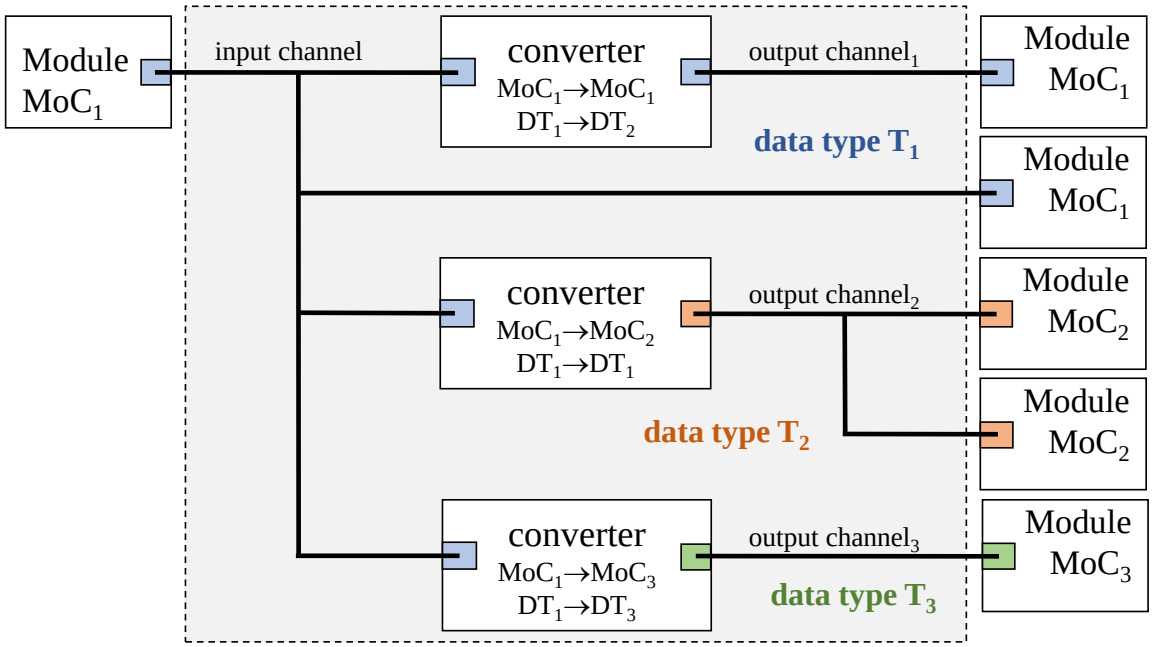


Figure 5.1: Internal structure of a Converterchannel (gray box), from [DHHV07]

An example of how a Converterchannel looks like inside is shown in Figure 5.1: For the writing side, it has an internal input channel of fitting MoC and data type, and for the writing side it has output channels for every MoC/data type combination needed.

The Converterchannel can be connected to any SystemC or SystemC AMS standard port (e.g. `sc_in<>`, `sc_fifo_out<>` or `sca_tdf::sca_out<>`. The MoCs of the reading side are not explicitly specified, but are determined automatically. Before we discuss the MoC conversions itself, we describe the technical implementation that enables this capability.

5.1 Technical Implementation

For the writing side, the Converterchannel contains an object `insignal`, which is an instance of a helper class `helpersignal`, which inherits from the correct signal class depending on the value of the template parameter `MOC_WRITE`, i.e. the writing side MoC. `helpersignal` is implemented as a template class, where one template parameter is of type `MoC` and the other specifies the data type. For each MoC, there is a template specialization (See Listing 5.1).

```
template <MoC THEMOC, typename T> // generic prototype classe
class helpersignal{
  public: helpersignal(){}        // dummy constructor
  public: helpersignal(int x){}   // dummy constructor with integer input. This is needed for
};                                // the FIFO specialization

template <typename T>                                          // FIFO specialization
class helpersignal<FIFO,T>: public sc:core::sc_fifo<T>{        // inheriting from sc_fifo<T>
```

```
  public: helpersignal(){}
  public: helpersignal(int size): sc:core::sc_fifo<T>(size){} // constructor to set FIFO size
};

// the rest of the template specializations are omitted
```

Listing 5.1: helpersignal helper class

For the writing side, the Converterchannel contains for each port-class it can connect to three pointers to a fitting channel (like `sc_signal<>`), one for each data type `T_W`, `T_R1`, `T_R2`; see Listing 5.2 for an example.

```
sca_tdf::sca_signal<T_W>* outsignal_tdf_0;  // pointers for tdf signals to be passed to reading ports
sca_tdf::sca_signal<T_R1>* outsignal_tdf_1;
sca_tdf::sca_signal<T_R2>* outsignal_tdf_2;

sc_fifo<T_W>* outsignal_fifo_0;   // pointers for FIFOs to be passed to reading ports
sc_fifo<T_R1>* outsignal_fifo_1;
sc_fifo<T_R2>* outsignal_fifo_2;
```

Listing 5.2: output signal pointers for the TDF MoC and the FIFO MoC

These pointers are all instantiated to `0`, and corresponding channels will only be created if the Converterchannel is connected to a corresponding port. Which of the internal signals (the input signal or the output signals) has to be connected to a specific port is now determined by overloading the `()` operator used in the SystemC syntax `module.port(signal)` to connect a port of a module to a signal, for example like in Listing 2.2 of section 2.1.1. How this works for the TDF ports is shown in Listing 5.3, for the other port classes this works essentially the same.

```
operator sca_tdf::sca_signal<T_W>& ()   // operator returns a tdf signal of type T_W
{
  if(MOC_WRITE!=TDF)                    // If TDF is not the writing MoC
  {                                     // we have to return outsignal_tdf_0
    if(outsignal_tdf_0==0){             // not created yet? => create it
      outsignal_tdf_0 = new sca_tdf::sca_signal<T_W>;
    }
    return *outsignal_tdf_0;
  }                                 // if TDF is the writing side MoC
  else return *(dynamic_cast<sca_tdf::sca_signal<T_W>*>(insignal));
}                                   // we simply return the (casted) input signal

operator sca_tdf::sca_signal<T_R1>& ()  // operator returns a tdf signal of type T_R1
{                                       // i.e. outsignal_tdf_1
  if(outsignal_tdf_1==0){               // which is created now it it has not been created yet
    outsignal_tdf_1 = new sca_tdf::sca_signal<T_R1>;
  }
  return *outsignal_tdf_1;
}

operator sca_tdf::sca_signal<T_R2>& ()  // same procedure (omitted)
```

Listing 5.3: connecting the Converterchannel to tdf ports

Note that overloading of the `()` operator together with casting techniques as shown in Listing 5.3 have also been used in [Sch07]. With this mechanism, the Converterchannel creates for each port it connects to the right signal during the elaboration phase. The sizes of internal FIFOs are set by passing them either

- to the constructor of the Converterchannel if `FIFO` is the writing side MoC.
- or to the method setReadingFIFOSize(int size) if `TDF` is the writing side MoC.

In the standard SystemC method `before_end_of_elaboration()` of the `sc_core::sc_channel` class we now connect converter modules to finalize the internal structure as shown in Figure 5.1:

```
if(outsignal_tdf_1!=0)                // tdf output signal 1 is present?
{                                     // create a converter to the TDF MoC
  conv_2tdf_1 = new converter<MOC_WRITE, TDF, T_W, T_R1>("conv_2tdf_1", options_1);
  conv_2tdf_1->in(*insignal);         // connect it to the input signal
  conv_2tdf_1->out(*outsignal_tdf_1); // and to outsignal_tdf_1
}
```

Listing 5.4: Connecting the converters

Listing 5.4 shows how this works for the first tdf output signal; the code for the other output channels is similar. The parameter `options_1` passed to the constructor of the converter is of type `converterchannel_options<T_W, T_R1>*`, and is used to pass the several conversion options of the Converterchannel to the converter. We won't get into details regarding the `converterchannel_options` class, but will discuss the several options regarding MoC-conversion and data type conversion of the Converterchannels later. .

Listing 5.5 below shows the code for the converter `converter<SC, TDF, T_WRITE, T_READ>` from DE to TDF. It is an `sc_module` which contains a TDF sub-module `converter_sc2tdf_sub` which does the actual conversion, i.e. the MoC conversion (by means of the converter port `sca_tdf::sca_de::sca_in<T_WRITE>`), as well as the data type conversion (by means of inheriting from `datconv<T_WRITE, T_READ>`).

If the option to use the `lost_val_alerter` is set, the converter creates another sub-module that is used to check how many value changes the input signal had in between two executions of the `processing()` method of `converter_sc2tdf_sub`.

```
template <typename T_WRITE, typename T_READ>
class converter<SC, TDF, T_WRITE, T_READ>: sc_module // DE to TDF converter, main module
{
  public:
  sc_in<T_WRITE> in;            // port for the DE input signal
  sca_tdf::sca_out<T_READ> out; // port for the TDF output signal

  lost_val_alerter<T_WRITE>* alerter; // determines if DE signal changes have been lost

  converter_sc2tdf_sub<T_WRITE, T_READ>* tdf_sub; // submodule which contains the actual converter

  converter(sc_module_name n, converterchannel_options<T_WRITE, T_READ>* options)
  {
    if(options->use_lost_val_alerter) // if the option is set
    {                                 // instantiate the detector for lost DE signals changes
      alerter = new lost_val_alerter<T_WRITE>("alerter");
      alerter->in(in);        // connect it to the input signal with port-to-port mapping
                              // instantiate the actual converter with a pointer to the alerter
      tdf_sub = new converter_sc2tdf_sub<T_WRITE, T_READ>("tdf_sub",options,alerter);
    }
    else
    { // instantiate the actual converter without alerter
      tdf_sub = new converter_sc2tdf_sub<T_WRITE, T_READ>("tdf_sub", options);
    }
```

```
    tdf_sub->in(in);    // connect the actual converter to the signals
    tdf_sub->out(out);  // via port-to-port mapping
  }
};

// The actual MoC converter
template <typename T_WRITE, typename T_READ>
class converter_sc2tdf_sub: sca_tdf::sca_module,
                            public datconv<T_WRITE, T_READ> // inherit from the right
{                                                           // data conversion class
  public:
  sca_tdf::sca_de::sca_in<T_WRITE> in;  // converter port SC -> TDF
  sca_tdf::sca_out<T_READ> out;         // TDF output port

  lost_val_alerter<T_WRITE>* alerter;

  converter_sc2tdf_sub(sc_module_name n,
                       converterchannel_options<T_WRITE, T_READ>* options,
                       lost_val_alerter<T_WRITE>* al = 0)
                      :datconv<T_WRITE, T_READ>(options)  // call constructor of data type converter
  {alerter=al;}

  private:
  void processing() {
    out.write(conv(in.read())); // write the input to the output after data type conversion
    if(alerter){                              // if the alerter option is used
      int lost_vals = alerter->get_count();   // check the number of value changes of the input
      if(lost_vals>0){                        // report a warning if there where value changes
        SC_REPORT_WARNING(...);
      }
    }
  };
};
```

Listing 5.5: DE to TDF converter

5.2 MoC conversions

The converter channels provide conversion means for basically every pair of MoCs in both directions, as shown in Table 5.1. A conversion between voltages and currents has not been implemented, since these are not really different MoCs, and also there is no real use case for this. SystemC AMS facilities like voltage controlled current sources can be used directly for this. Note that although electrical networks are treated as a MoC of its own right, they need to be connected to TDF modules in some way, e.g. with a TDF-controlled current source or a voltage-to-TDF converter. THis can be done directly, or by using a Converterchannel.

