
H.W.M. van Moll

Supervisor: prof. dr. H. Corporaal
Coach: Victor Reyes (NXP)
Date: August 2008
Abstract

With the increasing complexity of System-on-Chip (SoC) designs there is a growing need for new design techniques that are able to cope with this trend. Additionally, the time-to-market windows of new products are becoming shorter, which calls for shorter design times and more efficient modeling techniques. In recent years Electronic System Level (ESL) design has been generally accepted as the next step in system design and particularly Transaction Level Modeling (TLM) is rapidly becoming the new way of designing complex systems.

However, the typical issue with TLM techniques is the accuracy vs. simulation speed trade-off. Most models that have a high degree of accuracy (approaching the Register Transfer Level (RTL)) are inherently slow due to clock sensitive processes. Similarly, models that can run at high simulation speeds are often modeled at abstraction levels that make them unsuitable for detailed architecture exploration or performance verification.

This thesis introduces a new methodology that enables the creation of fast and cycle-accurate protocol specific bus-based communication models, based on the upcoming TLM 2.0 standard from the Open SystemC Initiative (OSCI). An example of how to apply this methodology for a specific communication protocol is also presented in this thesis. Measurement results in terms of accuracy and simulation speed for a setup containing a cycle-accurate bus model with a simple initiator and target are also provided.
Acknowledgements

The work presented in this paper is part of the Virtual Prototyping, Modeling and Design (VMD) innovation project, led by Marleen Boonen of the System Design Technology (SDT) group of NXP semiconductors. The author would like to thank Victor Reyes for his guidance and the other members of the VMD project team and the SDT group for their support.
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Approximately-Timed</td>
</tr>
<tr>
<td>CA</td>
<td>Cycle-Accurate</td>
</tr>
<tr>
<td>DTL</td>
<td>Device Transaction Level</td>
</tr>
<tr>
<td>ESL</td>
<td>Electronic System Level</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GMFL</td>
<td>Generic Modeling Features Library</td>
</tr>
<tr>
<td>GP</td>
<td>Generic Payload</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>LT</td>
<td>Loosely-Timed</td>
</tr>
<tr>
<td>OSCI</td>
<td>Open SystemC Initiative</td>
</tr>
<tr>
<td>PV</td>
<td>Programmers View</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SCML</td>
<td>SystemC Modeling Library</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-a-Chip</td>
</tr>
<tr>
<td>UT</td>
<td>Un-Timed</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>VP</td>
<td>Virtual Prototype</td>
</tr>
</tbody>
</table>
# Table of Contents

1  Introduction .................................................................................. 1  
   1.1  System Level Design ............................................................... 2  
   1.2  Thesis Outline ........................................................................ 2  

2  Transaction Level Modeling .......................................................... 5  
   2.1  Related Work .......................................................................... 6  

3  TLM 2.0 ......................................................................................... 8  
   3.1  Coding styles ........................................................................... 9  
      3.1.1  Loosely timed coding style ............................................... 9  
      3.1.2  Approximately timed coding style ....................................... 11  
   3.2  Generic Payload & Extension Mechanism .................................... 12  

4  Methodology for Cycle Accurate Modeling with TLM 2.0 ................. 14  
   4.1  Protocol Analysis ..................................................................... 16  
      4.1.1  Identify Protocol Attributes ............................................. 16  
      4.1.2  Identify Timing Points ..................................................... 17  
      4.1.3  Identify Protocol State Machines ....................................... 19  
   4.2  TLM 2.0 Mapping .................................................................... 19  
      4.2.1  Protocol Specific TLM 2.0 Sockets ...................................... 19  
      4.2.2  FSM Mapping .................................................................. 21  
   4.3  Transactor Creation .................................................................. 22  
      4.3.1  Transactor Templates ........................................................ 22  
      4.3.2  Modeling Libraries ........................................................... 23  
      4.3.3  DTL transactors example .................................................. 23  
   4.4  Conclusion ............................................................................... 24  

5  Cycle Accurate TLM 2.0 DTL Transactors ...................................... 26  
   5.1  The DTL protocol .................................................................... 26  
      5.1.1  DTL generic payload extensions ........................................ 27  
      5.1.2  DTL timing phases ........................................................... 29  
      5.1.3  DTL State Machines .......................................................... 31  
   5.2  TLM 2.0 DTL Initiator Transactor ........................................... 34  
      5.2.1  Command Channel FSM: ................................................ 34  
      5.2.2  Write Channel FSM: ....................................................... 36  
      5.2.3  Read Channel FSM: ......................................................... 38  
      5.2.4  Buffer Management Channel FSM: ................................... 40  
   5.3  TLM 2.0 DTL Target Transactor ............................................. 41  
      5.3.1  Command Channel Thread ............................................... 42  
      5.3.2  Write Channel Thread ....................................................... 43  
      5.3.3  Read Channel Thread ....................................................... 44  
      5.3.4  Buffer Management Channel Thread ................................... 46  
   5.4  Conclusion ............................................................................... 47  

6  Cycle Accurate TLM 2.0 DTL Interconnect ...................................... 48  
   6.1  TLM 2.0 DTL Initiator Bus Adapter ......................................... 49
Illustration Index

Figure 1: Example of how transactors can be used to translate between different interfaces and abstraction levels. 3
Figure 2: Example of a bus model using bus adapters. ................................................................. 3
Figure 3: Abstraction levels and typical design use cases. .............................................................. 6
Figure 4: TLM 2.0 Overview (Source: TLM 2.0 User Manual) ......................................................... 8
Figure 5: TLM 2.0 use cases, coding styles and mechanisms. (Source: TLM 2.0 User Manual) .......... 9
Figure 6: Message sequence chart of a transaction between initiator and target using the phases of the approximately-timed base protocol................................................................. 12
Figure 7: Example of how IP can be reused for different communication protocol buses. ................. 14
Figure 8: Overview of the methodology to create protocol specific, cycle accurate TLM 2.0 interfaces and transactors................................................................. 15
Figure 9: For any protocol, the signal attribute groups, timing points and protocol state machines should be identified ................................................................. 16
Figure 10: Identification of timing points. The timing points are aligned to the VALID and READY handshake signals, which indicate when the information is made valid and when it gets sampled. The falling edges of these signals are not considered to be timing points, since no information is transferred at these points. ................................. 17
Figure 11: Timing points for the Command and Read groups of the DTL protocol ................................ 18
Figure 12: Code for the DTL write channel payload extension (valid in the WRITE_VALID phase). ........ 21
Figure 13: DTL channels and phase flows. The arrows indicate the possible flows through the phases that a transaction can take ................................................................. 22
Figure 14: Schematic of the initiator and target transactor templates connected together through a bus, using the TLM 2.0 interfaces (backward and forward path). ................................. 23
Figure 15: TLM 2.0 DTL Initiator Transactor ................................................................. 25
Figure 16: TLM 2.0 DTL Target Transactor ................................................................. 25
Figure 17: Finite State Machine for the initiator side of the DTL protocol ........................................ 32
Figure 18: Finite State Machine for the target side of the DTL protocol ........................................ 33
Figure 19: TLM 2.0 DTL Initiator Transactor Command Channel FSM Block ................................ 35
Figure 20: TLM 2.0 DTL Initiator Transactor Write Channel FSM Block ........................................ 37
Figure 21: TLM 2.0 DTL Initiator Transactor Read Channel FSM Block ........................................ 39
Figure 22: TLM 2.0 DTL Initiator Transactor Buffer Management Channel FSM Block .................. 40
Figure 23: TLM 2.0 DTL Target Transactor Command Channel Thread ......................................... 42
Figure 24: TLM 2.0 DTL Target Transactor Write Channel Thread .............................................. 44
Figure 25: TLM 2.0 DTL Target Transactor Read Channel Thread .............................................. 45
Figure 26: TLM 2.0 DTL Target Transactor Buffer Management Channel Thread ......................... 46
Figure 27: TLM 2.0 DTL Interconnect ......................................................................................... 48
Figure 28: Simple DTL extension router, instantiated with one initiator and one target connection .......... 51
Figure 29: Testbench to test the DTL interface and transactors ..................................................... 52
Figure 30: Waveform of simulation with a write and a read transaction. Delays are modeled to simulate processing delays ................................................................. 53
Figure 31: Speed measurements of the TLM 2.0 DTL test setup (Figure 29), using the configuration with no processing delays (worst case scenario). ................................................................. 54
Figure 32: Speed measurements of the TLM 2.0 DTL test setup (Figure 29), using the configuration which has a 2 cycle processing delay for each command and data phase ......................................................... 55
Figure 33: Testbench to test the DTL interconnect, which is used to connect the initiator and target sides. Traffic monitors are used to trace the DTL signals ................................................................. 56
Figure 34: Speed measurements of the TLM 2.0 DTL interconnect test setup (Figure 33), using the configuration with no processing delays (worst case scenario). ................................................................. 57
Figure 35: Speed measurements of the TLM 2.0 DTL interconnect test setup (Figure 33), using the configuration which has a 2 cycle processing delay for each command and data phase ................................................................. 57
Figure 36: DTL Signal to TLM 2.0 DTL extension mappings .......................................................... 61
Figure 37: SCML post to TLM 2.0 DTL Transactor (Initiator DTL Transactor) .............................. 62
Figure 38: TLM 2.0 DTL post to PV Transactor (Target DTL Transactor) ....................................... 63
Figure 39: TLM 2.0 DTL to DTL Extensions Transactor (Initiator DTL Bus Adaptor) ..................... 64
Figure 40: DTL Extensions to TLM 2.0 DTL Transactor (Target DTL Bus Adaptor) ..................... 65

VII
1 Introduction

The market of consumer electronics is ever increasing. With products such as cell-phones, cameras and portable music players being more popular than they have ever been, there is a lot of pressure on the manufacturers of these devices to come up with new and innovative products. Over the last couple of years there has been a trend to add more and more functionality to devices. As such, today’s cell-phones are capable of much more than just making phone calls. Music and video playback, digital camera, Personal Digital Assistant (PDA) functionalities and Internet connectivity are becoming increasingly more common features of modern cell-phones. In addition to the extra functionality that is added to these devices, there is also a need to make them small. Small devices are more portable, consume less power and are usually cheaper in hardware manufacturing costs.

The technology that has made it possible for designers of electronic systems to create these increasingly complex devices has evolved rapidly over the last decades. It started in the 1960’s with the introduction of Small Scale Integration (SSI), which allowed around 100 transistors to be placed on a single Integrated Circuit (IC). Since then, advances in chip manufacturing have resulted in an increased capacity of ICs. The prediction of Gordon Moore that the number of transistors that can be implemented in a single IC would double every 24 months has held true for a long time. In the 1980’s Very Large Scale Integration (VLSI) was introduced, which allowed for more than 10,000 transistors per IC in 1980 and up to several hundred million transistors per IC in 2007.

The increasing complexity of ICs has made it possible to create entire electronic systems on a single chip: the so called System-on-Chip (SoC). A SoC consists of several intellectual property (IP) cores such as processors (CPUs), memories, Input/Output hardware, etc. connected together by a communication structure such as memory mapped busses or a network-on-chip (NoC).

As the technology for creating more transistors per IC increases, the potential for more advanced and complex SoCs becomes available. However, the design of these new SoCs is also becoming more and more difficult and time-consuming. In addition to the increase in hardware complexity, the amount of software applications that run on a SoC is also rapidly growing. Next to this increase in SoC design complexity there is also a big time-to-market pressure for embedded system designers to get their products to the market as fast as possible.

It has become apparent that the classic design flow for developing electronic systems, which has worked well enough in the past, is no longer suitable for the complex SoC design processes we face today. As a result, a new design paradigm is needed to cope with the current SoC design requirements and time-to-market constraints.

In the last few years, there has been much discussion and research into finding a good solution for this new design paradigm. The general consensus among researchers in the SoC design field is that the traditional Register Transfer Level (RTL) at which ICs were usually designed, has become less desirable for designing large and complex systems. The idea of re-using IP cores is an important field of study. When IPs can be used for
multiple different SoC designs, it can significantly reduce the time that is needed to
develop new embedded systems.

1.1 System Level Design
A promising direction of research into new modeling strategies for complex SoC design
lies in the raising of the abstraction level at which SoC models are designed. Developing
systems at abstraction levels above RTL is generally known as System Level Design. The
range of abstraction levels above RTL are collectively called the Electronic System Level
(ESL).

Since many of the implementation details of a SoC can be abstracted away into simpler
representations of the systems functionality when moving to higher abstraction levels, the
design of models at the system level takes considerably less time than designing it at the
RTL level.

System level design has become an established approach at most of the world’s leading
SoC design companies, and is being increasingly used in embedded system design. As of
today there is still no real consensus on how many different abstraction levels above RTL
there are and how they are composed. It is however generally agreed that the system level
consists of more than one abstraction level and that there is a Transaction Level (TL) of
abstraction that exists above the RTL level. This transaction level is the basis for
Transaction Level Modeling (TLM), which is rapidly gaining popularity with system
designers as the new way of designing complex electronic systems.

1.2 Thesis Outline
Virtual Prototyping technologies based on SystemC Transaction Level Models are rapidly
settling down in the semiconductor and Electronic Design Automation industries as a
solution to cope with the increasing design complexity and shorter time-to-market of
today’s SoC. Virtual Prototypes (VP) can be applied successfully on different design use-
cases along the SoC development cycle. Some VP use-cases, such as software
development or functional verification, have simulation speed as their main requirement,
since millions of instructions have to be executed during simulations. Other use-cases,
such as performance verification, have cycle accuracy as the main requirement, since the
obtained results are used to sign-off the hardware platform. However, as of today, TLM
techniques can not create cycle-accurate models that are fast enough to fulfill the speed
requirements of all VP use-cases.

The new TLM 2.0 standard proposes new coding styles and modeling mechanisms,
which enable the refinement from high level loosely-timed communication down to
cycle-accurate protocol specific communication. However, this standard still has to be
proven in an industrial case and currently does not provide a methodology for creating
cycle-accurate models.

Moreover, with the introduction of the new standard it is important that the IPs that are
being used in VPs today can be easily placed into this new TLM 2.0 environment. It is
not practical to remodel all current IPs to use TLM 2.0 interfaces. Instead it is easier to
use transactors to translate from a non-TLM 2.0 interface to a TLM 2.0 interface.
Figure 1 shows an example of how transactors can be used to translate communication from IP modules that use a proprietary protocol interface into communication that uses a TLM 2.0 protocol interface. In addition to translating from one interface to another interface, transactors can also be used to move between different levels of abstraction. If for instance the bus in the example of Figure 1 has been modeled at the cycle-accurate level and the initiator and target IP modules are un-timed models, the transactors will have to translate the un-timed communication into cycle-accurately timed communication. The initiator and target IPs can be wrapped together with their respective TLM 2.0 transactors to form modules which have the same behavior as the original initiator and target IPs, but use the TLM 2.0 interface to communicate with the other components of the system. This approach promotes the re-use of IP components because the same initiator and target blocks can be re-used for different bus models by simply exchanging the bus and the transactors.

![Diagram of Figure 1](image)

Figure 1: Example of how transactors can be used to translate between different interfaces and abstraction levels.

For the modeling of interconnect components such as a busses, another set of transactors is required to translate between the TLM 2.0 interface that is used by the (wrapped) initiator and target components and the interface that is used internally in the arbitration and routing mechanism of the bus. Figure 2 shows an example of a bus model that uses such transactors, which will be called bus adapters. The bus arbitration and routing module can be wrapped together with the bus adapters to form a bus model that has TLM 2.0 interfaces.

![Diagram of Figure 2](image)

Figure 2: Example of a bus model using bus adapters.
The goal of this thesis project is to provide a methodology that enables the creation of fast and cycle-accurate bus-based communication models, based on the new TLM 2.0 standard. This methodology will make it possible to design the transactors that are needed to connect IP models with a generic (proprietary) interface to a cycle-accurate and protocol specific TLM 2.0 interconnect model in a structured way. Furthermore, this methodology can also be used to create the bus adapters that are needed to create cycle-accurate interconnect models. As part of the thesis project, the methodology has been applied for the Device Transaction Level (DTL) communication protocol. A set of TLM 2.0 DTL initiator and target transactors has been created as well as an example DTL interconnect model. The transactors and interconnect model have been tested using a test scenario with a system that contains a simple initiator and target connected together using the DTL interconnect model. This test scenario serves as a proof of concept that shows that the TLM 2.0 standard can be used to create cycle-accurate models that can run at simulation speeds which are considerably faster than traditional signal-level SystemC based implementations.

The remainder of this document is organized as follows. Chapter 2 gives a brief introduction on Transaction Level Modeling and abstraction levels. The new TLM 2.0 standard and some of its core features are discussed in Chapter 3. The methodology for creating cycle-accurate and protocol specific models using the TLM 2.0 standard is presented in Chapter 4. In Chapter 5, the details of the DTL protocol and how it is mapped to TLM 2.0 is discussed. This chapter also explains the initiator and target TLM 2.0 DTL transactors that were created. The bus adapters and the router that are part of the cycle-accurate TLM 2.0 DTL interconnect are discussed in Chapter 6. Speed and accuracy measurement results of the test scenario that was used to verify the TLM 2.0 DTL transactors and interconnect are presented in Chapter 7. Finally, conclusions are drawn in Chapter 8.
2 Transaction Level Modeling

The move to higher levels of abstraction in the design of system-on-chips has created a big interest in Transaction Level Modeling (TLM). This new way of modeling enables system designers to cope with the increasing design complexity and time-to-market pressures that they are dealing with today.

The name 'transaction level' is misleading in the sense that it does not denote a single level of detail, but rather a continuum of abstraction levels that each vary in the amount of functional or temporal detail they express [1]. TLM is a transaction-based modeling approach founded on high-level programming languages such as SystemC [2]. The term 'transaction' can have several meanings in the SoC world. The definition that is generally used in TLM context is that a transaction refers to the exchange or synchronization of data and/or control information between two components of a modeled and simulated system [3][4]. A data transaction can vary in size from a few bits to a series of words to complex data structures that are transferred over a bus between system components.

The details of the communication protocols that are used to carry out these transactions are often not implemented for TLM models where achieving high simulation speeds is the main focus, such as models that are used for embedded software development. However, for certain development use cases such as architecture exploration or hardware verification, more details are desired and more detailed TLM models are required.

The amount of timing information that is implemented in TLM models can also vary widely, from completely un-timed models to fully clock cycle accurate models. How much timing detail is needed for a model depends on the use case for which it is intended.

Figure 3 gives an overview of the different design abstraction levels in both the ESL and RTL domains. At higher abstraction levels, models contain less detail and can generally run at high simulation speeds. On the other hand, models created at a low level of abstraction contain many implementation details and are therefore more accurate, but run at slower simulation speeds.

The technique of TLM is relatively new and as such there are many different ideas on how to create transaction level models. This has led to different IP design companies adopting their own TLM implementations which are generally not compatible with each other. As a result, the integration of IP models provided by different IP providers into a system can become a difficult and time-consuming task due to incompatibilities in the interfaces of the IPs.

The solution for this problem is to use industry-wide standards for the creation of TLM models. This ensures that IP models will be compatible with each other as long as they all follow the same standards for implementing the interfaces. A lot of research is currently being done in coming up with these standards. The Open SystemC Initiative (OSCI) is a major contributor to the development of TLM standards. Recently, OSCI has proposed the TLM 2.0 standard as the interoperability standard for creating virtual platforms. More information on TLM 2.0 is given in Chapter 3 of this document.
Design Domain Abstraction Level Design Language

**Algorithmic Level (AL)**
Algorithm design

**Untimed (UT)**
Early SW development

**Loosely Timed (LT)**
SW development, SW performance analysis

**Approximately Timed (AT)**
HW exploration, HW/SW verification

**Cycle Accurate (CA)**
HW/SW performance validation

**Signal Level (SL)**
HW design and implementation

**SystemC**

---

**Figure 3:** Abstraction levels and typical design use cases.

2.1 Related Work

Transaction level models have been successfully used in design tasks ranging from embedded software development to functional verification. The requirements of the TLM model depend on the use-case for which it is needed. Cai and Gajski [5] present an overview of six models that vary both in the abstraction level of communication and computation from completely un-timed to fully cycle-timed. The models that are created with the modeling methodology presented in this paper have a close resemblance to the bus-functional model of [5] and the time-accurate communication model of [6]. SystemC [2] is the modeling language that is most widely used for TLM and is also employed in the methodology of this thesis. It has been shown that SystemC can be successfully used to create TLM models that are suitable for embedded software development [4].

For the exploration of on-chip communication performance, models which have a high degree of timing accuracy are needed. In [7] Pasricha et al. present a transaction-based abstraction level called Cycle Count Accurate at Transaction Boundaries (CCATB) that is shown to have a simulation speed-up of 55% over traditional cycle-accurate models.

A study of the trade-off between speed and accuracy of different TLM models for the AMBA bus has been presented by Schirner and Dömer in [8]. Their conclusion was that
while TLM models can gain a speedup of $10^4$ over a cycle accurate model, the accuracy drops to around 45% when there is significant traffic on the bus.

The new TLM 2.0 standard from OSCI [9] introduces a new way of creating TLM models, through the use of mechanisms like timing annotation and non-blocking transport APIs. The work shown in this thesis contributes to extend the scope of the standard by using its extension capabilities to create cycle-accurate protocol-specific models.
3 TLM 2.0

The TLM 2.0 standard [10] has been developed by the OSCI Transaction-level Modeling Working Group (TLMWG). This working group consists of 120 individuals from 27 organizations [11]. The TLM 2.0 standard was recently released at the 45th Design Automation Conference (DAC) [12].

The OSCI TLM 2.0 standard addresses several of the shortcomings of the TLM 1.0 standard [13] with respect to model interoperability and simulation speed. The standard focuses on systems that are based on memory-mapped buses, since these types of buses are most common in today's SoC's.

TLM 2.0 consists of an interoperability layer for bus modeling, a set of utility classes, the analysis interfaces and ports and the TLM 1.0 interfaces, which are included for backward compatibility. The interoperability layer consists of the generic payload, the base protocol phases, initiator and target sockets and the TLM 2.0 core interfaces, which are: the blocking transport interface, the non-blocking transport interface, the direct memory interface (DMI) and the debug transport interface.

The generic payload is a transaction object that supports the modeling of simple abstract memory-mapped buses. The attributes and phases of the generic payload can be extended to model protocol-specific communication through the generic payload extension mechanism. However, using the extension mechanism can lead to a reduction in interoperability since only the models that can handle these custom extensions can be connected together.

![TLM 2.0 Overview](Source: TLM 2.0 User Manual)
Figure 4 shows a diagram of how the TLM 2.0 classes are layered on top of the SystemC class library. In order to maximize the interoperability between models it is recommended to use the classes from the interoperability layer: the TLM 2.0 core interfaces, sockets, generic payload and the base protocol phases. In cases where the generic payload is inappropriate, it is possible to use an alternative custom transaction type with the core interfaces and sockets, but doing so will significantly restrict the interoperability of the models.

3.1 Coding styles

The TLM 2.0 standard defines a set of coding styles that each supports a range of abstraction levels. Basically these coding styles are a set of guidelines for using TLM 2.0 features to create models with a certain degree of communication timing accuracy. At the moment, the standard focuses on two main coding styles: the loosely-timed (LT) coding style and the approximately-timed (AT) coding style.

Figure 5 shows the two coding styles and the use cases for which these styles are commonly used. The TLM 2.0 mechanisms that are used by the coding styles are also shown in the figure.

The following sections briefly describe the LT and AT coding styles. For a more comprehensive description of the coding styles and the TLM 2.0 mechanisms, please refer to the TLM 2.0 user manual [14].

3.1.1 Loosely timed coding style

The loosely timed coding style uses the blocking transport interface to perform the transactions that are being send from an initiator (master) module to a target (slave) module. Each transaction that is made through this interface has two timing points. The first timing point is the transport call from the initiator to the target and the second
timing point is the return of the transport function from the target back to the initiator. These timing points are typically associated with the beginning of the request and response phases of the transaction.

Since each transaction in the loosely timed coding style only has two timing points, the amount of timing details that can be modeled is limited. Basically the only time parameter that can be modeled is the overall latency of the transaction (the time between the start and the end of the transaction). The loosely timed coding style supports the modeling of simple timers and interrupts that are needed to boot an operating system and run software on a virtual platform model. Therefore the loosely timed coding style is best suitable for software development use cases, where a virtual platform model of a SoC with a microprocessor is used to run the software. This software often includes some form of operating system.

The main goal of TLM models that are used for software development is to achieve high simulation speeds so that the software that runs on these models can be executed fast enough to make the software development process efficient. The loosely timed style is focused at optimizing simulation speed and uses several TLM 2.0 mechanisms that are useful in creating high speed models. The blocking transport interface limits each transaction between an initiator and a target module to a single function call.