In the following every MoC conversion is described shortly with respect to the corresponding discussion in chapter 4.

5.2.1 TDF $\leftrightarrow$ SC

For these conversions, the SystemC AMS converter ports mentioned in section 2.2 are used. They provide the basic conversion semantic as discussed in section 4.1. In particular, the

Table 5.1: Overview on the MoC conversion capabilities, taken from [AND09]

from \ to	SC	TDF	FIFO	ELEC_VOLTAGE	ELEC_CURRENT
SC		✓	✓	✓	✓
TDF	✓		✓	✓	✓
FIFO	✓	✓		✓	✓
ELEC_VOLTAGE	✓	✓	✓		X
ELEC_CURRENT	✓	✓	✓	X	

`sca_tdf::sc_in` ports trigger the SystemC AMS kernel to yield to the SystemC simulation kernel before executing processes accessing such ports, and the `sca_tdf::sc_out` ports generate events (possibly with delays) which cause the corresponding `sc_signal<>` to be written at the right time. Also the static schedule is optimized to minimize the difference between SystemC time and SystemC AMS time.

For corner case handling, the SC→TDF converter provides the option to detect the value change events which occur between executions of the reading TDF process as has been shown above in Listing 5.5. For the TDF→SC conversion direction, there are no real corner cases. However, there might be applications where knowing the actual points in time the TDF side is written is helpful, since there are no value change events if consecutive TDF token are equal in vale. Therefore, by using the method `conv.clock_tdf();` of a converter channel `conv`, a Boolean clock signal can be accessed which runs with the speed of the TDF cluster. Every time the converter channel gets a new token from the TDF side, there is a rising edge.

5.2.2 TDF ↔ FIFO

The main conversion endeavor here is to manage the corner cases as discussed in section 4.3. For the FIFO→TDF conversion direction, the corner case is an empty internal FIFO. To set the corner case option, the Converterchannels have a method `setFIFO2TDFemptybuffer(TDF_FIFO_OPT opt);`, where `TDF_FIFO_OPT` is an enum data type with values `error` (which is the default), `hold` and `constant`. The semantics is as discussed in section 4.3.1. If the `constant` option is chosen, the value of the constant can be specified with the method `setFIFO2TDFemptybufferConstant(T_READ val)`.

For the TDF→FIFO conversion, the corner case is a full FIFO. The corner case handling can be set with the method `setTDF2FIFOfullbuffer(TDF_FIFO_OPT opt)`, where `opt` can have the values `error` (again the default), `discardOldest` and `discardCurrent`.

5.2.3 SC ↔ FIFO

For the con FIFO→SC conversion direction, the periodic reading approach as discussed in section 4.4 has been implemented. The time period *has* to be set with the Converterchannel method `set_sampling_period(sc_time t)`; failing to do so raises a run-time error. If the incoming FIFO is empty, the value of the outgoing `sc_signal` will *not* be updated, i.e. there is no additional corner case handling.

For the SC→FIFO conversion, a corner case handling for a full internal FIFO was implemented with the same options as in the TDF→FIFO case (which can be set with the Converterchannel method `setSC2FIFOfullbuffer(TDF_FIFO_OPT opt)`). The simple reason for this is that the Converterchannel is connected to a writing module via an `sc_out<>` port. Therefore, the `insignal` is an `sc_signal<>`, read by the internal converter module, which then writes the value to the internal `sc_fifo<>`. If that internal FIFO is full, a blocking write to it would block the internal converter, not the module which wrote to the Converterchannel.

So while an SC→FIFO conversion could be in theory trivial (see section 4.4), due to the Converterchannel approach to provide channels of fitting types to the modules connected it cannot be implemented that way.

5.2.4 Conversions towards electrical networks

These conversions have not been discussed in Chapter 4 since electrical networks in SystemC AMS are essentially (as mentioned in section 2.2) equation systems that are evaluated every time there is an access by a TDF facility like a voltage source controlled by a TDF signal.

If an `sca_eln::sca_terminal` (e.g. a connector of a resistor) is bound to a converter channel, it is internally bound to an `sca_eln::sca_node`. With the writing side MoC being `SC`, `TDF` or `FIFO`, the idea is that the incoming signal will control either a voltage or a current source. As the latter cannot be determined automatically, it has to be set with the Converterchannel method `setElec_Mode(MoC M)` where `M` is either `ELEC_VOLTAGE` or `ELEC_CURRENT`.

This generates an appropriate internal voltage or current source, which is either connected *directly* to the insignal in case of `TDF` and `SC`, as SystemC AMS provides also voltage/current sources controlled by `sc_signal`s, or via an additional FIFO→SC converter if `FIFO` is the writing side MoC. In the latter case a sampling period has to be specified.

The internal voltage/current source has a positive and a negative terminal to the electrical side. While the positive terminal is connected to the internal `sca_eln::sca_node`, we also need to connect the negative terminal. For this there is a second internal node, which can be accessed with the Converterchannel method `neg_elec_node()`. The voltage/current generated can also be scaled; this can be controlled with the method `setElec_Scaling(double val)`, where the default value is 1.

5.2.5 Conversions from electrical networks

These conversion direction means that an electrical voltage or an electrical current quantity is measured, and converter to a signal, i.e. `SC`, `TDF` or `FIFO`. Which quantity is measured depends on the writing side MoC template parameter of the converter channel, i.e. `ELEC_CURRENT` or `ELEC_VOLTAGE`.

The internal measurement unit has also be connected to two nodes. The internal node is connected to the positive side. For the negative side, there is also an internal negative node (to be accessed again with `neg_elec_node()`. However, if this node is not connected to a terminal of the electrical network on the writing side, the negative connector of the internal measurement unit is connected to ground.

The quantities measured can also be scaled with `setElec_Scaling(double val)`. For reading FIFOs, the corner case handling can be set with `setELEC2FIFOfullbuffer(TDF_FIFO_OPT opt)`.

When converting from `ELEC_CURRENT` to another MoC with a converter channel, and the negative internal electrical node is not connected by the user, the current flow from the converter channels internal positive electrical node to gnd is measured. However, in some cases the user might want to account for the resistance of a system part on the converter channel's reading side, which is still modeled non-conservatively.

Therefore, the converter channel provides a method `setTerminating_Resistance(double val)` to insert a terminating resistance with `val` Ohm, such that the current flow from the internal positive electrical node through the terminating resistance is measured.

5.3 Data type conversion

The data type conversion capabilities have not been discussed so far. As can be seen in Listing 5.5, the data type capabilities are implemented by letting the internal converter module inherit from `datconv<T_WRITE, T_READ>`, which then provides a method `conv(T_WRITE val)` with return type `T_READ`.

The nature of the data type conversion depends on the two data types involved. In some cases, a simple cast is sufficient, but in most cases there will be issues that have to be specified with methods to choose from different options; for example

- For conversion from floating point types to integer types, it can be chosen if values are rounded, or a floor/ceiling behavior is performed.
- How overflows are handled in cases when the value range of the writing side data type is larger than the reading side data type. The values can be clipped, for example, but in other cases a conversion with a modulo-like behavior can make more sense. This again depends on the use case.
- To scale value ranges when converting from floating point types to integer types. This is necessary, for example, when a Converterchannel effectively models an A/D converter which measures voltages from 0 to 5 Volts, and we want to convert them to the full range of an 16-bit vector

Some conversions are not supported at all, as there are no clear conversion semantics, e.g. when converting from the SystemC data type `sc_dt:sc_logic`, as this data type has beside 0 and 1 values to describe high impedance and undefined. There is no obvious conversion to a numerical type like `float` or `int` for these values.

We won't go into the details of all the Converterchannel data type conversion approaches. However, Figure 5.2 on the next page gives an idea on how the different conversion directions are handled, and what kind options are available for each.

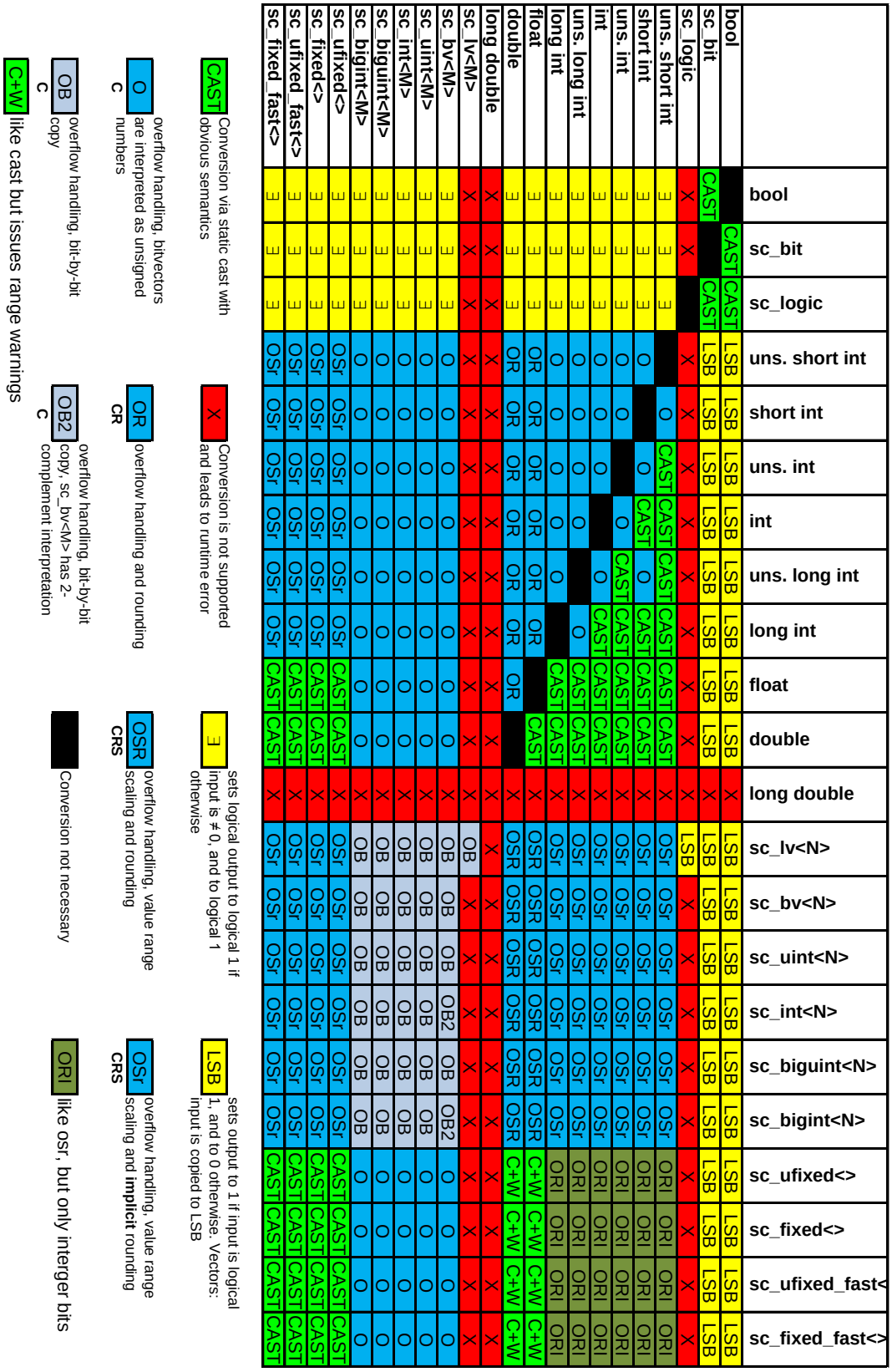


Figure 5.2: Datatype conversions handled by the Converterchannels, taken from [AND09]

6 TLM $\leftrightarrow$ TDF conversion in SystemC

In this chapter, we describe the implementation of the converters between TDF models and TLM models, using the concepts discussed in Section 4.5. In some cases, we have to deviate from these concepts a bit for technical reasons. For example some of the converters described get a transaction as an input, and directly access a TDF FIFO, like the $conv_{AT-TLM \rightarrow TDF}$ process in Section 4.5.1.2 (see also Figure 4.8). This cannot implemented that way in SystemC/SystemC AMS, since the TDF ports have to accessed from within the `processing()` method of a TDF module.

On the other hand, implementing TLM interfaces within TDF modules is prohibited by the SystemC AMS standard; even if it might work with the current proof-of-concept implementation, this might not be the case for future versions or other implementations.

Therefore, each converter is an `sc_core::sc_module`, which contains an `sca_tdf::sca_module` as a sub-module. The `sc_core::sc_module` contains all interface methods, callbacks and threads needed to handle the TLM part of the conversion, while the `sca_tdf::sca_module` handles the TDF part, i.e. accessing the TDF signals, but also synchronization with the SystemC simulation kernel. To this end, each of the TDF sub-modules contains an `sca_tdf::sca_de::sca_in<bool> sync` port, which is connected to a dummy DE signal by the `sc_core::sc_module`. The TDF sub-module actually never accesses this port; its presence alone ensures that the `processing()` method is executed as close to the SystemC simulation time as possible.

In general, the sub-module will also hold an internal buffer (e.g. a `std::dequeue`) to either store TDF read or to take token from to write to TDF. In the latter case these internal buffers hold surplus token from transactions where some token have been already reads from. This ensures an equivalent behavior as specified in section 4.5.1, which assumed on an *unbounded* FIFO.

In the following, we look at both conversion directions and discuss the relevant code parts of the four converters. In Section 6.3, we will show an application example.

6.1 TLM→TDF conversion

The general approach of conversion from TLM to TDF was discussed in section 4.5.1: The token from the write-transactions have to be streamed to the TDF side, and as the TDF side needs a steady supply of token which the TLM side might not be able to oblige, this corner-case has to be handled according to the three options discussed. The TLM→TDF converters for both coding styles use an enum type for these options:

```
enum TLM_2_TDF::conv_option{
    ERROR = 0,
    HOLD,
    CONSTANT
};
```

As also discussed in section 4.5.1, the incoming transactions need to be stored in some way, as they come in general with an offset to the current simulation time. We now look first at the AT-TLM converter, and then have a look at the differences to it of the corresponding LT-TLM converter.

6.1.1 AT-TLM converter

Listing 6.1 shows the constructor prototype of the `TLM_2_TDF::conv_AT` converter. Beside the necessary parameters for the TDF sub-module, i.e. data rate and time step, and conversion option parameters, there are also time parameters to model latencies. The `conv_option` enum type can take the values `ERROR`, `HOLD` and `CONSTANT`.

```
TLM_2_TDF::conv_AT(
  sc_core::sc_module_name nm,
  sc_core::sc_time _accept_delay,          // accept delay
  sc_core::sc_time _latency_per_byte,      // latency per byte processed
  sc_core::sc_time tdf_timestep,           // TDF time step
  unsigned int _rate = 1,                  // TDF data rate
  conv_option  _option = ERROR,            // converter option
  unsigned int _initial_queue_token = 0,   // number of initial token in internal queue
  T _empty_fifo_dummy  = (T)(0)            // value for the converter option CONSTANT
  )
```

Listing 6.1: constructor prototype of the converter

The `T` is a template parameter for the data type of the outgoing TDF port. Also, the parameter `_initial_queue_token` has to be explained: When the SystemC Simulation starts, the TLM initiators usually don't provide write-transactions right away. Depending on the model, it might take some time (in simulated time); all the while the TDF side needs to consume token. And when the `ERROR` conversion option was chosen, this means that the simulation stops in the first δ-step of time 0.

Therefore the internal queue can be filled with initial values according to the `_empty_fifo_dummy` parameter, which therefore has a use even if the `CONSTANT` corner-case option is not chosen.

In section 3.5.2 we pointed out that in our formalism according to Definition 3.1, the AT-TLM MoC can be realized in a simpler fashion than in SystemC, as the *CopyInvoke* operation provides us with an alternative way for non-blocking access. Because of that, we don't need to use facilities like payload event queues (PEQs). Of course in the implementation we have to use PEQs.

In the following, we look at the important code segments of the converter, with the idea to show how a transaction is passing through the different methods and elements involved. Therefore we look first at the implementation of the `nb_transport_fw` method, where we will also encounter the PEQ:

```
tlm::tlm_sync_enum TLM_2_TDF::conv_AT::nb_transport_fw(
  tlm::tlm_generic_payload& trans,
  tlm::tlm_phase&           phase,
  sc_core::sc_time&         delay)
{
  switch (phase){
    case tlm::BEGIN_REQ:                // a fresh transaction arrived
      delay += accept_delay;            // add accept delay to the transaction delay
      trans_queue.notify(trans, delay); // put the transaction on the PEQ
      phase = tlm::END_REQ;             // update phase
      return tlm::TLM_UPDATED;          // and return
      break;
    case tlm::END_RESP:                 // the initiator is done with the transaction
      end_resp_event.notify (delay);    // "unblock" the converter
      return tlm::TLM_COMPLETED;
      break;
    default:                            // all other phases are errors
    SC_REPORT_ERROR(name(), "Bad phase");
    return tlm::TLM_COMPLETED;
  }
}
```

Listing 6.2: The `nb_transport_fw` method of the converter

We come back to the `end_resp_event` later. For now it suffices to know that the transaction is on the PEQ `trans_queue`, which is of type `tlm_utils::peq_with_get<>`. This kind of PEQ does not employ callback functions; instead it stores the transactions in order of their time-stamps, and offers an event (accessible by the PEQ method `get_event()`) which is notified every time a transaction is ready. With the PEQ method `get_next_transaction()`, the transactions can then be retrieved. The converter method `process_transaction()`, which we look at next in Listing 6.3, is sensitive to the PEQ event:

```
virtual void TLM_2_TDF::conv_AT::process_transaction(){ // sensitive to trans_queue.get_event()
  T value;
  unsigned char*   data_ptr;
  unsigned int     data_len, str_wid;
  tlm::tlm_generic_payload* transp;
  sc_core::sc_time delay = sc_core::SC_ZERO_TIME;
  // take all transactions which are read from the queue:
  while ( (transp = trans_queue.get_next_transaction()) != NULL){
    if(check_transaction( // check_transaction() checks if the transaction can be processed
        *transp,          // e.g. is streaming width = sizeof(T)?
        sizeof(T),        // if so, it extracts the data-pointer and the data length
        data_ptr,         // and returns it, as the last two parameters
        data_len          // are passed by reference
        )){               // note that check_transaction() also sets an error response...
      str_wid = sizeof(T);  // ...in case the transaction does not fit here
      delay = data_len*latency_per_byte;  // compute processing delay
      transp->set_response_status( tlm::TLM_OK_RESPONSE );
      for(unsigned int i = 0; i < data_len; i += str_wid ){ // copy the values from the
        memcpy(&value, data_ptr + i, sizeof(T));            // transaction to the token queue
        writer.ready_queue.push(value);                     // of the TDF-sub-module
      }
```

```
      if(writer.ready_queue.size() > writer.max_queue_token)  // For statistics: keep track of
      writer.max_queue_token = writer.ready_queue.size();     // maximum number of token
    }                                                         // in the queue
    tlm::tlm_phase  phase = tlm::BEGIN_RESP;  // update phase and send via backward path
    tlm::tlm_sync_enum status = socket->nb_transport_bw( *transp, phase, delay);

    switch (status){
      case tlm::TLM_COMPLETED:
        next_trigger (delay);                 // continue only after delay is realized
        return; break;
      case tlm::TLM_ACCEPTED:
        next_trigger (end_resp_event);        // continue only after end response was received
        return; break;
      default:
        SC_REPORT_ERROR(name(), "Bad status");
      }
    }
    next_trigger( trans_queue.get_event() );  // restore original sensitivity
  }
};
```

Listing 6.3: Taking the transactions from the PEQ and responding via the backward path

Listing 6.3 shows that `process_transaction()` actually removes as many transactions from the queue as possible. Apart from that, `process_transaction()` works mostly like the $conv_{AT-TLM \rightarrow TDF}$ process in section 4.5.1.2 after the *Freeze*: It copies the values from the transaction to the FIFO (which is the internal token queue in the implementation, updates the phase and responds via the backward path.

There is no ERROR response resulting from a full FIFO, as the internal token queue is unbounded. However, `process_transaction()` might return error responses via the backward path because of other reasons (e.g. streaming width does not fit or wrong command), which are captured by the `check_transaction()` method.

Here we also see why the `end_resp_event` (triggered in in the Listing AT-conv-transport in the `nb_transport_fw` method)is needed: When the initiator returns `TLM_ACCEPTED`, this also means that the delay is realized there, after which the initiator completes the transaction with a final call on the forward path. The converter has to to wait for this call, and as `wait()` cannot be called, it just changes the sensitivity of `process_transaction()` to the `end_resp_event` and returns. When `process_transaction()` is called next, the original sensitivity is restored[1].