In addition to the blocking transport interface, the loosely timed coding style also supports the Direct Memory Interface (DMI). The DMI enables an initiator to directly access a specific area of memory in a target, bypassing any interconnect components such as the system bus that a normal transaction would have to go through. This speeds up memory transactions by avoiding the communication function call overhead of the interconnect components.

Another TLM 2.0 mechanism that was designed to boost simulation speed and that can be used in the loosely timed coding style is temporal decoupling. Temporal decoupling enables certain processes in the system to run ahead of the global SystemC time of the system until they reach a point where synchronization with the system is needed. The amount of time that a process is allowed to run ahead of the rest of the system is called the quantum. Using temporal decoupling can significantly increase the simulation speed of a virtual platform model, because the amount of scheduling and process context switching in a SystemC simulation is reduced.

For example, software that is running on a processor model spends most of its time fetching and executing instructions from system memory, and only interacts with the rest of the system when an interrupt occurs in the system. If the interrupts occur at 1ms intervals, the processor could be allowed to run ahead of the SystemC simulation time with a quantum of up to 1ms. The accesses to the memory can be made using the DMI, but synchronization with the rest of the system only occurs at the rate of the interrupts. This means that the processor model does not have to be locked to the simulation time clock of the rest of the system, which can give a simulation speed improvement of up to 1000X over more traditional simulations without temporal decoupling and DMI.
3.1.2 Approximately timed coding style

While the loosely timed coding style is suitable for software development, software performance analysis and some high-level hardware analysis use cases, the limited amount of timing details in the transactions that can be modeled with this coding style makes it unsuitable for more detailed hardware architecture analysis, hardware verification and hardware performance analysis.

The approximately timed coding style adds more timing details to the transactions that are send between the components of a system by using multiple timing points (phases) for each transaction. For the approximately timed coding style, the TLM 2.0 non-blocking transport interface is used. The non-blocking transport interface differs from the blocking transport interface that is used in the loosely timed coding style in several ways. The most important features of the non-blocking transport are:

- A phase argument is send with the non-blocking transport call to indicate the current state of the transaction. Each transaction will go through a number of phases before it is completed.
- A target that receives a non-blocking transport call is not allowed to consume any time during the call and must return from the call without any delay. Communication delays may be annotated to the time parameter of the transport call to model how much time a transaction phase takes.
- Communication through the non-blocking transport interface is bi-directional. There is a forward path for transfers from the initiator to the target and there is a backward path for transfers that go from the target to the initiator.

The base protocol for the approximately-timed coding style of the TLM 2.0 standard defines four timing points for each transaction, which mark the begin request phase, the end request phase, the begin response phase and the end response phase. Figure 6 shows how the four phases of the base protocol can be used to model the request and response accept delays and the latency of the target. For specific communication protocols that require additional timing points, the base protocol can be extended with custom phases.

Because the approximately-timed coding style focuses on timing accuracy, the temporal decoupling mechanism is generally not used for this coding style and processes typically run in lock-step with the SystemC scheduler. Delays that are annotated to the non-blocking transport are consumed by using \texttt{wait}(delay) or \texttt{notify}(delay) statements to either stall the execution of a process or to notify an event.

Although the approximately-timed coding style enables the creation of more accurate models than the loosely-timed coding style, the amount of communication timing details that can be modeled is still limited when only the four transaction phases of the base protocol. For SoC design use cases such as detailed performance analysis and hardware/software validation that require even more accuracy the base protocol of the approximately-timed coding style is insufficient. Typically these use cases require the communication to be accurate at the level of the system clock. At this cycle-accurate level of abstraction, the approximately-timed base protocol must be replaced with a more specific communication protocol.
While no coding-style for cycle-accurate (CA) modeling is strictly defined within the TLM 2.0 standard, the modeling methodology presented in Chapter 4 of this document demonstrates that the TLM 2.0 standard provides the necessary mechanisms to create protocol-specific cycle-accurate models.

![Message sequence chart of a transaction between initiator and target using the phases of the approximately-timed base protocol](image)

**Figure 6: Message sequence chart of a transaction between initiator and target using the phases of the approximately-timed base protocol**

### 3.2 Generic Payload & Extension Mechanism

The generic payload is intended to improve the interoperability of memory-mapped bus models. The generic payload provides a general-purpose payload that guarantees immediate interoperability when creating abstract models of memory-mapped buses where the precise details of the bus protocol are unimportant. Furthermore, the generic payload can be used as the basis for creating detailed models of specific bus protocols, with the advantage of reducing the implementation cost and increasing simulation speed when there is a need to bridge between different protocols.

The generic payload is specifically aimed at modeling memory-mapped buses. It includes some of the attributes found in typical memory-mapped bus protocols such as command, address, data, byte enables, single word transfers, burst transfers, streaming, and response status. The generic payload may also be used as the basis for modeling protocols other than memory-mapped buses.

The generic payload does not include every attribute found in typical memory-mapped bus protocols, but it does include an extension mechanism so that applications can add their own specialized attributes.

The generic payload should be used together with the initiator and target sockets, which provide the forward and backward paths as well as a mechanism to enforce strong type checking between different protocols whether or not they are based on the generic payload.
The generic payload can be used with both the blocking and non-blocking transport interfaces. It can also be used with the direct memory and debug transport interfaces, in which case only a restricted set of attributes are used.

The generic payload extension mechanism permits any number of extensions of any type to be defined and added to a transaction object. Each extension represents a new set of attributes, transported along with the transaction object. Extensions can be created, added, written and read by initiators, interconnect components, and targets. The extension mechanism itself does not impose any restrictions. However, undisciplined use of this extension mechanism would compromise interoperability, so disciplined use is strongly encouraged.
4 Methodology for Cycle Accurate Modeling with TLM 2.0

The goal of the modeling methodology presented in this chapter is to provide a structured way to create custom interfaces that enable the modeling of cycle accurate communication for a given protocol using the mechanisms provided by TLM 2.0. An important part of this process is to identify and create extensions to the generic payload data structure and transaction phases defined in the standard. Furthermore, this methodology proposes a way of creating the transactors that are needed to connect the generic high-level interfaces that are typically used by the IP models to the cycle accurate TLM 2.0 interfaces. Using a generic interface for the IP models enables the clear separation of computation and communication within a system. This makes it possible to reuse the same behavioral IP model for different communication protocols and/or modeling styles. The separation, or orthogonalization of concerns [15] is one of the key aspects that enables the reusability of models.

Figure 7 shows an example of how the same initiator and target IP models can be reused for different bus protocols by separating the behavior of the IPs from the communication over the bus and using transactors to go from the generic protocol that is used by the IPs to the specific (cycle-accurate) protocol that is used in the bus. Since each bus protocol requires its own set of TLM 2.0 payload extensions, transaction phases and transactors, it is important to have a good methodology for creating these interfaces and transactors.

![Figure 7: Example of how IP can be reused for different communication protocol busses.](image)

In this work we have chosen to use the generic high-level communication interfaces provided by the SystemC Modeling Library (SCML) [16] from CoWare [17] for the initiator and target IP models. The SCML library provides the $\text{PVTarge}_\text{t\_port}$ (blocking) port which is used for the interface of target IP models and the $\text{scml\_port\_port}$ (non-blocking) port which is used for the interface of initiator IP models.
The methodology to create cycle-accurate interfaces and transactors is composed of three distinct steps: Protocol Analysis, TLM 2.0 Mapping and Transactor Creation. Each step is split further into one or more tasks. Figure 8 gives an overview of all the steps in the methodology.

The first step in the methodology is the analysis of the protocol that is to be modeled. The input for this step is the protocol specification datasheet. There are three relevant aspects of the protocol that need to be obtained: signal attributes, timing points and the protocol state machines. Signal attributes that are related, typically, are ordered in groups. Timing points indicate where certain signal groups become valid and are used as synchronization points between the initiator and target. Finally, the state machines capture the behavior of the protocol on the initiator and target sides. Figure 9 shows the signal attribute groups, timing points and state machines that should be identified for the protocol that is to be implemented.

The second step of the methodology is the mapping of the protocol attributes, timing points and state transitions to TLM 2.0 structures. Attributes should either be mapped directly to generic payload (GP) attributes or to custom payload extensions based on the extension mechanism provided by the standard. Timing points are translated into transaction phases. Related payload extensions and timing phases are grouped into channels. A communication protocol can have one or more channels. The output of this task is a pair of custom initiator and target TLM 2.0 sockets that form the initiator and target interfaces of the protocol. Another action carried out in this second step is the mapping of the protocol channels to the state transitions defined in the first step. This task results in a set of Finite State Machines (FSMs) that will later be used in the implementation of the initiator and target transactors.
A signal group contains the attributes that are send from initiator to target (or vice-versa) at a specific timing point.

Figure 9: For any protocol, the signal attribute groups, timing points and protocol state machines should be identified.

The final step is the creation of the transactors that enable the connection of the TLM 2.0 cycle-accurate protocol interface to IP models with the generic high-level interface. When implementing the transactors, the protocol sockets and state machines that were created in the previous step are used to complete a transactor template both for the initiator and target transactor. Specific modeling libraries (SCML and NXP proprietary) are used to create the generic interfaces and to model the behavior (FSMs) of the transactors. For the creation of the bus adapters that are needed to model interconnect components the state machines are not needed and a different set of templates is used.

These steps are meant as general guidelines that apply to most memory mapped bus protocols. Examples of protocols that are in the scope of this work are, for instance, AMBA AHB and AXI from ARM [18], the Open Core Protocol from OCP-IP [19] and the Device Transaction Level (DTL) [20] and Memory Transaction Level (MTL) [21] protocols from NXP.

The following sections will discuss the steps of the methodology in more detail. The DTL protocol from NXP is used as an example to further explain the steps of the methodology. Chapter 5 gives a more complete overview of the DTL protocol and how it was implemented using TLM 2.0.

4.1 Protocol Analysis

4.1.1 Identify Protocol Attributes

The goal of this task is to examine the protocol signals and group related signals together. These signal groups will form the basis for the TLM 2.0 generic payload attributes and extensions in the TLM 2.0 mapping step. The output of this task is a set of protocol
attribute groups. It is possible that there is some overlap of attributes between groups
(some signals may belong to more than one group), but in general the attribute groups
should be independent from each other.

For the DTL protocol the identification of attribute groups is fairly simple since the
protocol specification already defines a set of six signal groups [20]. The "system group"
contains the clock and reset signals. The "command group" consists of the address and
other control signals. The "write group" has the write data signals. The "read group"
contains the read data signals. There is a "write buffer management group" that has
signals to control the write buffer of the target. Finally, there is an "error/abort group"
that consists of signals that are used for signaling errors.

The clock and reset will not be part of the TLM 2.0 interface (clock sensitive process are
not used in the implementation) and therefore the "system group" will not be added to
the set of attribute groups. Signals from the "error/abort group" are closely related to
other signals of the protocol and can therefore be easily merged with the other groups,
and therefore no separate group is needed for those signals either. Table 1 shows the four
attribute groups that are identified for the DTL protocol.

Table 1: DTL Protocol Attribute Groups

<table>
<thead>
<tr>
<th>Attribute Group</th>
<th>Contains Signals From</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>command and error/abort group</td>
</tr>
<tr>
<td>Write</td>
<td>write and error/abort group</td>
</tr>
<tr>
<td>Read</td>
<td>read and error/abort group</td>
</tr>
<tr>
<td>Buffer Management</td>
<td>buffer management and error/abort group</td>
</tr>
</tbody>
</table>

Figure 10: Identification of timing points. The timing points are aligned to the VALID and
READY handshake signals, which indicate when the information is made valid and when it gets
sampled. The falling edges of these signals are not considered to be timing points, since no
information is transferred at those points.
4.1.2 Identify Timing Points

Timing points are defined as the moment where a signal or a group of signals becomes valid and can be sampled by the receiver. In principle, a timing point could be identified for each signal transition of the protocol. However, only a few are actually required to model the protocol with the desired degree of accuracy. Only those signal transitions that actually mark a point in time where information is being transferred or where the state of the protocol is changed are considered to be good timing points. Identifying any other timing points would only increase the complexity of the model, while not adding more accuracy. Figure 10 shows an example of identifying timing points.

Identifying the timing points for the DTL protocol is relatively straightforward since the protocol uses handshake signals to indicate when signal groups become valid and when the receiver has sampled the signals. This means that the timing points for the DTL protocol can be aligned to the rising edges of the handshake signals. Figure 11 shows how the timing points for the command and read groups are identified. Since the protocol defines that all the signals of a group become invalid in the clock cycle after the accept handshake signal was asserted, the falling edges of the signals (in cycles 2 and m+2 in Figure 11) are considered ignorable and are not identified as separate timing points.