We finally look at the `processing()` method of the TDF sub-module. Here the token from the internal token queue are copied to the TDF signal. If there are not enough token, this is handled according to the corner-case option `option`.

```
void processing(){
  int num_el = (ready_queue.size() < rate) ? ready_queue.size() : rate;
  int i;
  for(i=0; i < num_el; i++){                    // writes as many token from the internal
    last_written_element = ready_queue.front(); // token queue to the output as possible
    out.write(last_written_element , i);        // ideally, <rate> many
    ready_queue.pop();
  }
  if(num_el < rate){                            // corner case handling
```

[1] Unless the next transactions can then already be processed, which might again lead to a `TLM_ACCEPTED` response by an initiator

```
    switch(option){
      case CONSTANT:  last_written_element = empty_fifo_dummy;
      case HOLD:      dummy_token_used += rate - num_el; break;
      case ERROR:     SC_REPORT_ERROR("TLM_2_TDF::conv_AT", "Not enough elements to write to TDF");
    }
  }   // write dummy token to the remaining slots:
  for( int j=i ; j < rate; j++) out.write(last_written_element , j);
}
```

Listing 6.4: The processing method of the TDF sub-module

6.1.2 LT-TLM converter

The prototype of the LT-TLM→TDF converter constructor is exactly the same as for the AT-TLM converter, as can be seen in the Listing below. The main differences to the AT-TLM converter[2] are

- The implementation of the transaction queue: We use a custom reduced transaction type instead of making full transaction copies and store them in a custom queue which is like a PEQ without the events.
- Retrieval of the transactions: This is done by the TDF-submodule

```
TLM_2_TDF::conv_LT(
          sc_core::sc_module_name nm,
          sc_core::sc_time _accept_delay,
          sc_core::sc_time _latency_per_byte,
          sc_core::sc_time tdf_timestep,
          unsigned int _rate = 1,
          conv_option  _option = ERROR,
          unsigned int _initial_queue_token = 0,
          T _empty_fifo_dummy  = (T)(0)
          )
```

Listing 6.5: Constructor prototype of the TLM-LT→TDF converter

The `b_transport()` method of the LT-converter acts exactly like the $conv_{LT-TLM \to TDF}$ converter in section 4.5.1.1: It puts a copy of the transaction to the transaction queue and returns with an OK response. The only difference is that we don't actually copy the whole transaction, but only the parts which we need, i.e. the time stamp and the data. For this, a struct `smalltrans` is used:

```
struct smalltrans{
  sc_core::sc_time  timestamp;
  unsigned int      length;
  unsigned char*    data;
};
```

and the internal transaction queue uses this struct. The next main difference is that the transactions stay in this queue as long as possible. Only if the TDF sub-module needs token to write, it removes a transaction from the queue and copies the token to the FIFO, which in the implementation is like in the AT case an internal token queue. Therefore the TDF sub-module acts like the $fetch^{p}_{TLM}$ process in section 4.5.1.1 (see Figure 4.7) there. Listing 6.6 shows the `processing()` method of the TDF sub-module.

[2] beside the general differences regarding the protocol

```
void processing(){
  while(   (ready_queue.size() < rate)                  // not enough token in token queue?
        && (trans_queue.size() > 0)                     // ...and transactions in queue?
        && (trans_queue.top().timestamp <= get_time())  // ...which can already be used?
       ){
    T value;                    // Then we copy the data from the first transaction in the queue
    for(unsigned int i = 0; i < trans_queue.top().length; i= i + sizeof(T) ){
      memcpy(&value, (trans_queue.top().data + i), sizeof(T));
      ready_queue.push(value);  // to the internal token queue
    }
    trans_queue.pop();          // remove the transaction
  }
  // the rest is the same as for the TDF sub-module in the AT-TLM case
}
```

Listing 6.6: The processing method of the TDF sub-module of the LT converter

This concludes the description of the SystemC TLM→TDF converters. Apart from certain technical adaptations, they implement the conversion strategies as discussed in section 4.5.1.

6.2 TDF→TLM conversion

We now turn to processing read-transactions that want to read token from a TDF input stream. Similar to the converters in the previous section, the TDF-submodules of the TDF→TLM hold internal unbounded token queues to copy the incoming TDF token, So this is the FIFO we actually read from. As discussed in section 4.5.1, the corner-case here is that there not enough token to read for the transaction, which gives rise to five different ways to deal with this. These can be chosen in the TDF→TLM converter implementations using the enum type below:

```
enum TDF_2_TLM::conv_option{
    ERROR = 0,  // throw an error
    HOLD,       // use the last value that could be read
    CONSTANT,   // use a constant value
    WAIT,       // wait until there are enough token
    DISMISS     // return the transaction with an error
  };
```

We also need to employ a transaction queue. Note, however, that this queue is functionally[3] only needed for of the `WAIT` option, when the converter waits until the TDF side has supplied the missing token. This will work, however, very differently now for the two coding styles.

Also, there is a use-case driven functionality added: The converters can be configured such that read-transactions always leave a certain constant amount of token on the queue, even when they are copied to the transaction. The reason for this is that many digital signal processing applications process signals piecewise, where the pieces often overlap. In section 6.3, we will see an example for this.

6.2.1 LT-TLM converter

Listing 6.7 shows the constructor prototype, which is basically the same as the previous two in section 6.1, except for the added `_overlap` parameter.

[3] In AT-TLM it is needed in general for protocol reasons.

```
TDF_2_TLM::conv_LT(
          sc_core::sc_module_name nm,
          sc_core::sc_time _accept_delay,
          sc_core::sc_time _latency_per_byte,
          sc_core::sc_time tdf_timestep,
          unsigned int _rate = 1,
          conv_option  _option = ERROR,
          T _no_data_dummy  = (T)(0),
          unsigned int _overlap = 0     // new parameter to support DSP use cases
         )
```

Listing 6.7: Constructor prototype of the TDF→LT-TLM converter

In section 4.5.2.1 we concentrated on the `WAIT` option handling, and this option was actually the only reason we needed to employ a transaction queue for the $conv_{TDF \to LT-TLM}$ process. If the `WAIT` option is not chosen, the most straightforward approach in LT-TLM is to wait until the transaction is valid (i.e. wait for the delay in the SystemC implementation), and then process the transaction using the token which are present in the internal token queue[4].

If the `WAIT` option is chosen, the idea is to freeze (i.e. suspend the transaction) until there are enough token available. In 4.5.2.1 we used a special transaction queue which also held references to the different process copies which processed the transactions. If there were enough token present, the $read_F$ process invoked the process of the oldest token in the queue.

In the actual SystemC implementation, we have to use a trick to achieve this behavior, since we can't simply freeze the `b_transport()` method in an indeterminate manner - we have to wait for a certain event. Therefore we use a data structure for the transaction queue (which in this case is a `std::priority_queue<>`) that stores the transaction (and its delay) together with an event:

```
struct transSuspend{
  tlm::tlm_generic_payload* transp;              // pointer to the transaction
  sc_core::sc_time          timestamp;           // the timestamp
  sc_core::sc_event*        event;               // event to notify if *transp can be processed
};

class comparator{           // comparator function for the std::pririty_queue<transSuspend>
  public: comparator(){}    // entries are sorted according to time stamp
  bool operator() (const transSuspend& lhs, const transSuspend& rhs){
    return (lhs.timestamp >= rhs.timestamp);
  }
};
```

Listing 6.8: Helper structure for suspending transaction in LT-TLM

The next helper class manages the priority queue with the `transSuspend` entries:

```
SC_MODULE(transactionSuspendManager){
  std::priority_queue<transSuspend, std::vector<transSuspend>, comparator> trans_queue;

  SC_CTOR(transactionSuspendManager){}

  // put a transaction in the queue:
  sc_core::sc_event* checkIn(tlm::tlm_generic_payload& trans, sc_core::sc_time& delay){
    transSuspend* susTrans = new transSuspend();  // new transSuspend entry
    susTrans->transp = &trans;                    // copy the pointer
    susTrans->timestamp = sc_core::sc_time_stamp() + delay;   // compute and store time stamp
```

[4]Which might not be enough, but then the corner-case handling comes into play

```
    susTrans->event = new sc_core::sc_event();              // create and store the event
    trans_queue.push(*susTrans);                            // push it all to the queue
    return susTrans->event;                                 // return the event
  }

  // function that checks if there are transactions in the queue;
  // if so, it also suspends the transaction passed and returns the corresponding event
  sc_core::sc_event* checkSuspend(tlm::tlm_generic_payload& trans, sc_core::sc_time& delay){
    if(trans_queue.size() == 0 )  // no transactions in queue?
      return NULL;                // => transaction can be processed right away
    return checkIn(trans, delay); // otherwise we check it in
  }

  // this method gets passed the number of token in the internal token queue
  // ...and un-suspends transactions accordingly
  void unsuspend_if_ready(unsigned int _token_ready){
    unsigned int token_ready = _token_ready;
    unsigned int token_needed;
    sc_core::sc_time now = sc_core::sc_time_stamp();
    sc_core::sc_time time_suspended;
    tlm::tlm_generic_payload* transp;

    bool go_on = true;
    while(go_on && trans_queue.size() > 0){
      transp = trans_queue.top().transp;      // look at the oldest transaction
      token_needed = transp->get_data_length()/transp->get_streaming_width();
      if( token_needed <= token_ready             // enough token available and...
          && trans_queue.top().timestamp <= now){ //...timestamp is not in the future?
        // => un-suspend the transaction
        trans_queue.top().event->notify(sc_core::SC_ZERO_TIME); // notify event
        wait(sc_core::SC_ZERO_TIME);  //...and wait. After that wait, the transaction is done
        delete(trans_queue.top().event);  // delete old event
        trans_queue.pop();                //...and remove the entry
      }
      else go_on = false;
      token_ready -= token_needed;  // adjust number of token ready for next round
    }
  }
};
```

Listing 6.9: Module to mange transactions suspension in LT-TLM

With the `transactionSuspendManager` in Listing 6.9, we effectively implement a priority-queue like mechanism, but for loosely-timed targets (and interconnects), However, while the AT-TLM PEQs always use a timed notifications on the same event, the `transactionSuspendManager` uses one event for each transaction which can be notified individually to react to certain situations.

The next Listing shows the `processing()` method of the TDF sub-module. It effectively implements the $read_F$ process in section 4.5.2.1.

```
void processing() {
  for(int i=0; i<rate; i++){          // store token in internal queue
    read_queue.push_back(in.read(i));
  }   // call the transactionSuspendManager with an update on the token count:
  suspendedTransactions->unsuspend_if_ready(read_queue.size());
} // suspendedTransactions is the transactionSuspendManager instance
```

Listing 6.10: The processing method of the TDF sub-module

Listing 6.11 now shows the relevant parts of the `b_transport()` method of the converter:

```
virtual void TDF_2_TLM::b_transport( tlm::tlm_generic_payload& trans, sc_core::sc_time& delay ){
  //check_transaction() etc (ommitted, similar to the previous converters)
  sc_core::sc_time call_time = sc_core::sc_time_stamp();  // current time
  sc_core::sc_time suspend_time = sc_core::SC_ZERO_TIME;  // variable to store suspension time
  if(option == WAIT){         // WAIT option selected
    sc_core::sc_event* waitevent = suspendedTransactions.checkSuspend(trans, delay);
      if(waitevent != NULL){  // if previous transactions were suspended
        wait(*waitevent);     //...this one will also be suspended
      }
      else if( num_token > reader.read_queue.size() ){  // if there are not enough token
        sc_core::sc_event* waitevent = suspendedTransactions.checkIn(trans, delay);
        wait(*waitevent);                               //... we also suspend
      }
      suspend_time = sc_core::sc_time_stamp() - call_time;  // compute suspension time
    }
    else if(num_token > reader.read_queue.size()){  // corner case handling
        if( option == ERROR ){                      //...for ERROR option
          SC_REPORT_ERROR("TDF_2_TLM::conv_LT", "Not enough elements to read from TDF");
        }
        if( option == DISMISS ){                    //...and DISMISS option
          trans.set_response_status( tlm::TLM_GENERIC_ERROR_RESPONSE );
          return;
        }
    }
    delay += accept_delay + latency_per_byte*data_len;  // add latency to delay
    if(suspend_time > sc_core::SC_ZERO_TIME)            // and if we suspended...
    {
      if(delay > suspend_time) delay -= suspend_time;   //...subtract the suspension time
      else delay = sc_core::SC_ZERO_TIME;
    }
    // token_time is a variable which holds the timestamp of the token last read
    sc_core::sc_time latest_token_time = token_time + num_token*reader.timestep;
    if (sc_core::sc_time_stamp() + delay < latest_token_time ){ // adjust delay to the
      delay = latest_token_time - sc_core::sc_time_stamp();     //...time stamp of the token
    }
    trans.set_response_status( tlm::TLM_OK_RESPONSE );
    dequeIterator it = reader.read_queue.begin();
    unsigned int new_dummy_token_used = 0;
    for(unsigned int i=0; i<data_len; i = i+str_wid ){  // copy token from queue
      if(it != reader.read_queue.end()){
        last_read_element = *it++;
        token_time += reader.timestep;
      }
      else {  // this can happen only for options HOLD and CONSTANT
        new_dummy_token_used++;
        if(option == CONSTANT) last_read_element = no_data_dummy; // nothing to do for option = HOLD
      } // copy token to transaction:
      memcpy(data_ptr+i, reinterpret_cast<unsigned char*>(&last_read_element), str_wid);
    }
    // now we delete only the token not in the overlap, but also consider eventual dummy token
    unsigned int effective_overlap = 0;
    if(overlap > new_dummy_token_used) effective_overlap = overlap - new_dummy_token_used;
    it -= effective_overlap;
    token_time -= effective_overlap*reader.timestep;
    reader.read_queue.erase(reader.read_queue.begin(), it);
  }
```

```
}
```

Listing 6.11: `b_transport` method of the TDF→LT-TLM converter

The `b_transport()` method in Listing 6.11 is the most complicated process of the SystemC TLM↔TDF converter implementation. The reason for this is the elaborate transaction suspension mechanism, the need to adapt the delay according to the token time stamp and the suspended time, and the management of the overlap.