For the write and buffer management groups, similar timing points are identified. A total of 8 timing points (2 for each signal group) are defined for the DTL protocol.

![Figure 11: Timing points for the Command and Read groups of the DTL protocol.](image-url)
4.1.3 Identify Protocol State Machines

Creating a finite state machine that captures the behavior of the communication protocol can be quite difficult, especially for complex protocols with many timing points and signal groups. It is therefore wise to limit the number of timing points and signal groups that were identified by the previous tasks to an absolute minimum, yet following the protocol specifications. Many protocols are designed to support a range of applications and it is often perfectly acceptable to only implement a subset of the protocol features. Doing so simplifies the design of the protocol FSM and the other steps of this methodology. However, it also limits the functionality of the protocol and may reduce model inter-operability. The bases for designing the FSM are the timing points of the protocol and how they relate to each other. Identifying the possible flows of timing points that are supported by the protocol helps creating the FSM. At this point the state machine can be described in a relatively abstract way. Further details will be added later, in the TLM 2.0 mapping step. It helps at this point to split the FSM into two separate state machines, one for the control of the initiator side of the protocol and one for the control of the target side of the protocol.

In the design of the FSM for the DTL protocol some limitations were placed on the protocol functionality to make the implementation easier. For instance, the possibility to cancel the transfer of control information by de-asserting the `cmd_valid` signal will not be supported by the TLM 2.0 implementation of the protocol, since it would introduce extra timing points and would make the FSM more complex. See Chapter 5.1.3 for more details on the DTL state machines.

4.2 TLM 2.0 Mapping

The first task of this step is to map attribute groups and timing points together and create the required TLM 2.0 payload and phase extensions. These custom extensions are grouped in channels and are used to create the protocol specific TLM 2.0 interface sockets. The second task is the creation of the initiator and target state machines.

4.2.1 Protocol Specific TLM 2.0 Sockets

The TLM 2.0 non-blocking transport interface implements the `nb_transport` method that takes three arguments: a transaction object, a phase object and a time object. The transaction and phase objects can be customized to allow protocol specific modeling. The default generic payload transaction object will be extended with payload extensions to enable the transfer of protocol specific attributes and the phase object will be replaced with a custom enumerator that captures the timing points (phases) of the specific protocol. Some of the protocol attributes that were identified in the first step can be mapped directly to attributes of the TLM 2.0 generic payload (e.g. the address and data signals), while other attributes will require a custom payload extension. Alternatively, it is possible to not use the generic payload attributes and only use custom payload extensions or create a completely customized transaction object class and not use the generic payload at all. However, not using the generic payload will seriously reduce the inter-operability of the model. The choices of how the protocol attributes are mapped depend on the implementation of the protocol transactors and interconnect.
If the timing points of the protocol were identified correctly there will be a direct one-to-one mapping of the timing points to transaction phases. Each timing point corresponds to one transaction phase. The phase of a transaction indicates which attributes of the generic payload and/or payload extensions are considered valid. If the protocol phases are in-line with the AT phases of the TLM 2.0 standard, the default `tlm_phase` object can be used. Otherwise a custom enumeration object must be defined that captures all of the transaction phases of the protocol.

When mapping the protocol attributes to payload extensions it is often a good idea to map each signal group to its own separate extension. This supports protocols with pipelining capabilities, because each extension can be processed and routed separately from the other extensions. The custom payload extensions (or custom transaction object class) and custom phase object are used to implement a protocol specific pair of initiator and target TLM 2.0 sockets.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD_VALID</td>
<td>Initiator has set the command channel signals</td>
</tr>
<tr>
<td>CMD_ACCEPT</td>
<td>Target has accepted the command channel signals</td>
</tr>
<tr>
<td>WRITE_VALID</td>
<td>Initiator has set the write channel signals</td>
</tr>
<tr>
<td>WRITE_ACCEPT</td>
<td>Target has accepted the write channel signals</td>
</tr>
<tr>
<td>READ_VALID</td>
<td>Target has set the read channel signals</td>
</tr>
<tr>
<td>READ_ACCEPT</td>
<td>Initiator has accepted the read channel signals</td>
</tr>
<tr>
<td>BUFFER_MGMNT</td>
<td>Initiator has set the buffer management channel signals</td>
</tr>
<tr>
<td>TAG_ACK</td>
<td>Target has set the <code>tag_ack</code> signal</td>
</tr>
</tbody>
</table>

There were 8 timing points identified for the DTL protocol in the previous step. These can be directly mapped to 8 transaction phases. Table 2 shows the phases that have been defined. The names of the phases indicate which set of signals is made valid or has been accepted.

A choice was made to give each phase its own generic payload extension that contains all the DTL signals that are valid during that phase. The attributes of the generic payload itself are not used. This maximizes the ability to process each channel separate from the other channels. For the DTL protocol, 7 payload extensions have been defined. The `READ_ACCEPT` phase is the only phase which does not have a payload extension because there is only one signal that is valid during this phase (the `read_accept` signal) and
this signal is assumed to always be high when the READ_ACCEPT phase is issued. Figure 12 shows a code example of the payload extension for the write channel.

```c
struct tlm2_dtl_wr_ext{
    tlm::tlm_extension<tlm2_dtl_wr_ext> {
        typedef unsigned char uint8;
        typedef unsigned long long uint64;
        // Constructor
        tlm2_dtl_wr_ext() : tlm2_dtl_wr_valid(false),
            tlm2_dtl_wr_data(0x00),
            tlm2_dtl_wr_mask(0xFF),
            tlm2_dtl_wr_last(false)
    };
    // clone method
    virtual tlm_extension_base* clone() const {
        // Create a deep-copy of the extension
    };
    // DTL signals
    bool tlm2_dtl_wr_valid;
    uint64 tlm2_dtl_wr_data;
    uint8 tlm2_dtl_wr_mask;
    bool tlm2_dtl_wr_last;
};
```

Figure 12: Code for the DTL write channel payload extension (valid in the WRITE_VALID phase).

4.2.2 FSM Mapping

In this step channels are mapped to the FSM that was created during the analysis of the protocol. During this task, more details are added to the protocol state machine. Furthermore, the state machine is split into an initiator FSM and a target FSM in preparation for the transactor implementation. When splitting the state machine into a initiator and a target part, it is important to look at which transaction phases belong to the initiator side and which belong to the target side. In order to achieve a higher degree of parallelism it is wise to further split both the initiator and target FSMs into multiple state machines, each dedicated to the processing of a different channel. When doing so, care should be taken that the state machines are be properly synchronized whenever there are inter-channel dependencies.

For the DTL protocol each channel has its own FSM on the initiator and target side of the channel. This means that there are a total of eight state machines for the entire protocol. Each of the DTL channels has two phases and one phase of every channel is on the initiator side, which means that each phase has its own FSM. Figure 13 shows the DTL channels and the phases that are associated with each channel.

The figure also shows the phase flows that can occur in a DTL transaction. The FSM for each phase processes the protocol attributes (payload extensions) that are associated with that phase. The state machines are designed to enable pipelining of phases. This means that once a channel has completed the phases of one transaction it can start sending
phases of the next transaction. There are some inter-channel phase dependencies (as can also be seen in Figure 13), which means that synchronization between the FSMs of the different channels is needed.

Figure 13: DTL channels and phase flows. The arrows indicate the possible flows through the phases that a transaction can take.

4.3 Transactor Creation

The final step is the creation of the initiator and target transactors that are needed to connect the core IP models with generic SCML interfaces to the protocol-specific TLM 2.0 interface. The basic structure of the transactors is always the same, therefore a set of transactor templates have been defined to simplify their modeling. The transactor templates are described in Chapter 4.3.1. Furthermore, extra modeling libraries are used in the implementation of the transactors; these are discussed in Chapter 4.3.2. The creation of the transactors for the DTL protocol is briefly discussed in Chapter 4.3.3. More details on the DTL transactors can be found in Chapter 5.

4.3.1 Transactor Templates

Figure 14 shows the templates for the initiator and target transactors. Each transactor has an active part that contains the protocol channel state machines which initiate the outgoing transactions and a reactive part that process the incoming transactions. The TLM 2.0 socket interfaces contain both a forward and a backward path. The forward path is used to send transactions from the initiator to the target, while the backward path is used to send transactions from the target to the initiator. Therefore, the active part of the initiator transactor drives the forward path and the active part of the target transactor drives the backward path. The reactive parts of both the initiator and target transactors implement the $\text{nb\_transport}$ interface method and handles incoming transport calls. Between the forward and backward paths of the transactors a synchronization layer is required. Finally, both initiator and target transactors have a generic high-level interface that handles the communication on the IP side of the transactors. These generic interfaces also have to be synchronized with the functionality of the transactors.
4.3.2 Modeling Libraries

Specialized modeling libraries are used for the implementation of the transactors. The SystemC 2.2 library forms the basis, since both transactors will be implemented as SystemC modules. The TLM 2.0 library [10] is required for the TLM 2.0 interfaces and sockets. For the generic interfaces on the initiator and target sides, the scml_post interface and the PVInitiator_port port from the CoWare SCML library [16] are used.

The implementation of the state machines for the behavior (active part) of the transactors is simplified through the use of the Generic Modeling Features Library (GMFL), which is a NXP proprietary modeling library built on top of SCML and contains dedicated classes for the modeling of Finite State Machines.

For the synchronization that is required between the TLM 2.0 forward and backward paths SystemC events and FIFOs are used.

4.3.3 DTL transactors example

The task of implementing the transactors for a specific protocol consists of taking the sockets and protocol state machines that were created in the TLM 2.0 mapping step and filling in the transactor templates (see Chapter 4.3.1). The active parts of the transactor templates are filled in with one or more FSM or thread processes, which control the channels of the protocol. Mechanisms for the synchronization between the different processes and between the active and reactive parts within the transactors need to be implemented, for example with events for control (phase) synchronization and fifos for data synchronization.

Figure 15 and Figure 16 show the block diagrams of the initiator and target transactors for the DTL protocol. Since the control of the phases in the target transactor is fairly simple for this protocol, standard threads are used to implement the protocol functionality in the target transactor. The initiator transactor has more complex control structures so they are implemented as FSMs (using the FSM class from the GMFL library). Synchronization between the state machines, threads and generic interfaces is
realized with SystemC events and FIFOs. The \texttt{nb\_transport()} function that implements the TLM 2.0 backward path in the initiator transactor notifies the correct FSM in the active part whenever a particular phase was received. For instance, when the CMD\_ACCEPT phase is received via the backward path, an event is notified. The FSM for the command channel will be waiting for this event and will continue execution after this event has been triggered.

In the target transactor, fifos are used to send the transactions (including the payload extensions) to the correct thread (based on the phase that was received) which will process the transaction further.

The generic interface in the initiator transactor implements the SCML post interface and does nothing more than place each incoming transaction request into a fifo. This fifo is then read by the command FSM that starts processing the transaction. In the target transactor, the generic interface uses a CoWare PV Target port connection to communicate with the target. The PV interface module of the target transactor handles the communication with the target by translating the DTL extensions that are send to this module by the channel threads in the active part into PV transport function calls. Additional read and write data buffers are implemented in this module to make the PV transactions more efficient by transferring bigger blocks of data at once and storing them in these intermediate buffers.

For more details on how these transactors work, please refer to Chapter 5 where they are explained in more detail.

4.4 Conclusion
This chapter introduced a new methodology for creating cycle-accurate protocol specific TLM 2.0 interfaces and transactors. The steps presented in this methodology can be used as guidelines for implementing cycle-accurate protocol interfaces with the TLM 2.0 standard. It should be possible to use the methodology to implement most memory mapped bus protocols, although at this point in time the methodology has only been applied to two protocols. For the DTL protocol, the steps of the methodology have been briefly explained in this chapter. More details on the DTL protocol and the TLM 2.0 cycle-accurate DTL implementation can be found in Chapter 5. The second protocol that has been implemented using the new methodology is the Memory Transaction Level (MTL) protocol from NXP. The MTL protocol closely resembles the DTL protocol and its TLM 2.0 implementation is beyond the scope of this thesis project.

The most complicated task in this methodology is the creation of the protocol state machines that are used in the initiator and target transactors. Especially for complex protocols such as DTL, it takes quite a bit of effort to correctly implement these state machines. Using the FSM classes from the GMFL library reduces the modeling effort. In the future, it might be interesting to examine the possibility of automatically generating the transactors. Some research on the automatic generation of TLM transactors has already been published. For instance, Bombieri and Fummi describe a methodology for automatic transactor generation in TLM by using Extended Finite State Machines.
(EFSMs) [22]. Possibly a similar approach could be used with the methodology that was presented in this chapter.

Figure 15: TLM 2.0 DTL Initiator Transactor.

Figure 16: TLM 2.0 DTL Target Transactor.
5 Cycle Accurate TLM 2.0 DTL Transactors

This chapter discusses the DTL protocol and the extensions for the TLM 2.0 generic payload and transaction phases that are needed to implement the DTL protocol at the cycle-accurate level using TLM 2.0 interfaces. Chapter 5.1 gives a short introduction to the DTL protocol and explains the payload extensions, transaction phases and abstract state machines that were created. Chapters 5.2 and 5.3 describe in detail the initiator and target transactors that were developed to implement the cycle-accurate TLM 2.0 DTL interfaces. The example diagram of Figure 1 shows how these transactors would typically be used in a system that has a TLM 2.0 DTL bus. The actual implementation of the DTL bus interconnect is discussed in Chapter 6.

5.1 The DTL protocol

The Device Transaction Level (DTL) protocol is a generic point-to-point data communication protocol that can be used for either direct IP to IP communication as well as IP to bus connections [20]. The size of the data that can be transferred with the DTL protocol can range from a single byte to complete blocks of data.

There are several different use cases for the DTL protocol. The four most common applications as they are identified in the DTL Protocol Specification document are:

- Memory Mapped Input/Output (MMIO)
- Memory Mapped Block Data Flow (MMBD)
- Memory Mapped Streaming Data Flow (MMSD)
- Peer-to-Peer Streaming Data Flow (PPSD)

The basic DTL protocol comprises 24 signals divided into 6 groups. For special addressing modes, two-dimensional block operations or secure transactions there are 6 more signals that comprise the extended DTL protocol. The complete DTL protocol therefore consists of 30 signals. All signals have to be driven to a logical '1' or '0' state. There are no tri-state signals in the DTL protocol.

Many of the DTL signals are optional and are not needed for certain specific applications. When a signal is not implemented, the default value for that signal is assumed.

The extensions that are made to the TLM 2.0 generic payload and phases in order to implement the DTL protocol are aimed to model DTL communication at the cycle-accurate level.

The standard TLM 2.0 generic payload (GP) that is used for simple communication protocols is insufficient to model DTL communication. Therefore it is necessary to add extensions to the GP to add the signals of the DTL protocol. Similarly, the timing phases of the base protocol that is defined in the TLM 2.0 standard are not enough to model communication cycle accurately. The most accurate coding style that the TLM 2.0 standard has is the approximately-timed coding style, which only contains four timing phases (see Chapter 3.1.2). This is not enough for cycle-accurate DTL, which means that additional timing phases are needed to accurately model DTL communication.
The following sections describe the extensions that have to be added to the TLM 2.0 generic payload and timing phases in order to model the DTL protocol cycle-accurately. These extensions can be obtained for any communication protocol using the methodology described in Chapter 4. Figure 36 in Appendix A gives a complete overview of all the TLM 2.0 DTL payload extensions and transaction phases.

5.1.1 DTL generic payload extensions

The TLM 2.0 generic payload DTL Extensions capture all of the DTL signals, with the exception of the DTL clock and reset signals. The clock signal is not needed because the TLM 2.0 DTL target and initiator IPs operate at the transaction level and therefore do not contain processes that are sensitive to a clock signal. A global reset signal could be implemented to initialize the TLM 2.0 DTL interface in the target and initiator but is generally not needed because the interfaces are initialized at the start of a SystemC simulation.

Seven DTL payload extensions have been defined that can be added to the default TLM 2.0 generic payload. Each DTL extension is associated with a DTL timing phase and contains a set of DTL signals which are valid for that timing phase. The signals of a DTL extension should only be read or written during the timing phase that corresponds with the extension, since outside that phase the validity of the signals is not defined.

Table 3 shows the DTL payload extensions and the timing phase in which they are considered to be valid. The table also shows which DTL signals are available in each extension. The payload extensions and timing phases which are related are grouped together in a channel. There are four channels for the TLM 2.0 DTL interface: Command Channel, Write Channel, Read Channel and Buffer Management Channel. See Appendix A for a more detailed mapping of the DTL signals to the channels and TLM 2.0 extensions. For more information on the DTL signals, please refer to the DTL protocol Specification document [20].

As can be seen in Table 3, there are two DTL signals that exist in more than one extension. These signals are err_wr (which is part of the tlm2_dtl_cmd_accept_ext, tlm2_dtl_wr_err_ext and tlm2_dtl_tag_ack_ext extensions) and err_rd (which is part of the tlm2_dtl_cmd_accept_ext and tlm2_dtl_rd_ext extensions). The reason that these signals are available in multiple extensions is because the DTL protocol indicates that these signals can be set at several points in time, depending on the application. For instance, targets that implement write buffers generally cannot indicate a write error (with the err_wr signal) until after the command has been transferred, and sometimes not until after all data has been transferred. On the other hand, targets that do not implement write buffers (e.g. MMIO ports) must indicate any write error during command transfer (see the DTL specification for more details). This means that the DTL protocol supports err_wr to be set during the CMD_ACCEPT, WRITE_ACCEPT and TAG_ACK timing phases. A similar story can be told for the err_rd signal. To avoid having to check more than one DTL extension in a timing phase, these two error signals are made available in each extension that is valid in a timing phase where those signals are allowed to be set.
Another thing that can be observed from Table 3 is that the **READ_VALID** timing phase does not have a payload extension associated with it. Since there is only one DTL signal that can be set in the **READ_VALID** timing phase (the `rd_accept` signal) and because this signal is always assumed to be high in this phase there is no need to have a payload extension for this phase. The `rd_accept` signal is still available as part of the tlm2_dtl_rd_ext extension but serves no real purpose and is only included to make the set of DTL signals complete.
5.1.2 DTL timing phases

The TLM 2.0 standard uses a phase argument that is passed with each transport call and indicates the state that the current transaction is in. A transition from one phase to another phase marks a timing point. During each phase the payload and extension attributes are considered to be stable. Any change to the transaction object should only become visible to other components at the next timing point.

For the DTL payload extensions this means that changes in the values of the DTL signals are only allowed to be made visible to outside components at the transition from one timing phase to another. From the DTL Protocol Specification documentation eight timing phases were identified that will make cycle-accurate modeling of DTL communication possible. The phases are: CMD_VALID, CMD_ACCEPT, WRITE_VALID, WRITE_ACCEPT, READ_VALID, READ_ACCEPT, BUFFER_MGMNT and TAG_ACK.

Each of these timing phases indicates a stage in the DTL protocol. The meaning of each timing phase is discussed next.

CMD_VALID

The Command Valid (CMD_VALID) timing phase is used to indicate the start of a transaction. During this phase the signals that are in the command group of the DTL protocol are valid. The generic payload extension that contains these signals is the Command extension (tlm2_dtl_cmd_ext). The CMD_VALID phase is always send by the initiator of the transaction and must always be the first phase of a transaction. Prior to making a call to `nb_transport`, the initiator must set the correct values for all the signals in the Command extension. After the `nb_transport` call has been made with the phase argument set to CMD_VALID the values of the Command extension can no longer be altered and are considered to be constant until the Command Accept (CMD_ACCEPT) phase is reached. The `cmd_valid` signal is assumed to be high during the CMD_VALID timing phase.

CMD_ACCEPT

The Command Accept (CMD_ACCEPT) timing phase indicates the end of a command transfer. This phase is used to signal an initiator that the signals of the DTL command group have been accepted by the target. The generic payload extension that accompanies this phase is the command accept extension (tlm2_dtl_cmd_accept_ext). A target must send the CMD_ACCEPT always after it has received a CMD_ACCEPT phase and before it sends a WRITE_ACCEPT or READ_VALID phase. The `cmd接受` signal of the command accept extension is always assumed to be high during the CMD_ACCEPT phase. The `err_wr` and `err_rd` signals that are also part of this extension can be used in this phase to signal that the target is unable to process a write or read transaction at this time.

WRITE_VALID

The Write Valid (WRITE_VALID) timing phase initiates the transfer of write data from the initiator to the target. During this phase, the signals from the DTL write group that
are part of the write extension (tlm2_dtl_wr_ext) are considered to be valid. The wr_valid handshake signal is assumed to be high in this phase. An initiator may only send the WRITE_VALID phase if a write transaction is being performed (the cmd_read signal was low during the CMD_VALID phase) and the phase must come after the CMD_ACCEPT phase, but may come before the CMD_ACCEPT phase. Only when the data that is to be written is actually valid and available at the initiator should the WRITE_VALID phase be issued. For burst transactions, every beat of the burst should start with a WRITE_VALID phase and be accepted by the target with a WRITE_ACCEPT phase.

**WRITE_ACCEPT**
A target uses the Write Accept (WRITE_ACCEPT) phase to indicate that the write data has been received and processed. The generic payload extension that is used in this phase is the write error extension (tlm2_dtl_wcercext). This extension contains the wr_accept signal, which is assumed to be high during the WRITE_ACCEPT phase; and the err_wr signal, which can be used to indicate a write error. A WRITE_ACCEPT phase must always be preceded by a WRITE_VALID phase from the initiator and must be send after the CMD_ACCEPT phase.

**READ_VALID**
The Read Valid (READ_VALID) phase is issued when read data is transferred from the target to the initiator. This phase can only occur if the transaction is a read transaction (the cmd_read signal was high during the CMD_VALID phase) and must come after the CMD_ACCEPT phase. The read extension (tlm2_dtl_rd_ext) is considered valid during this phase and contains the signals from the DTL read group. The rd_valid signal of the extension is always assumed to be high in this phase and the rd_accept signal is not used at all. Optionally the target can use the err_rd signal to indicate that a read error has occurred. The READ_VALID phase is send by the target when the data for the read transaction has become valid and available. Burst transactions will one READ_VALID phase for each beat of the burst. Every READ_VALID phase must be answered with a READ_ACCEPT phase from the initiator.

**READ_ACCEPT**
An initiator uses the Read Accept (READ_ACCEPT) phase to accept incoming read data. There is no generic payload extension associated with this phase, since the only DTL signal that would be transmitted during this phase, the rd_accept signal, will always assume to be high in this phase anyway. The READ_ACCEPT phase must only be send by an initiator after it has received a READ_VALID phase from the target.

**BUFFER_MGMT**
The Buffer Management (BUFFER_MGMNT) phase can be used by the initiator to perform buffer management operations on the target. There are three signals present in the buffer management (tlm2_dtl_buffer_mgmnt_ext) extension that is used during this phase. The flush signal is used to force the target to flush all the write data that still resides inside the write buffers. The tag signal can be used to tag the last data element
that was written to a write buffer. The \textit{abort\_all} signal will cause all outstanding
transactions to be aborted. When the initiator wants to perform a tag or flush operation,
the \texttt{BUFFER\_MGMT} phase should be issued directly after the \texttt{WRITE\_VALID} phase
that transferred the data that is to be tagged or flushed. The signals are considered to be
received by the target when the \texttt{WRITE\_ACCEPT} phase is send. Alternatively, the
\texttt{BUFFER\_MGMT} phase can be send after a write transaction (after the
\texttt{WRITE\_ACCEPT} phase), in which case the signals that are set in the buffer
management extension will be assumed to be high for a single clock-cycle.

\textbf{TAG\_ACK}

The Tag Acknowledge (TAG\_ACK) phase is send by the target when a write data
element that was previously tagged (in the \texttt{BUFFER\_MGMT} phase) has reached it’s
destination (no longer resides in write buffers). The extension that is valid during this
phase is the Tag Acknowledge (\texttt{tlm2\_dlt\_tag\_ack\_ext}) extension. The \texttt{tag\_ack} signal of
this extension is always assumed to be high during the TAG\_ACK phase. Furthermore,
the Tag Acknowledge extension also contains the \texttt{err\_wr} signal that can be used to
indicate that an error has occurred during the writing of the data. The TAG\_ACK phase
may only be issued after a \texttt{BUFFER\_MGMT} phase (with the \texttt{tag} signal set) was
received earlier. If the \texttt{BUFFER\_MGMT} phase was received after a WRITE\_VALID
phase but before the WRITE\_ACCEPT phase was send, the TAG\_ACK phase must be
send after the WRITE\_ACCEPT phase.