6.2.2 AT-TLM converter

The constructor prototype of the TDF→AT-TLM converter is shown below, it takes again the same parameters as the LT-TLM version.

```
TDF_2_TLM::conv_AT(
          sc_core::sc_module_name nm,
          sc_core::sc_time _accept_delay,
          sc_core::sc_time _latency_per_byte,
          sc_core::sc_time tdf_timestep,
          unsigned int _rate = 1,
          conv_option  _option = ERROR,
          T _no_data_dummy  = (T)(0),
          unsigned int _overlap = 0
         )
```

Listing 6.12: Constructor prototype of the TDF→AT-TLM converter

Regarding the conversion processes, the situation is as in section 4.5.2.2, i.e. they behave very similar to the LT conversion processes. The $conv_{TDF \to AT-TLM}$ (Figure 4.10) is just a version of the $conv_{TDF \to LT-TLM}$ process (Figure 4.9) which also handles the additional state updates.

Regarding the SystemC implementation, these similarities persist. While the `nb_transport_fw` method of `TDF_2_TLM::conv_AT` looks just like the one of `TLM_2_TDF::conv_AT`, the corresponding `process_transaction()` looks very much like the `b_transport()` method in Listing 6.11, except of course the additional call to the backward path.

To suspend a transaction to wait for token to arrive, `process_transaction()` now simply leave the transaction for a longer time in the PEQ we are using anyway to implement the standard protocol:

```
if( option == WAIT && num_token > reader.read_queue.size() ){
  time_before_suspend = sc_core::sc_time_stamp(); // store time of suspension
  next_trigger (reader.token_event);              // continue when there are enough token
  return;
}
```

Listing 6.13: TDF→AT-TLM converter handling for the `WAIT` option

Similarly to Listing 6.10, the TDF sub-module notifies the `reader.token_event` when there are enough token for the oldest transaction to be processed. The other corner-case options are handled the same way as in Listing 6.11.

6.3 Application Example

In this section we give an application example for the TLM↔TDF converters. The idea is to demonstrate functionality and show some examples on what kind of conclusions can be drawn with the help of the converters. To this end, we use an N^{th} order finite impulse response (FIR) filter [RG75] implemented by a DSP with multiple cores. An FIR filter computes a weighted sum

$$y_n = \sum_{i=0}^{N} c_i x_{n-i} \tag{6.1}$$

where the c_i are the filter coefficients, and x_k, y_k is the k^{th} input- and output-token, respectively. That is, to compute m output token, we have to read $N + m$ input token.

Suppose we want to split the computation such that DSP_1 computes the first m output token $y_n, ..., y_{n+m-1}$, and DSP_2 computes the next m output token $y_{n+m}, ..., y_{n+2m-1}$. Obviously, DSP_1 has to read the $N+m$ input token $x_{n-N}, ..., x_{n+m-1}$. However, if DSP_2 would take the next $N+m$ input token $x_{n+m}, ..., x_{N+n+2m-1}$ to compute $y_m, ..., y_{2m-1}$, then

$$y_{n+m+j} = \sum_{i=0}^{N} c_i x_{N+n+m+j-i} \text{ for } j = 0, ..., m-1$$

however, according to 6.1, we must have

$$y_{n+m+j} = \sum_{i=0}^{N} c_i x_{n+m+j-i} \text{ for } j = 0, ..., m-1$$

Therefore, DSP_2 needs to read the token $x_{n+m-N}, ..., x_{n+2m-1}$, which means that the token $x_{n+m-N}, ..., x_{n+m-1}$ have to be used in *both* computations. In other words, there is an overlap of N token.

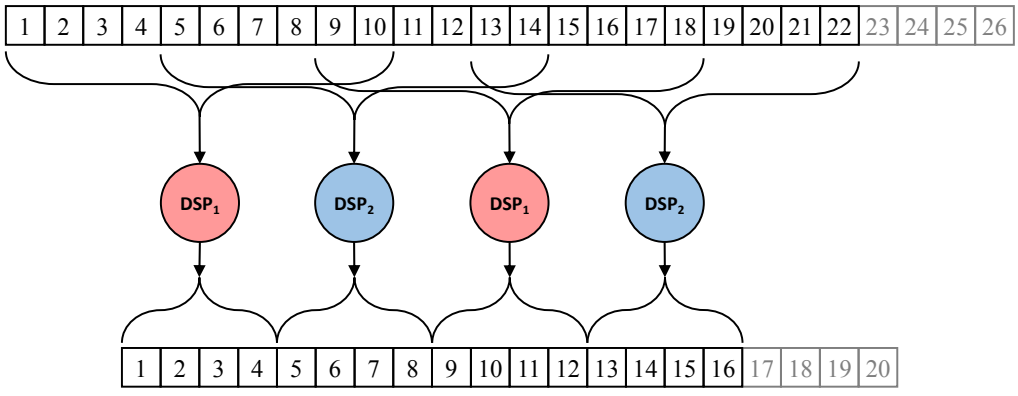


Figure 6.1: Example of the FIR parallelization with $N = 6$, $m = 4$, and two DSPs

Figure 6.1 shows an example where two DSPs compute $m = 4$ output token alternately using an FIR filter of order 6. This yields a simple parallelization approach: Each DSP core reads a number of $N + m$ token from the input, but *consumes* only m token, while *producing* m output token.

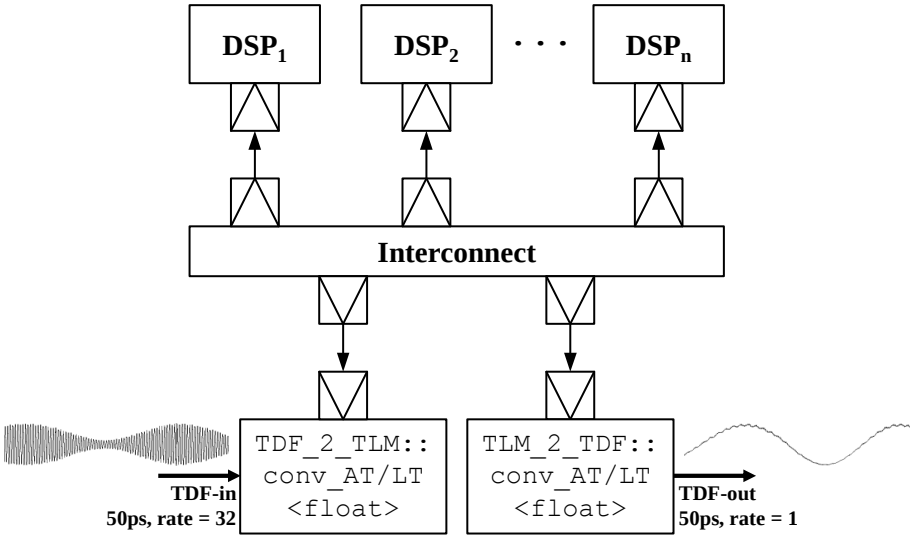


Figure 6.2: Overview of the simulation setup

In our application example, we use an FIR filter of order 31 to implement a low pass filter which de-modulates an incoming signal which uses amplitude modulation (AM). The frequency of the carrier signal is 2.4 GHz, and the payload signal is under 100 MHz, so we designed the filter to cut off at 500 MHz.

The modulated TDF input signal (of type `float`) is read by a TDF→TLM converter with a data rate of 32 token (as we need to read at least that many), with a time step of 50 ps. The DSPs issue READ-transactions for $4 \cdot t$ bytes (as a float is 4 bytes wide), with a streaming width of 4 bytes, to read $t > 32$ token. When a DSP gets the data it transforms it back to a series of floats, takes their absolute value, and then processes these values with the FIR algorithm. The resulting $t - 31$ token are then send as transactions with $4(t - 31)$ bytes to the TLM→TDF converter, which streams them to the output with a data rate 1. The time step of the TDF input- as well as the TDF output-signal is 50 ps. Figure 6.2 shows an overview of this setup. Figure 6.3 shows an example trace of (from top to bottom) the payload signal, the modulated signal, the DSP output, and, for comparison, the result of filtering the modulated signal with a simple 1^{st} order lowpass filter modeled in SystremC AMS.

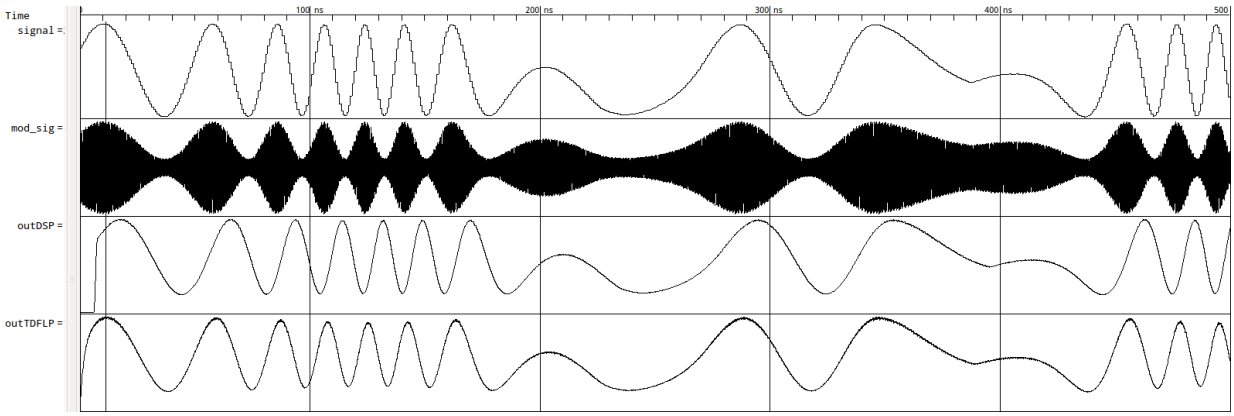


Figure 6.3: Example simulation trace

The converters model a 5 ps accept delay, plus a 0.5 ps delay per byte. The TDF→TLM converter

uses the `WAIT` option for the empty FIFO corner case. This makes the most sense since for example, replacing missing token with dummies (with options `CONSTANT` or `HOLD`) and then applying the filter on this corrupted signal would also corrupt the output signal. The TLM→TDF converter uses the `HOLD` option, since this shows very well if the DSPs can't hold up with the pace of the TDF token stream, although the `CONSTANT` option would be also suitable for this.