5.1.3 \textit{DTL State Machines}

Having identified the DTL phases and payload extensions, the next step is to define a set
of finite state machines (FSMs) that capture the DTL protocol. In order to create these
state machines it helps to see what possible phase flows can occur during a DTL
transaction. For the DTL protocol it is required that every transaction starts with the
transfer of the command signals. This means that the \texttt{CMD\_VALID} phase will always be
the first phase of a transaction. The possible phase flows for the DTL protocol are:

\textit{For write transactions:}

\begin{align*}
\texttt{CMD\_VALID} & \rightarrow \texttt{CMD\_ACCEPT} \rightarrow \texttt{WRITE\_VALID} \rightarrow \texttt{WRITE\_ACCEPT} \\
\texttt{CMD\_VALID} & \rightarrow \texttt{WRITE\_VALID} \rightarrow \texttt{CMD\_ACCEPT} \rightarrow \texttt{WRITE\_ACCEPT}
\end{align*}

\textit{Note:}

- For burst transactions the flow can be extended with more WRITE\_VALID and
  WRITE\_ACCEPT phases for as long as there is data to write.
- Write data can be tagged or flushed by inserting a \texttt{BUFFER\_MGMT} phase after
  a WRITE\_VALID phase.
- The TAG\_ACK phase to acknowledge that the tagged data has been processed can
come at any time after a \texttt{BUFFER\_MGMT} phase with the tag signal high has
  been issued.

\textit{For read transactions:}

\begin{align*}
\texttt{CMD\_VALID} & \rightarrow \texttt{CMD\_ACCEPT} \rightarrow \texttt{READ\_VALID} \rightarrow \texttt{READ\_ACCEPT}
\end{align*}
Note:
- For burst transactions the flow can be extended with more READ_VALID and READ_ACCEPT phases for as long as there is data to read.

Having identified the possible phase flows of a transaction, the state machines for the DTL initiator and target transactors can be determined. Figure 17 and Figure 18 show a high-level representation of the state machines for the initiator and target side of the DTL protocol, respectively. For the implementation of the state machines in the DTL transactors, each FSM is split into four separate state machines that each control one of the channels of the DTL protocol. More details on these FSMs can be found in Chapter 5.2 and Chapter 5.3.

![Finite State Machine for the initiator side of the DTL protocol.](image-url)
Figure 18: Finite State Machine for the target side of the DTL protocol.
5.2 TLM 2.0 DTL Initiator Transactor

The TLM 2.0 DTL Initiator transactor is used to connect an initiator IP component to a TLM 2.0 DTL cycle-accurate interconnect. This transactor has a SCML Post interface on the initiator side and a TLM 2.0 DTL interface on the side that is connected to the interconnect. A simplified diagram of this transactor was shown in Figure 15. For a more detailed schematic of the transactor including all the module ports and interfaces, please refer to Figure 37 in Appendix B.

The initiator transactor consists of three main parts. The “active part” contains the Finite State Machines that drive the DTL transactions on the TLM 2.0 forward path. The “reactive part” implements the TLM 2.0 `nb_transport( ...)` method that receives the incoming transactions via the TLM 2.0 backward path. The “synchronization” part handles all the synchronization between the active and reactive parts and the SCML Post interface.

The `nb_transport( ...)` function in the reactive part does nothing more than notify the `CmdAcceptEvent`, `WriteAcceptEvent`, `TagAckEvent` and `ReadValidEvent` events when the corresponding `CMD_ACCEPT`, `WRITE_ACCEPT`, `TAG_ACK` or `READ_VALID` phase is received.

Since there are four channels defined for the DTL protocol, the active part of the initiator transactor contains four separate state machines. Each FSM handles one of the DTL channels. The state machines are implemented using the GMFL_FSM_THREAD class from the NXP GMFL library [23]. Extra synchronization between the channel state machines is also implemented in the active part. The Command Channel FSM handles the incoming SCML transaction requests that are placed in the SCML Transaction FIFO by the SCML Post Interface. If the transaction is a read transaction it is placed in the Read transaction FIFO where it is read by the Read Channel FSM for further processing. Similarly, a write transaction is placed in the Write FIFO to be processed by the Write Channel FSM. The Buffer Management FSM handles the buffer management features (tag, flush and abort) of the DTL protocol.

The following sections describe the modules of the DTL Initiator Transactor in further detail. Refer to Figure 37 for more details on how the modules are connected together.

5.2.1 Command Channel FSM:

The Command Channel state machine (`TLM2_DTL_CMD_FSM`) module handles the transactions of the DTL Command Channel. The module is implemented using a GMFL_FSM_THREAD and has seven states. Figure 19 shows a diagram of this FSM module with the states, state transitions and external method calls made through the input and output ports.

The state machine starts in the `S_READ_FIFO` state, where first a `DTLTransactionStruct` structure is claimed from the TransactionPool. This structure is a convenience class that contains pointers to store the SCML transaction request as well as the TLM 2.0 generic payload and DTL extensions. Next, a `scml_transaction_request` is read from the SCMLTransactionFIFO fifo. If there are no
transaction requests pending, the state machine will block until the initiator has posted one.

In the next state, S_FILL_CMD_EXT, the DTL command extension (tlm2_dtl_cmd_ext) is filled in, based on the attributes of the SCML transaction request object. In addition to the DTL command extension, the attributes of the TLM 2.0 generic payload are also filled in. It is possible that a burst transaction needs to be split up into multiple smaller transactions, in which case only the first command phase of those transactions will contain all the command signals. Subsequent command phases will have the dtl_cmd_trans signal set low, indicating that the signals from the previous command phase should be used. If the transaction that is being processed is an abort transaction, the FSM moves to the S_ABORT state. Otherwise, the state machine goes to the S_STORE_READ state in case of a read transaction and to the S_STORE_WRITE state in case of a write transaction.

The S_ABORT state will cancel all outstanding and pending transactions. First it will write the current transaction object to the BufferTransaction value (where it will be read by the Buffer Management Channel FSM, which will send the abort transaction to the target). Secondly, the AbortEvent event is notified to signal the other state machines in the transactor that all transactions should be aborted. Finally, all the transactions that are in the SCMLTransactionFIFO fifo are read and immediately sent to the TransactionCompletion module.

![TLM 2.0 DTL Initiator Transactor Command Channel FSM Block](image-url)
The $S_{\text{STORE\_READ}}$ state writes the current transaction to the $\text{ReadFIFO}$ fifo. This fifo will be read by the $\text{TLM2\_DTL\_RD\_FSM}$ FSM, which will further process the read transaction.

The $S_{\text{STORE\_WRITE}}$ state writes the current transaction to the $\text{WriteFIFO}$ fifo. This fifo will be read by the $\text{TLM2\_DTL\_WR\_FSM}$ FSM, which will further process the write transaction.

After the transaction has been stored in either the $\text{ReadFIFO}$ or $\text{WriteFIFO}$ fifo, the Command Channel FSM reaches the $S_{\text{TRANSPORT}}$ state. In this state the $\text{nb\_transport\_fw(...)}$ method of the TLM 2.0 interface is called to transport the CMD\_VALID phase. The return value of the transport call is evaluated. A value of TLM\_COMPLETED will move the FSM to the $S_{\text{ERROR}}$ state because the DTL protocol does not allow transactions to be completed after only the CMD\_VALID phase has been send. If a return value of TLM\_ACCEPTED or TLM\_UPDATED is received, the $S_{\text{WAIT}}$ state is reached.

The $S_{\text{ERROR}}$ state should not be reached during normal operation of the transactor. Only a bad response from the $\text{nb\_transport(...)}$ call in the $S_{\text{TRANSPORT}}$ state can cause the FSM to reach this state. The SystemC simulation will be stopped when the $S_{\text{ERROR}}$ state is reached.

In the $S_{\text{WAIT}}$ state, the FSM thread first waits until the $\text{CmdAcceptEvent}$ event is triggered (which indicates that the CMD\_ACCEPT phase has been received). Once this event has been triggered the thread will wait for a fixed delay before continuing execution. This delay is configurable and indicates the amount of time there is in between two subsequent command transactions. The default value for this delay is one clock cycle and should not be set lower than that.

If the current transaction is a large burst transaction that needs to be split into multiple transactions, the Command FSM moves to the $S_{\text{FILL\_CMD\_EXT}}$ again to send another CMD\_VALID phase. Otherwise, the FSM moves back to the $S_{\text{READ}}$ state to process the next transaction from the fifo.

### 5.2.2 Write Channel FSM:

The Write Channel state machine ($\text{TLM2\_DTL\_WR\_FSM}$) handles the write data transfers of the DTL Write channel. The state machine is implemented as a GMFL\_FSM\_THREAD and consists of nine states. Figure 20 shows a diagram of this FSM module with the states, state transitions and external method calls made through the input and output ports.

The write channel FSM starts in the $S_{\text{READ\_FIFO}}$ state. In this state, the $\text{WriteFIFO}$ fifo is read to obtain a pointer to the next write transaction. After the fifo has been read, the state machine will wait for a certain delay before continuing to the next state. This models the delay between the CMD\_VALID phase and the WRITE\_VALID phase. If the $\text{AbortEvent}$ event is notified during the $S_{\text{READ\_FIFO}}$ state, the FSM moves to the $S_{\text{ABORT}}$ state. Otherwise, the $S_{\text{DATA\_SYNC}}$ state is reached.
The **S_ABORT** state will cancel the current transaction as well as all pending transactions. The current transaction and all the transactions that are in the **WriteFIFO** are send to the **TransactionCompletion** module.

In the **S_DATA_SYNC** state, the FSM will claim the next data element that should be send from the scmi array of the initiator. A pointer to this scmi array is retrieved through the **scmi_transaction_request** object that was send to the initiator transactor by the initiator.

The **S_FILL_WR_EXT** state will read the current data element from the scmi array of the initiator and place the data in the DTL Write extension (tlm_dtl_wr_ext) of the current transaction. The other attributes of the Write extension are also filled in here.

Next comes the **S_SENDTOBUFFER** state. In this state the transactor has the option to either tag the current write data element or to flush the contents of the write buffer in the target. Two configuration parameters of the transactor determine when data will be tagged or flushed. If a tag or flush operation is needed, the Write Channel FSM will place the current transaction in the BufferTransaction value and notify either the **TagEvent** or **FlushEvent** event.

In the **S_TRANSPORT** state, the WRITE_VALID phase of the current transaction is send via the TLM 2.0 forward path interface. After returning from the **nb_transport(...)** method call with a **TLM_COMPLETED**, **TLM_ACCEPTED** or **TLM_UPDATED** response, the **S_WAIT** state is reached.
The `S_ERROR` state should not be reached during normal operation of the transactor. Only a bad response from the `nb_transport(...)` call in the `S_TRANSPORT` state can cause the FSM to reach this state. The SystemC simulation will be stopped when the `S_ERROR` state is reached.

In the `S_WAIT` state, the FSM first waits until the `WriteAcceptEvent` event is triggered (which indicates that the WRITE_ACCEPT phase has been received). Once this event has been triggered the thread will wait for a fixed delay before continuing execution. This delay models the amount of time there is in between the transfer of two subsequent data elements in a burst. The default value for this delay is one clock cycle and should not be set lower than that.

If a write error has occurred (indicated by the `err_wr` signal set high in the `tlm2_dtl_wr_err_ext` extension), or if the `AbortEvent` event is triggered by the Command FSM, the state machine goes to the `S_ABORT` state. If there are more data elements to be sent as part of the current burst transaction, the FSM moves back to the `S_DATA_SYNC` state to claim another data element. When the last data element has been sent and there is no outstanding tagged write data element, the `S_COMPLETE_TRANSACTION` state is reached. If the transactor is still waiting for the acknowledgement of a tagged data item, the Write Channel FSM goes to `S_READ_FIFO` state to process the next transaction. In this case the Buffer Management Channel FSM will handle the completion of the transaction when the `TAG_ACK` phase is received.

The `S_COMPLETE_TRANSACTION` state completes the current write transaction by sending it to the TransactionCompletion module.

### 5.2.3 Read Channel FSM:

The Read Channel state machine (`TLM2_DTL_RD_FSM`) handles the incoming read data transfers of the DTL Read channel. The state machine is implemented as a GMLI_FSM_THREAD and consists of eight states. Figure 21 shows a diagram of this FSM module with the states, state transitions and external method calls made through the input and output ports.

The read channel FSM starts in the `S_READ_FIFO` state. In this state, the `ReadFIFO fifo` is read to obtain a pointer to the next read transaction `RTLTransactionStruct`. After the fifo has been read, the state machine will move to the `S_WAIT` state.

In the `S_WAIT` state, the FSM waits until the `ReadValidEvent` event is triggered (which indicates that the READ_VALID phase has been received). Once this event has been triggered the thread will wait for a fixed delay before continuing execution. This delay is set by the `ReadAcceptDelay` configuration parameter and indicates the number of cycles before the READ_ACCEPT phase will be send.

If a read error has occurred (indicated by the `err_rd` signal set high in the `tlm2_dtl_rd_err_ext` extension), or if the `AbortEvent` event is triggered by the Command FSM, the state machine goes to the `S_ABORT` state. Otherwise, the FSM moves to the `S_DATA_SYNC` state.
The \texttt{S\_ABORT} state will cancel the current transaction as well as all pending transactions. The current transaction and all the transactions that are in the \texttt{ReadFIFO} are send to the \texttt{TransactionCompletion} module.

In the \texttt{S\_DATA\_SYNC} state, the FSM will claim space for the next data element that is to be stored in the \texttt{scm} array of the initiator. A pointer to this \texttt{scm} array is retrieved through the \texttt{scm\_transaction\_request} object that was send to the transactor by the initiator.

The \texttt{S\_FILL\_RD\_EXT} state does nothing more than set the phase of the transaction to \texttt{READ\_ACCEPT}. Since this phase does not have its own extension, no extension attributes have to be filled in this state.

The \texttt{S\_TRANSPORT} state transmits the \texttt{READ\_ACCEPT} phase of the current transaction via the TLM 2.0 forward path interface. If TLM\_COMPLETED, TLM\_ACCEPTED or TLM\_UPDATED is returned from the \texttt{nb\_transport(\ldots)} method call, the FSM goes to the \texttt{S\_TRANSACTION\_COMPLETION} state if all data for the transaction has been received, or back to the \texttt{S\_WAIT} state if more data needs to be received.

The \texttt{S\_ERROR} state should not be reached during normal operation of the transactor. Only a bad response from the \texttt{nb\_transport(\ldots)} call in the \texttt{S\_TRANSPORT} state can cause the FSM to reach this state. The SystemC simulation will be stopped when the \texttt{S\_ERROR} state is reached.

The \texttt{S\_COMPLETE\_TRANSACTION} state completes the current read transaction by sending it to the \texttt{TransactionCompletion} module.
5.2.4 Buffer Management Channel FSM:

The Buffer Management Channel state machine \( TLM2\_DTL\_BUFFER\_FSM \) handles the tag, flush and abort features of the DTL protocol. The state machine is implemented as a GMFL\_FSM\_THREAD and has six states. Figure 22 shows a diagram of this FSM module with the states, state transitions and external method calls made through the input and output ports.

The Buffer Management Channel FSM starts in the \( S\_GET\_TRANSACTION \) state. The DTL initiator transactor can only handle buffer management for one transaction at a time. Therefore, the input for the Buffer Management FSM is a simple gmfl_value instead of a fifo (like with the other state machines). In the first state, the FSM waits for the TagEvent, FlushEvent or AbortEvent event. Once one of these events has been triggered, the transaction is read from the BufferTransaction value.

In the \( S\_FILL\_BUFF\_EXT \) state, the DTL Buffer Management extension (tlm2\_dtl\_buffer\_mgmnt\_ext) is filled in. Depending on which event was triggered, either the tag, flush or abort_all signals are set. Note that in case of a flush operation, the tag signal will be set as well. A tag acknowledgement from the target indicates that the buffers have been flushed.

The \( S\_TRANSPORT \) state sends the BUFFER_MGMNT phase of the current transaction via the TLM 2.0 forward path interface. After the \( nb\_transport(\ldots) \) call has been returned, the FSM goes to the \( S\_TRANSACTION\_COMPLETION \) state if the current transaction was an abort transaction, or to the \( S\_WAIT \) state otherwise.

The \( S\_ERROR \) state should not be reached during normal operation of the transactor. Only a bad response from the \( nb\_transport(\ldots) \) call in the \( S\_TRANSPORT \) state can cause the FSM to reach this state. The SystemC simulation will be stopped when the \( S\_ERROR \) state is reached.

![Figure 22: TLM 2.0 DTL Initiator Transactor Buffer Management Channel FSM Block](image-url)
In the _S_WAIT_ state, the FSM waits until the TagAckEvent event is triggered (which indicates that the TAG_ACK phase has been received). After this event has been triggered the thread will go to the _S_COMPLETE_TRANSACTION_ state to complete the transaction. Alternatively, if the current transaction is a write transaction that has not yet send all the data elements the FSM goes back to the _S_GET_TRANSACTION_ state to wait for the next transaction. In this case the Write Channel FSM will handle the completion of the transaction.

The _S_COMPLETE_TRANSACTION_ state completes the current read transaction by sending it to the TransactionCompletion module.

### 5.3 TLM 2.0 DTL Target Transactor

The TLM 2.0 DTL Target transactor converts incoming TLM 2.0 DTL transactions into PV transport calls that are send to the target. This transactor is used to connect a target IP component to a TLM 2.0 DTL interconnect. On the target side the transactor has a PV transport interface and on the side that is connected to the interconnect a TLM 2.0 DTL interface is used. A simplified diagram of this transactor has been shown in Figure 16. For a more detailed schematic of the transactor including all the module ports and interfaces, please refer to Figure 38 in Appendix B.

Similar to the DTL initiator transactor, the DTL target transactor also consists of three parts. The "reactive part" implements the _nb_transport(...)_ method that handles all incoming transactions from the TLM 2.0 forward path interface. The "active part" consists of four separate thread modules that drive the TLM 2.0 backward path of the DTL channels. Since the control of the DTL channels on the target side is relatively simple, the state machines for this transactor are implemented using simple thread classes from the GMFL library. In between the active and reactive part is a "synchronization" part that handles the synchronization between the forward and backward paths and the PV interface.

The _nb_transport(...)_ function in the reactive part places incoming transactions in the CommandFIFO, WriteFIFO, ReadFIFO or BufferFIFO fifos, depending on the transaction phase that is received and the type of transaction. When the READ_ACCEPT phase is received, the ReadAcceptEvent event is notified.

The PV interface block incorporates buffers for both the read and the write data that is send and received from the PV target. The threads from the active part of the transactor use the TLM 2.0 non-blocking transport interface to send the DTL payload extensions to the PV interface. Write data from the TLM 2.0 interface is stored in the Write Buffer until the last element is received or the buffer is full (buffer size can be set with a configuration parameter). All data in the write buffer is transported in a single PV transport call via the _PV_Initiator_port_ to the target. For read transactions, when the Read Buffer is empty, data is pre-fetched from the target into this buffer. The PV interface block notifies the TagAckEvent event whenever the Write Buffer contains a tagged data element and has been written to the target.

The following sections discuss the blocks of the DTL target transactor in more detail.
5.3.1 Command Channel Thread

The Command Channel Thread \((TLM2\textsubscript{DTL}\_CMD\_THREAD)\) handles the transactions that are received in the CMD\_VALID phase. Figure 23 shows the SystemC code definition of this module.

```c
GMFL\_FUNCTION\_THREAD(TLM2\_DTL\_CMD\_THREAD) {
public:
    \* Port Declarations \*
    sc_port<tlm::tlm_bw_nb_transport_if<tlm2_dtl::tlm2_dtl\_payload_types>> p_TLM2BackwardPath;
    sc_port<tlm::tlm_fw_nb_transport_if<tlm2_dtl::tlm2_dtl\_payload_types>> p_Transport;
    sc_fifo_in<TransactionPointersType> p_CommandFIFO;
    gmfl\_event\_wait\_port p_AbortEvent;
    sc_fifo_out<bool> pCanAcceptWrite;
    sc_fifo_out<bool> pCanSendRead;

    \* Constructor \*
    TLM2\_DTL\_CMD\_THREAD(sc_module_name name, tlm2_dtl::DTL\_ConfigStruct config) :
        gmfl\_function\_thread(name), Config(config), LastValid(false) {}

    \* main functionality \*
    void main() {  
        //Read transaction from Command FIFO:
        Transaction = p_CommandFIFO.read();
        p_AbortEvent->clearEvent();

        \* Use the signals from the previous transaction \*
        Transaction.Ext_dtl_cmd = LastDTLCmdGroup;

        \* Keep a copy of the transaction \*
        LastDTLCmdGroup = Transaction.Ext_dtl_cmd;
        LastValid = true;

        \* Send the command extension to the Target\_PV\_Driver module \*
        TransactionType::PhaseType p(tlm2_dtl::CMD\_VALID);
        p_Transport->nb_transport_fw(*Transaction.Payload,p,*Transaction.Time);

        \* Wait for command accept \*
        wait(*Transaction.Time + Config.CmdAcceptDelay, p_AbortEvent->wait\_for\_event());

        \* If transaction aborted then return \*
        if(p_AbortEvent->isEvent()) {  
            REPORT("Abort Event Triggered, Flushing Command FIFO...");
            \* Clear Event \*
            p_AbortEvent->clearEvent();
        }

        \* Send CMD\_ACCEPT command \*
        Send CMD\_ACCEPT phase:
        p_TLM2BackwardPath->nb_transport_bw(  
            *Transaction.Payload,*Transaction.Phase,*Transaction.Time);

        \* Clear transaction \*
        p_CommandFIFO.write();
        p_AbortEvent->clearEvent();

        \* Send CMD\_ACCEPT command \*
        Send CMD\_ACCEPT phase:
        p_TLM2BackwardPath->nb_transport_bw(  
            *Transaction.Payload,*Transaction.Phase,*Transaction.Time);

    }

    \* Processor \*
    tlm2_dtl::DTL\_ConfigStruct Config;
    TransactionPointersType Transaction;
    TransactionPointersType::DTL\_CmdExtType LastDTLCmdGroup;
    LastValid;
}
```

Figure 23: TLM 2.0 DTL Target Transactor Command Channel Thread
After the transaction object is read from the CommandFIFO the cmd_trans signal is evaluated. If this signal is set to false, the attributes from the previous transaction will be used. Otherwise the attributes of the current transaction are used. The transaction object is send to the Target_PV_Driver module through a nb_transport(...) call. To model the accept delay a certain amount of time is consumed before the CMD_ACCEPT phase is send via the TLM 2.0 backward path to the initiator. Finally, synchronization with the Write Channel Thread or Read Channel Thread is done by writing a value of 1 to the CanAcceptWrite or CanSendRead fifo.

### 5.3.2 Write Channel Thread

The Write Channel Thread (TLM2_DTL_WR_THREAD) handles the transactions that are received in the WRITE_VALID phase. Figure 24 shows the SystemC code of this module.

```systemc
GMFL_FUNCTION_THREAD(TLM2_DTL_WR_THREAD) {
    // Port Declarations
    sc_port<tlm::tlm_bw_nb_transport_if<tlm2_dtl::tlm2_dt1::payload_types>> p_TLM2BackwardPath;
    sc_port<tlm::tlm_fw_nb_transport_if<tlm2_dtl::tlm2_dt1::payload_types>> p_Transport;
    sc_fifo_in<TransactionPointersType> p_WriteFIFO;
    sc_fifo_in<bool> p_CanAcceptWrite;
    gmfl_event_wait_port p_AbortEvent;

    // Constructor
    TLM2_DTL_WR_THREAD(sc_module_name name, tlm2_dtl::DTL_ConfigStruct config) :
        gmfl_function_thread(name), Config(config), FirstBeat(true) {}{

    // Main functionality
    void main() {
        Transaction = p_WriteFIFO.read(); // Get transaction from Write FIFO

        if (FirstBeat) {
            p_CanAcceptWrite.read();
            FirstBeat = false;
        }

        sc_time WriteLatency = SC_ZERO_TIME;
        TransactionPointersType::PhaseType = tlm2_dt1::WRITE_ACCEPT;


        WriteLatency = std::max(WriteLatency, Config.MinWriteAcceptDelay);

        // Consumer write latency delay can be interrupted by abort event
        wait(*Transaction.Time + WriteLatency, p_AbortEvent->wait_for_event());
        if (p_AbortEvent->isEvent()) {
            REPORT("Abort Event Triggered, Flushing Write FIFO...");
            if (p_WriteFIFO.num_available() > 0) { // Remove transaction from queue
                p_WriteFIFO.read();
            }
        }

        *Transaction.Time = SC_ZERO_TIME;
        Transaction.Ext_dt1_wr_err->tlm2_dt1::wr_accept = true;
        if (Transaction.Ext_dt1_wr->tlm2_dt1::wr_last) {
            FirstBeat = true;
        }

        *Transaction.Phase = tlm2_dt1::WRITE_ACCEPT;

        // Send WRITE_ACCEPT phase via backward path:
    }
}
```
First, the transaction object is read from the WriteFIFO. If the current transaction represents the first element of a burst, synchronization with the Command Channel Thread is needed to make sure that the CMD_VALID phase is send before the WRITE_ACCEPT phase. This is done by performing a blocking read of the CanAcceptWrite fifo. Then the transaction is send to the Target_PV_Driver module through a nb_transport(...) call. If the time delay that is annotated by the PV driver module is less than a fixed minimum delay (set by the MinWriteAcceptDelay configuration parameter), the minimum delay is consumed. Finally the thread sends the WRITE_ACCEPT phase via the TLM 2.0 backward path to the initiator.