The DSPs can be configured as LT or AT initiators, and model in both cases an internal latency of 2 ns per transaction, plus 50 ps per token. Note that there is no synchronization mechanism in place. The idea is now to look how setups with a different number of DSP cores and different values of m perform. Let's start with 6 DSPs reading 64 token per transaction (i.e. $m = 33$) in a LT model. The simulation trace of the first 100 ns looks like this:

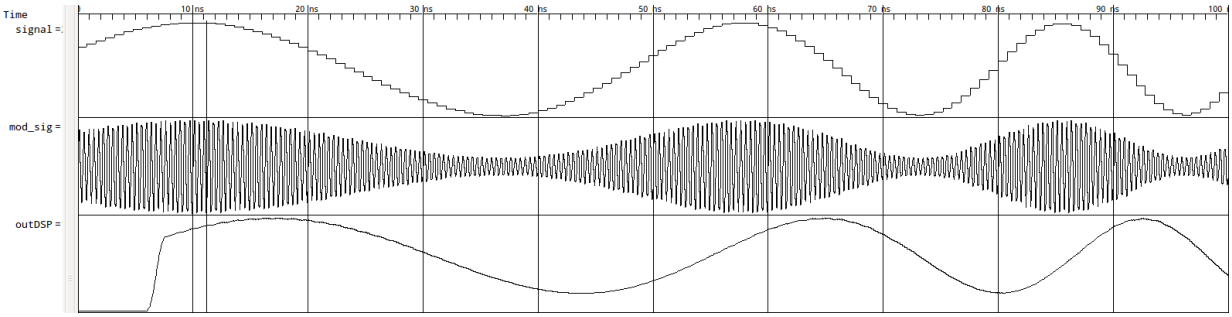


Figure 6.4: Simulation trace of LT model with 6 cores and $m = 33$

The result looks fine; there is a certain delay which is to be expected as there must be (in this case) at least a delay of 64 token, i.e. 3.2 ns. And with an FIR Filter of order 31, a 32 token (i.e. 1.6 ns) delay (or phase shift) is inevitable anyway. However, this delay looks bigger. A look at the reports of the converters will give more insight:

```
conv-LT-TDF-TLM-1 REPORT on 1212 transactions
    Elements in queue: 35
    past maximum: 95
    suspended transactions: 760 (62.7063%)
    additional delay: 2589384 ps for 1212 transactions (100%), 2136 ps on average
conv-LT-TLM-TDF-1 REPORT on 39897 token:
    Elements in queue: 21
    past maximum: 53
    dummy token used: 124
```

Listing 6.14: Converter reports of LT model with 6 cores and m = 33.

The TDF→TLM conversion report states that there have been a maximum of 95 token in the input queue, which is an indicator of how large a buffer in the implementation has to be. About 63% of the transaction where suspended, and each transaction got an average of 2.1 ns additional delay (over the whole 1212 transactions), because of suspension and/or to adaption to larger time-stamps of TDF token. Note that some of the code used to compute these statistics has been removed from the Listings in section 6.1 and section 6.2 in order to provide a more concise presentation.

The main information in the TLM→TDF conversion report is the number of 124 dummy tokens used, which translates to the 6.2 ns delay we can see in Figure 6.4. The transaction suspensions and the delays added by the TDF→TLM converter tells us that the 6 cores seems to be more than capable to handle the workload. Let's see if we can get by with four cores (Figure 6.5):

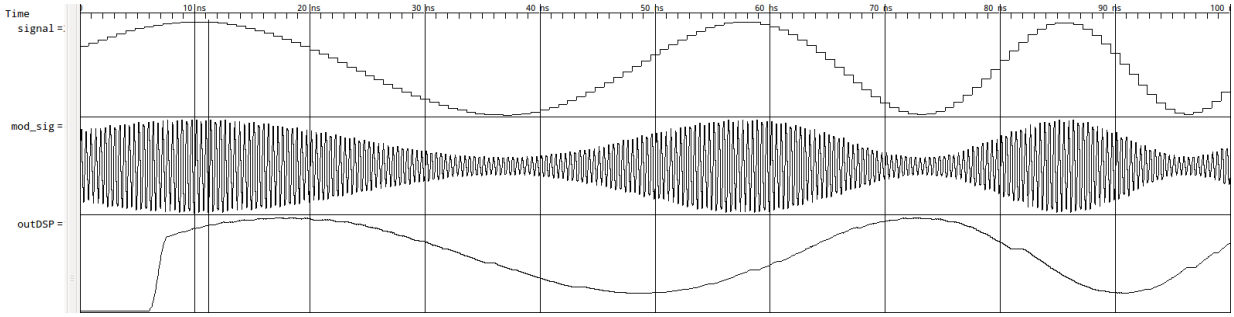


Figure 6.5: Simulation trace of LT model with 4 cores and $m = 33$

Apparently not. The TDF→TLM conversion report below tells us that only 4 transaction where suspended, which would be the first four transactions issued by the four DSP cores at time 0, as they had to wait for input token to arrive. Instead, more than 5000 token piled up in the internal queue during the 2μs simulated time. Consequently, the TLM→TDF converter had to insert more than 5000 token, and the output signal moves more and more out of phase with the original payload signal.

```
conv-LT-TDF-TLM-2 REPORT on 1052 transactions
    Elements in queue: 5315
    past maximum: 5351
    suspended transactions: 4 (0.380228%),
    additional delay: 6700 ps for 4 transactions (0.380228%), 6 ps on average
conv-LT-TLM-TDF-1 REPORT on 34683 token:
    Elements in queue: 33
    past maximum: 52
    dummy token used: 5350
```

Listing 6.15: Converter reports of LT model with 4 cores and m = 33.

However, 5 DSP cores seem just right for the payload:

```
conv-LT-TDF-TLM-2 REPORT on 1211 transactions
    Elements in queue: 68
    past maximum: 108
    suspended transactions: 5 (0.412882%),
    additional delay: 794812 ps for 1211 transactions (100%), 656 ps on average
conv-LT-TLM-TDF-1 REPORT on 39897 token:
    Elements in queue: 21
    past maximum: 53
    dummy token used: 124
```

Listing 6.16: Converter reports of LT model with 5 cores and m = 33.

Only one more transaction got suspended, and an average of 656*ps* delay was added to a transaction. The dummy token inserted by the TLM→TDF converter is 124 again, as it was with 6 cores. The trace result looks again as in Figure 6.4.

When using $m = 97$ (i.e. reading 128 token instead if 64), it turns out that only 3 DSPs are needed, in which case the TDF→TLM converter adds about 2.4 ns on average per transaction. However, the delay grows to 188 token, as that many dummy token were added by the TLM→TDF converter. Again, this is to be expected with the larger packet size. However, less computation power is needed, since the overlap is smaller, i.e. there is a trade-off between preserving computation power and delay.

Next suppose we have 8 cores at our disposal and want to reduce the delay as much as possible. How low can we go with m? We start with $m = 29$, which means a read transaction reads 60 token, and go down in decrements of 4 token. Table 6.1 shows the results:

Table 6.1: Results for 8 cores with varying m

m	29	25	21	17	13
avg. added delay (ps)	835	827	1497	133	2
dummy token used	120	115	111	108	7760

The added delay-spike for $m = 21$ is unexpected, but the result is clear none the less: The 8 cores can take the workload of $m = 17$, i.e. 48 byte packet size, but for $m = 13$ they cannot hold up with the incoming data stream.

Finally, we have a look how the AT converters perform. To this end, we take the three optimal cases from the previous LT analysis and simulate them with an AT model:

Table 6.2: Results for the three optimal LT cases simulated with an AT model

case	3 cores, $m = 97$	5 cores, $m = 33$	8 cores, $m = 17$
avg. added delay (ps)	1182	1352	658
dummy token used	128	64	48

That is, the number of dummy token inserted by the TLM→TDF converter corresponds exactly to the packet sizes, which means that the AT models with our AT converters were able to simulate these cases with the minimum delay possible.

7 Conclusion

Choosing a MoC (or several MoCs) for a certain use case (or modeling goal) is *relatively* straightforward most of the time. It might be dictated by the domain of the system under consideration, together with the targeted abstraction level, or simply by the simulation environment which is habitually used. With MoC conversion, this is not true. Often, the exact conversion semantics depend on the case at hand.

In this thesis we analyzed several MoC conversion directions with the goal to implement them in SystemC. To this end, a formalism was used which modeled process control and computation in an abstract manner, with no predefined notion of communication (other than variable access) whatsoever. We defined several MoCs in this formalism, and then founded the MoC conversion analysis on these abstract MoC definitions.

Based on this analysis, SystemC converters were implemented. Converterchannels on the one hand, which also used already existing converter facilities in SystemC AMS, while also providing automatic conversion capabilities and data type conversion. And the TLM↔TDF converters on the other hand, which provide configurable conversion means to stream TDF signals to TLM models and vice versa.

When looking at the Converterchannels, we have to conclude that in view of the numerous corner cases that can occur and the multiple options to choose from (especially regarding data type conversion, although we didn't go much into detail on this topic), the objective to provide *automatic* MoC conversion was missed to a certain degree. There seems to be no way to replace these options with automated choices, as there are too many factors to consider depending on the use cases and modeling goals. In its current state, the Converterchannels are more of an API to a configurable converter library than an automatic conversion facility. This converter library, however, covers the conversions supported in detail.

The TLM↔TDF converters developed (based on the formal analysis) provide an adequate way to connect TLM models to TDF models on a high abstraction level at an early design phase. For example, a real-life implementation of the model in section 6.3 could be implemented in different ways: There could be a task pulling data from an A/D converter to the memory of the system, where this (or another) task then also manages distribution of the data to the different cores including managing the overlaps. Or a hardware component or FPGA could implement an I/O component that behaves like the TDF→TLM converter internal queue, i.e. keeping a certain amount of token in the queue even when they are read.

Whatever the final implementation might be, the TLM↔TDF converters provide means to bring TDF models together with TLM models in a meaningful way for early software development on

virtual platforms. Especially the `WAIT`-style conversion semantics of the TDF→TLM converters are very suitable for this as they provide an inherent self-synchronization mechanism between TLM and TDF. Our simple example in section 6.3 also showed how the TLM↔TDF converters can provide statistics to help with architectural exploration (choice of the number of cores), as well as application development (choice of m).

Regarding the definition of the TLM MoCs in our formalism, an interesting result apart from MoC conversion was that the approximately timed coding style could be described in a simpler manner than in SystemC TLM2 because our formalism provided an alternative way to realize non-blocking access. It would be interesting to see if this could be implemented in SystemC as well. The most straightforward way (at least from a semantic perspective) to do so would be the implementation of a non-blocking variant of the `wait()` function. That way the non-blocking transport methods could be implemented in a more straightforward manner, without the need for PEQs or callback-functions. This and the temporal decoupling techniques of the loosely-timed coding style show that sacrificing global simulation control for simulation speed by self-controlled models is an important aspect of TLM2.

Literature

[Acc04] Accellera: *Verilog-AMS Language Reference Manual Version 2.2.* 2004. – http://www.verilog.org/verilog-ams/

[Acc13] Accellera Systems Initiative, AMS Working Group: *Analog/Mixed-signal (AMS) Language Reference Manual, Release 2.0.* 2013

[AND09] ANDRES Consortium: D1.6b Overall Modelling Framework for ANDRES / EC FP6 Project Deliverable. 2009. – Forschungsbericht

[Ayn09] Aynsley, John: *OSCI TLM-2.0 Language Reference Manual.* http://www.systemc.org: Open SystemC Initiative (OSCI), 2009

[Bac05] Bacic, Marko: On Hardware-in-the-Loop Simulation. In: *Proceedings of the 44th IEEE Conference on Decision and Control – European Control Conference (CDC-ECC)* IEEE, 2005, S. 3194–3198

[BAGK07] Burton, Mark ; Aldis, James ; Günzel, Robert ; Klingauf, Wolfgang: Transaction Level Modelling: A Reflection on what TLM is and how TLMs may be classified. In: *Proceedings of the Forum on Design Languages (FDL)*, 2007, S. 92–97

[BEG+11] Barnasconi, Martin ; Einwich, Karsten ; Grimm, Christoph ; Maehne, Torsten ; Vachoux, Alain: Advancing the SystemC Analog/Mixed-Signal (AMS) Extensions. In: *Open SystemC Initiative* (2011)

[BH98] Bellows, P. ; Hutchings, B.: JHDL – an HDL for reconfigurable systems. In: *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 1998, S. 175–184

[BSS07] Beltrame, Giovanni ; Sciuto, Donatella ; Silvano, Cristina: Multi-Accuracy Power and Performance Transaction-Level Modeling. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 26 (2007), Oct, Nr. 10, S. 1830–1842. – ISSN 0278–0070

[CG03] Cai, Lukai ; Gajski, Daniel D.: Transaction Level Modeling: An Overview. In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis.* New York, NY, USA : ACM, 2003 (CODES+ISSS '03). – ISBN 1–58113–742–7, S. 19–24

[CGO01] Cai, Lukai ; Gajski, Daniel D. ; Olivarez, Mike: Introduction of System Level Architecture Exploration using the SpecC Methodology. In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)* Bd. 5, 2001, S. 9–12 vol. 5

[Cho56] Chomsky, Noam: Three Models for the Description of Language. In: *Information Theory, IRE Transactions on* 2 (1956), Nr. 3, S. 113–124

[CL08] Cassandras, Christos G. ; Lafortune, Stephane: *Introduction to Discrete Event*

Systems. Springer, 2008

[DGH+08] Damm, Markus ; Grimm, Christoph ; Haase, Jan ; Herrholz, Andreas ; Nebel, Wolfgang: Connecting SystemC-AMS models with OSCI TLM 2.0 models using temporal decoupling. In: *Proceedings of the Forum on Design Languages (FDL)*, 2008, S. 25–30

[DHG08a] Damm, Markus ; Haase, Jan ; Grimm, Christoph: Co-Simulation of mixed HW/SW and Analog/RF systems at architectural level. In: *Behavioral Modeling and Simulation Workshop, 2008. BMAS 2008. IEEE International* IEEE, 2008, S. 84–89

[DHG+08b] Damm, Markus ; Haase, Jan ; Grimm, Christoph ; Herrera, Fernando ; Villar, Eugenio: Bridging MoCs in SystemC Specifications of Heterogeneous Systems. In: *EURASIP J. Embedded Syst.* 2008 (2008), S. 1–16. – ISSN 1687–3955

[DHHV07] Damm, Markus ; Haase, Jan ; Herrera, Fernando ; Villar, Eugenio: Using Converter Channels within a Top-Down Design Flow in SystemC. In: *Proceedings of the 15th Austrian Workshop on Microelectronics Austrochip*, 2007

[DMHG10] Damm, Markus ; Moreno, Javier ; Haase, Jan ; Grimm, Christoph: Using transaction level modeling techniques for wireless sensor network simulation. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)* European Design and Automation Association, 2010, S. 1047–1052

[DW08] Damm, Markus ; Wenninger, Joseph: D1.5b Modelling Extensions for Polymorphic Signals, final Library Elements / EC FP6 Project Deliverable. 2008. – Forschungsbericht

[ECN+01] Einwich, Karsten ; Clauss, Christoph ; Noessing, Gerhard ; Schwarz, Peter ; Zojer, Herbert: SystemC extensions for mixed-signal system design. In: *Proceedings of the Forum on Design Languages (FDL)*, 2001

[EGV+02] Einwich, Karsten ; Grimm, Christoph ; Vachoux, Alain ; Martinez-Madrid, Natividad ; Moreno, Felipe R. ; Meise, Christian: Analog Mixed-signal Extensions for SystemC. In: *White Paper and Proposal for the Foundation of an OSCI Working Group (SystemC-AMS working group)* (2002)

[EJL+03] Eker, Johan ; Janneck, Jorn ; Lee, Edward A. ; Liu, Jie ; Liu, Xiaojun ; Ludvig, Jozsef ; Neuendorffer, Stephen ; Sachs, Sonia R. ; Xiong, Yuhong: Taming Heterogeneity — The Ptolemy Approach. In: *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software* 91 (2003), January, Nr. 1, S. 127–144

[ES03] Einwich, Karsten ; Schwarz, Peter: SystemC-AMS Steps towards an Implementation. In: *Proceedings of the Forum on Design Languages (FDL)*, 2003

[FHL+01] Ferdinand, Christian ; Heckmann, Reinhold ; Langenbach, Marc ; Martin, Florian ; Schmidt, Michael ; Theiling, Henrik ; Thesing, Stephan ; Wilhelm, Reinhard: Reliable and Precise WCET Determination for a Real-Life Processor. In: *Embedded Software* Springer, 2001, S. 469–485

[Fis73] Fishman, George: *Concepts and Methods in Discrete Event Digital Simulation*. Wiley, 1973

[FN01] Fujita, Masahiro ; Nakamura, Hiroshi: The Standard SpecC language. In: *Proceedings of the 14th International Symposium on System Synthesis*, 2001, S. 81–86

[FO00] Frey, Peter ; O'Riordan, Donald: Verilog-AMS: Mixed-signal simulation and cross domain connect modules. In: *Proceedings of the IEEE/ACM International Workshop on Behavioral Modeling and Simulation (BMAS)*, 2000, S. 103–108

[GBA+07] Goderis, Antoon ; Brooks, Christopher ; Altintas, Ilkay ; Lee, Edward A. ; Goble, Carol: Heterogeneous Composition of Models of Computation / EECS

Department, University of California, Berkeley. 2007 (UCB/EECS-2007-139). – Forschungsbericht

[GBVE08a] Grimm, Christoph ; Barnasconi, Martin ; Vachoux, Alain ; Einwich, Karsten: An Introduction to Modeling Embedded Analog/Mixed-signal Systems using SystemC AMS Extensions. In: *DAC2008 International Conference*, 2008

[GBVE08b] Grimm, Christoph ; Barnasconi, Martin ; Vachoux, Alain ; Einwich, Karsten. *An Introduction to Modeling Embedded Analog/Mixed-Signal Systems using SystemC AMS Extensions.* June 2008

[GHP+09] Gerstlauer, A. ; Haubelt, C. ; Pimentel, A.D. ; Stefanov, T.P. ; Gajski, D.D. ; Teich, J.: Electronic System-Level Synthesis Methodologies. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28 (2009), Oct, Nr. 10, S. 1517–1530. – ISSN 0278–0070

[GL70] Gross, Maurice ; Lentin, André: *Introduction to Formal Grammars.* Springer, 1970

[GR94] Gajski, Daniel D. ; Ramachandran, Loganath: Introduction to High-Level Synthesis. In: *Design Test of Computers, IEEE* 11 (1994), Winter, Nr. 4, S. 44–54. – ISSN 0740–7475

[GSWB07] Grimm, Christoph ; Schroll, Rüdiger ; Waldschmidt, Klaus ; Brame, Florian: Mixed-level Simulation heterogener Systeme. In: *Proceedings of the ITG/GI/GMM-Workshop: Multi-Nature Systems*, 2007

[GVNG94] Gajski, Daniel D. ; Vahid, Frank ; Narayan, Sanjiv ; Gong, Jie: *Specification and Design of Embedded Systems.* Prentice Hall PTR, 1994

[GZD+00] Gajski, Daniel D. ; Zhu, Jianwen ; Domer, Rainer ; Gerstlauer, Andreas ; Zhao, Shuqing: SpecC: Specification Language and Methodology. (2000)

[Har78] Harrison, Michael A.: *Introduction to Formal Language Theory.* Addison-Wesley Longman Publishing Co., Inc., 1978

[HBH+99] Hutchings, B. ; Bellows, P. ; Hawkins, J. ; Hemmert, S. ; Nelson, B. ; Rytting, M.: A CAD suite for high-performance FPGA design. In: *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999, S. 12–24

[Her08] Herrera, F: *Heterogeneous Specification and Automatic Software Generation from SystemC for Embedded Systems*, Ph. D. thesis, University of Cantabria, Diss., 2008

[Hoa78] Hoare, Charles Antony R.: Communicating Sequential Processes. In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677

[Hof99] Hoffmann, Josef: *MATLAB und SIMULINK in Signalverarbeitung und Kommunikationstechnik.* Pearson Deutschland GmbH, 1999

[HOH+07] Herrholz, Andreas ; Oppenheimer, E ; Hartmann, Philipp A. ; Schallenberg, Andreas ; Nebel, Wolfgang ; Grimm, Christoph ; Damm, Markus ; Haase, Jan ; Brame, E ; Herrera, Fernando [u. a.]: The ANDRES project: Analysis and Design of Run-Time Reconfigurable, Heterogeneous Systems. In: *International Conference on Field Programmable Logic and Applications (FPL)* IEEE, 2007, S. 396–401

[HPJW+92] Hudak, Paul ; Peyton Jones, Simon ; Wadler, Philip ; Boutel, Brian ; Fairbairn, Jon ; Fasel, Joseph ; Guzmán, María M ; Hammond, Kevin ; Hughes, John ; Johnsson, Thomas [u. a.]: Report on the Programming Language Haskell: a non-strict, purely functional language version 1.2. In: *ACM SigPlan notices* 27 (1992), Nr. 5, S. 1–164

[HR13] Haetzer, Bastian ; Radetzki, Martin: Systemc Transaction Level Modeling with Transaction Events. In: *Proceedings of the Forum on Design Languages (FDL)*,

2013

[Huf14] Hufnagel, Simon: *Towards the Efficient Creation of Accurate and High-Performance Virtual prototypes*, Ph. D. thesis, University of kaiserslautern, Diss., 2014

[HV06] Herrera, F. ; Villar, E.: A Framework for Embedded System Specification under different Models of Computation in SystemC. In: *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006. – ISSN 0738–100X, S. 911–914

[HVG+07] Herrera, Fernando ; Villar, Eugenio ; Grimm, Christoph ; Damm, Markus ; Haase, Jan: A general Approach to the Interoperability of HetSC and SystemC-AMS. In: *Proceedings of the Forum on Design Languages (FDL)*, 2007, S. 32–37

[HVG+08] Herrera, Fernando ; Villar, Eugenio ; Grimm, Christoph ; Damm, Markus ; Haase, Jan: Heterogeneous Specification with HetSC and Systemc-AMS: Widening the Support of MoCs in SystemC. In: *Embedded Systems Specification and Design Languages*. Springer Netherlands, 2008, S. 107–121

[IEE05] IEEE: *SystemC™*. 2005. – IEEE Std. 1666

[IEE07] IEEE: *VHDL-AMS*. 2007. – IEEE Std. 1076

[ISS99] Isermann, R. ; Schaffnit, J. ; Sinsel, S.: Hardware-in-the-Loop Simulation for the Design and Testing of Engine-Control Systems. In: *Control Engineering Practice* 7 (1999), Nr. 5, S. 643 – 653. – ISSN 0967–0661

[Jan04] Jantsch, Axel: *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann Pub, 2004

[Jan06] Jantsch, A.: Models of Computation for Networks on Chip. In: *Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD)*, 2006. – ISSN 1550–4808, S. 165–178

[Kah74] Kahn, G.: The Semantics of a simple Language for Parallel Programming. In: Rosenfeld, J. L. (Hrsg.): *Information Processing*. Stockholm, Sweden : North Holland, Amsterdam, Aug 1974, S. 471–475

[Lee08] Lee, Edward A.: Cyber Physical Systems: Design Challenges. In: *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008, S. 363–369

[Lee09] Lee, Edward A.: Finite State Machines and Modal Models in Ptolemy II / EECS Department, University of California, Berkeley. 2009 (UCB/EECS-2009-151). – Forschungsbericht

[LG88] Lis, Joe S. ; Gajski, Daniel D.: Synthesis from VHDL. In: *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)* IEEE, 1988, S. 378–381

[LJ99] Lee, Edward A. ; John, II. *Overview of the Ptolemy Project*. 1999

[LLSV98] Liu, Jie ; Lajolo, Marcello ; Sangiovanni-Vincentelli, Alberto: Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator. In: *Proceedings of the 6th International Workshop on Hardware/Software Codesign*. Washington, DC, USA : IEEE Computer Society, 1998 (CODES/CASHE '98). – ISBN 0–8186–8442–9, S. 65–69

[LM87a] Lee, E. ; Messerschmitt, D.G.: Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. In: *Computers, IEEE Transactions on* C-36 (1987), Jan, Nr. 1, S. 24–35. – ISSN 0018–9340

[LM87b] Lee, Edward A. ; Messerschmitt, David G.: Synchronous Data Flow. In: *Proceedings of the IEEE* 75 (1987), Sept, Nr. 9, S. 1235–1245. – ISSN 0018–9219

[LMSG02a] Liao, Thorsten Grötkerand Stan ; Martin, Grant ; Swan, Stuart ; Grötker,

Thorsten: *System Design with SystemC*. Springer, 2002

[LMSG02b] Liao, Thorsten Grötkerand Stan ; Martin, Grant ; Swan, Stuart ; Grötker, Thorsten: *System Design with SystemC*. Springer, 2002

[LP95] Lee, Edward A. ; Parks, Thomas M.: Dataflow Process Networks. In: *Proceedings of the IEEE* 83 (1995), May, Nr. 5, S. 773–801. – ISSN 0018–9219

[LSU89] Lipsett, Roger ; Schaefer, Carl F. ; Ussery, Cary: *VHDL: Hardware Description and Design*. Bd. 12. Springer, 1989

[LSV98] Lee, Edward A. ; Sangiovanni-Vincentelli, Alberto: A Framework for comparing Models of Computation. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 17 (1998), Nr. 12, S. 1217–1229

[LT10] Lee, Edward A. ; Tripakis, Stavros: Modal Models in Ptolemy. In: *EOOLT* Citeseer, 2010, S. 11–21

[LWFK02] Loo, S.M. ; Wells, B.E. ; Freije, N. ; Kulick, J.: Handel-C for Rapid Prototyping of VLSI coprocessors for Real Time Systems. In: *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, 2002. – ISSN 0094–2898, S. 6–10

[MBP10] Martin, Grant ; Bailey, Brian ; Piziali, Andrew: *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, 2010

[MDG02] Mueller, Wolfgang ; Dömer, Rainer ; Gerstlauer, Andreas: The Formal Execution Semantics of SpecC. In: *Proceedings of the 15th International Symposium on System Synthesis*, 2002, S. 150–155

[MDH+12] Moreno, Javier ; Damm, Markus ; Haase, Jan ; Grimm, Christoph ; Holleis, Edgar: Unified and comprehensive electronic system level, network and physics simulation for wirelessly networked cyber physical systems. In: *Proceedings of the Forum on Design Languages (FDL)*, 2012, S. 68–74

[NP73] Nagel, Laurence W. ; Pederson, Donald O.: *SPICE: Simulation Program with Integrated Circuit Emphasis*. Electronics Research Laboratory, College of Engineering, University of California, 1973

[Ope04] Open SystemC Initiative, Synthesis Working Group: *SystemC synthesizable subset, Draft 1.1.18*. 2004

[Ope10] Open SystemC Initiative, AMS Working Group: *Standard SystemC AMS Extensions Language Reference Manual, Release 1.0*. 2010

[Pag96] Page, Ian: Closing the Gap between Hardware and Software: Hardware-Software Cosynthesis at Oxford. In: *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, 1996, S. 2/1–211

[PDBR04] Pasricha, Sudeep ; Dutt, Nikil ; Ben-Romdhane, Mohamed: Extending the Transaction Level Modeling Approach for Fast Communication Architecture Exploration. In: *Proceedings of the 41st Annual Design Automation Conference*. New York, NY, USA : ACM, 2004 (DAC '04). – ISBN 1–58113–828–8, S. 113–118

[Plu06] Plummer, Andrew R.: Model-in-the-Loop Testing. In: *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 220 (2006), Nr. 3, S. 183–199

[PPP95] Parks, Thomas M. ; Parks, Thomas M. ; Parks, Thomas M.: Bounded Scheduling of Process Networks. 1995. – Forschungsbericht

[PS04] Patel, Hiren D. ; Shukla, Sandeep K.: *SystemC Kernel Extensions for Heterogeneous System Modeling - a Framework for Multi-MoC Modeling and Simulation*. Kluwer, 2004. – I–XXXII, 1–172 S. – ISBN 978–1–4020–8087–6

[RE12] Reuther, Christiane ; Einwich, Karsten: A SystemC AMS Extension for Controlled Modules and Dynamic Step Sizes. In: *Proceedings of the Forum on Design*

Languages (FDL), 2012, S. 90–97

[RG75] Rabiner, Lawrence R. ; Gold, Bernard: Theory and Application of Digital Signal Processing. In: *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.* 1 (1975)

[RLSS10] Rajkumar, Ragunathan ; Lee, Insup ; Sha, Lui ; Stankovic, John: Cyber-physical Systems: The Next Computing Revolution. In: *Proceedings of the 47th Design Automation Conference*. New York, NY, USA : ACM, 2010 (DAC '10). – ISBN 978–1–4503–0002–5, S. 731–736

[RRG03] Rust, Carsten ; Rettberg, Achim ; Gossens, Kai: From High-Level Petri Nets to SystemC. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics* Bd. 2 IEEE, 2003, S. 1032–1038

[RSP+05] Rose, Adam ; Swan, Stuart ; Pierce, John ; Fernandez, Jean-Michel [u. a.]: *Transaction Level Modeling in SystemC*. www.systemc.org: Open SystemC Initiative, 2005

[Rua01] Ruan, Ken G.: Initialization of Mixed-signal Systems in VHDL-AMS. In: *Proceedings of the Fifth IEEE International Workshop on Behavioral Modeling and Simulation (BMAS)*, 2001, S. 53–58

[San03] Sander, Ingo: System Modeling and Design Refinement in ForSyDe. (2003)

[Sch97] Schaller, R.R.: Moore's Law: Past, Present and Future. In: *Spectrum, IEEE* 34 (1997), Jun, Nr. 6, S. 52–59. – ISSN 0018–9235

[Sch07] Schroll, Rüdiger: *Design komplexer heterogener Systeme mit Polymorphen Signalen*, Ph. D. thesis, Institut für Informatik, Universität Frankfurt, Frankfurt, Germany, Diss., 2007

[SGW05] Schroll, Rüdiger ; Grimm, Christoph ; Waldschmidt, Klaus: Verfeinerung von Mixed-signal Systemen mit polymorphen Signalen. In: *ANALOG'05* 46 (2005), S. 79

[SJ04] Sander, Ingo ; Jantsch, Axel: System Modeling and Transformational Design Refinement in ForSyDe [Formal System Design]. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 23 (2004), Nr. 1, S. 17–32

[SKR11] Salimi Khaligh, Rauf ; Radetzki, Martin: A Metamodel and Semantics for Transaction Level Modeling. In: *Proceedings of the Forum on Design Languages (FDL)*, 2011

[SMG05] Schlegel, Michael ; Müller-Glaser, Ing Klaus D.: *Mixed-Level-Simulation heterogener Systeme mit VHDL-AMS durch Multi-Architecture-Modellierung*, Chemnitz: Der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität Chemnitz, Diss., 2005

[VGE03a] Vachoux, A. ; Grimm, C. ; Einwich, K.: Analog and Mixed-signal Modelling with SystemC-AMS. In: *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on* Bd. 3, 2003, S. III–914–III–917 vol.3

[VGE03b] Vachoux, A. ; Grimm, C. ; Einwich, K.: SystemC-AMS Requirements, Design Objectives and Rationale. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2003. – ISSN 1530–1591, S. 388–393

[VGE05] Vachoux, A. ; Grimm, C. ; Einwich, K.: Extending SystemC to support Mixed Discrete-Continuous System Modeling and Simulation. In: *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, 2005, S. 5166–5169 Vol. 5

[VN93] Von Neumann, John: First Draft of a Report on the EDVAC. In: *IEEE Annals of the History of Computing* 15 (1993), Nr. 4, S. 27–75

[YWL02] Yan, Quan-Zhong ; Williams, John M. ; Li, Jim: Chassis Control System Development using Simulation: Software in the Loop, Rapid Prototyping, and Hardware

in the Loop / SAE Technical Paper. 2002. – Forschungsbericht

[Zav82] Zave, P.: An Operational Approach to Requirements Specification for Embedded Systems. In: *Software Engineering, IEEE Transactions on* SE-8 (1982), May, Nr. 3, S. 250–269. – ISSN 0098–5589

[Zei84] Zeigler, Bernard P.: *Multifacetted Modelling and Discrete Event Simulation.* Academic Press Professional, Inc., 1984

[ZG99] Zhu, Jianwen ; Gajski, Daniel D.: A Retargetable, Ultra-Fast Instruction Set Simulator. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 1999, S. 298–302

[ZPK00] Zeigler, Bernard P. ; Praehofer, Herbert ; Kim, Tag G.: *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic press, 2000