### 5.3.3 Read Channel Thread

The Read Channel Thread (TLM2_DTL_RD_THREAD) handles read transactions by driving the READ_VALID phases on the TLM 2.0 backward path. Figure 25 shows the code definition of this module.

```cpp
// Code definition of TLM2_DTL_RD_THREAD module.

GMFL_FUNCTION_THREAD(TLM2_DTL_RD_THREAD) {
    // Port Declarations
    sc_port<tlm::tlm_bw_nb_transport_if<tlm2_dtl::tlm2_dtl_payload_types>> p_TLM2BackwardPath;
    sc_port<tlm::tlm_fw_nb_transport_if<tlm2_dtl::tlm2_dtl_payload_types>> p_Transport;
    sc_fifo_in<TransactionPointersType> p_ReadFIFO;
    sc_fifo_in<bool> p_CanSendRead;
    gmfl_event_wait_port p_ReadAcceptEvent;
    gmfl_event_wait_port p_AbortEvent;

    TLM2_DTL_RD_THREAD(sc_module_name name, tlm2_dtl::DTL_ConfigStruct config) :
        gmfl_function_thread(name), Config(config), BurstCount(0), BeatCounter(0) {}

    void main() {  // Main function
        Transaction = p_ReadFIFO.read();  // Get transaction from read FIFO
        BurstCount = (Transaction.Ext_dtl_cmd->tlm2_dtl_cmd_block_size+1) ... 
                      (Transaction.Ext_dtl_cmd->tlm2_dtl_cmd_lines+1);
        bool firstbeat = true;

        for (int i = 0; i < BurstCount; i++) {
            sc_time ReadLatency = SC_ZERO_TIME;
            TransactionPointersType::PhaseType p(tlm2_dtl::READ_VALID);

            if (firstbeat) {
                // Start sending data elements
                firstbeat = false;\n                p_Transport->nb_transport_fw(*Transaction.Payload,p,ReadLatency);
            } else {
                Send transaction to PV driver block:
                p_Transport->nb_transport_fw(*Transaction.Payload,p,ReadLatency);
            }

            // Round down time to the nearest clock cycle boundary
            ReadLatency = std::max(ReadLatency,Config.MinNextReadDelay);

            // Consume time:
            wait(*Transaction.Time + ReadLatency,p_AbortEvent->wait_for_event());
            if (p_AbortEvent->wasEvent()) {
                REPORT("Abort Event Triggered, Flushing Read FIFO...");
                while(p_ReadFIFO.num_available() > 0) {
                    p_ReadFIFO.read();
                }
                return;
            }
        }
    }
}
```
After the transaction object is read from the ReadFIFO, the burst size is calculated. The thread then enters a loop that will handle the individual beats of the burst. For every beat, the transaction is sent to the Target_PV_Driver module through a `nb_transport(...)` call, where a data element will be placed in the DTL extension attributes. If the time delay that is annotated by the PV driver module is less than a fixed minimum delay (set by the MinNextReadDelay configuration parameter), the minimum delay is consumed. If the current data element is the last of the burst, or if the maximum burst size has been reached, the `rd_last` signal is set to true.

For the first element of a burst, synchronization with the Command Channel Thread is needed to make sure that the CMD_VALID phase is sent before the READ_VALID phase. This is done by performing a blocking read of the CanSendRead fifo. After this, the thread sends the READ_VALID phase via the TLM 2.0 backward path to the initiator. The loop then waits for the ReadAcceptEvent event to be triggered, after which the loop will start over.
5.3.4 Buffer Management Channel Thread

The Buffer Management Channel Thread (TLM2_DTL_BUFFER_THREAD) handles tag, flush and abort operations that are received in the BUFFER_MGMNT phase. Figure 25 shows the SystemC code for this module.

```systemc
GMF1_FUNCTION_THREAD(TLM2_DTL_BUFFER_THREAD) {
    // Port Declarations
    sc_port<tlm::tlm_bw_nb_transport_if<tlm2_dt1::tlm2_dt1_payload_types>> p_TLM2BackwardPath;
    sc_port<tlm::tlm_fw_nb_transport_if<tlm2_dt1::tlm2_dt1_payload_types>> p_Transport;
    sc_fifo_in<TransactionPointersType> p_BufferFIFO;
    gmfl_event_wait_port p_TagAckEvent;
    gmfl_event_notify_port p_AbortEvent;

    // Constructor
    TLM2_DTL_BUFFER_THREAD(sc_modu1e_name name, t1m2_dt1::DTL_ConfigStruct config) :
        gmfl_function_thread(name), Config(config) {
    }

    main() {
        Transaction = p_BufferFIFO.read();
        TransactionPointersType::PhaseType p(t1m2_dt1::BUFFER_MGMNT);
        p_Transport->nb_transport_fw(*Transaction.Payload,*Transaction.Phase,*Transaction.Time);

        if (Transaction.Ext_dtl_buffer_mgmt->tlm2_dt1_abort_all) {
            PRINT_DEBUG(1, "Aborting all pending commands and data phases!!!");
            p_AbortEvent.notify();
            if (p_BufferFIFO.num_available() > 0) {
            p_BufferFIFO.read();
            }
            yield();
            p_AbortEvent.cancel();
            Tagged = false;
        } else {
            if (Transaction.Ext_dtl_buffer_mgmt->tlm2_dt1_tag) {
                Tagged transaction, wait for acknowledgment...
                p_TLM2BackwardPath->nb_transport_bw(*Transaction.Payload,*Transaction.Phase,*Transaction.Time);
                Transaction.Ext_dtl_tag_ack->tlm2_dt1_tag_ack = true;
            }
            Transaction.Phase = t1m2_dt1::TAG_ACK;
            Send TAG_ACK phase via Backward Path
            p_TLM2BackwardPath->nb_transport_bw(*Transaction.Payload,*Transaction.Phase,*Transaction.Time);
        }
    }
}
```

Figure 26: TLM 2.0 DTL Target Transactor Buffer Management Channel Thread

After the transaction object is read from the BufferFIFO, the transaction is send to the Target_PV_Driver module through a `nb_transport(...)` call, where a tag, flush or abort will be performed. In case of an abort transaction, the AbortEvent event is notified to signal the other threads in the transactor to abort all transactions. If a tag operation was done, the thread will wait until the TagAckEvent event has been notified by the Target_PV_Driver module. Finally, the thread sends the TAG_ACK phase via the TLM 2.0 backward path to the initiator.
5.4 Conclusion

The TLM 2.0 extensions for the DTL protocol have been discussed in this chapter. The DTL generic payload extensions and transaction phases together form the cycle-accurate TLM 2.0 DTL initiator and target interfaces that are needed to model cycle-accurate DTL communication. The initiator and target transactors that implement these interfaces have been described in great detail in this chapter as well.

From the description of the transactors in this chapter and the figures in Appendix B it can be seen that these transactors are quite complex and designing them took quite a lot of time. Designing the FSM blocks is the most time-consuming task, since a lot of the details of the DTL protocol had to be taken into account. Fortunately, the DTL transactors can now be used as a template for creating transactors for other protocols. The basic structure of the transactors (as shown in Figure 14) will not change. Due to the modular approach of the GMFL strategy it is relatively easy to swap the FSM blocks of the DTL protocol with FSM blocks for another protocol.

The creation of the transactors for the MTL protocol took considerable less time, because a lot could be re-used from the DTL transactors. This is a great benefit of using the GMFL library for the design of the transactors.
6 Cycle Accurate TLM 2.0 DTL Interconnect

For the creation of cycle accurate TLM 2.0 interconnect models another set of transactors is needed. These transactors, or bus adapters, translate the TLM 2.0 protocol interface that is connected to the other components of the system into a fifo interface, which is used internally in the interconnect model. Figure 2 shows an example diagram of how these bus adapters would typically be used in a system. The fifo interface that is used inside the interconnect model consists of a number of fifos in which the individual extensions of the protocol are placed. This fifo interface is connected to a module which performs the actual routing of the TLM 2.0 protocol extensions.

The method for creating the TLM 2.0 bus adapters for any protocol is basically the same as the methodology for creating the TLM 2.0 initiator and target transactors (discussed in Chapter 4), except there is no need to create state machines for the bus adapters. From the TLM 2.0 interfaces that are defined in the TLM 2.0 mapping step of the methodology, the bus adapters can be implemented using a template for the initiator and target bus adapters.

In order to test if the TLM 2.0 DTL protocol can be used to create cycle-accurate interconnect models, a simple TLM 2.0 DTL interconnect has been created. This interconnect consists of a simple router that routes each DTL extension individually. TLM 2.0 DTL initiator and target bus adapter have been created using the same TLM 2.0 interfaces that were used for the TLM 2.0 DTL initiator and target transactors (See Chapter 5.1). These adapters extract (on initiator side) and attach (on target side) the seven DTL extensions from the generic payload, so that they can be routed by the DTL extension router. Figure 27 shows how the bus adapters and the router are connected to form the DTL interconnect. In a more advanced bus model, multiple target and initiator adapters would be used to connect multiple initiators and targets.

The bus adaptors and the simple router are discussed in the following sections.
6.1 TLM 2.0 DTL Initiator Bus Adapter

The TLM 2.0 DTL Initiator Bus Adapter (tlm2_dtl_initiator_buffer_adaptor) will extract the DTL payload extensions from the generic payload and place them in synchronization buffers (fifos), so they can be read by the router and transported to their destination. The phase of the TLM 2.0 transaction determines which extension is extracted from the payload. Figure 39 in Appendix C shows all the modules that are inside this adaptor and how they are connected together.

The InitiatorBufferAdaptor_Transport_Interface module implements the nb_transport(...) function that handles the TLM 2.0 transactions that come from the initiator via the forward path. When the CMD_VALID phase is received, the transaction is placed in the CommandTransactionFIFO fifo. In case of a read transaction, the transaction is also placed in the ReadTransactionFIFO fifo. If a WRITE_VALID phase is received, the transaction is placed in the WriteTransactionFIFO fifo. For the BUFFER_MGMNT phase, the transaction is placed in the BufferTransactionFIFO fifo. The READ_ACCEPT phase will trigger the ReadAcceptEvent event. All the transaction fifo's, with the exception of the ReadTransactionFIFO fifo, are of size 1. The nb_transport(...) method will perform non-blocking writes to these fifo's. A value of TLM_COMPLETED is returned when trying to write to a fifo that is full.

There are four GMFL_FUNCTION_METHOD modules that all basically operate in the same way. These methods are made sensitive to the events of the fifo's that they are connected to and to the AbortEvent event. This means that whenever data is written to an input fifo or read from an output fifo, the method gets triggered. The Command, Write and Buffer methods will first read from their respective transaction fifo's. Then, the corresponding DTL extension (tlm2_dtl_cmd_ext, tlm2_dtl_wr_ext or tlm2_dtl_buffer_mgmt_ext) is extracted from the payload and placed in the outgoing extension fifo. When the response extension (tlm2_dtl_cmd_accept_ext, tlm2_dtl_wr_err_ext or tlm2_dtl_tag_ack_ext) is placed in the incoming extension fifo by the router, the method will read the fifo and attach it to the transaction. Finally the method will send the corresponding response phase (CMD_ACCEPT, WRITE_ACCEPT or TAG_ACK) via the TLM 2.0 backward path to the initiator.

The Read method will also first get a transaction from the ReadTransactionFIFO fifo. Then the method will wait for an incoming read extension (tlm2_dtl_rd_ext) and send the READ_VALID phase to the initiator via the TLM 2.0 backward path. The method will then wait for the ReadAcceptEvent event to be triggered, after which it will either wait for the next incoming read extension (if more data for this transaction is expected) or read the next transaction from the ReadTransactionFIFO fifo.

If an abort transaction is received by the Buffer Method, the AbortEvent will be notified. This will cause all methods to empty the transaction fifos and incoming extension fifos.

6.2 TLM 2.0 DTL Target Bus Adapter

The TLM 2.0 DTL Target Bus Adapter (tlm2_dtl_target_buffer_adaptor) receives the DTL payload extensions coming from the router and attach them to a payload, which is
then send to the target via the TLM 2.0 DTL interface with the correct phase set. Figure 40 in Appendix C shows all the modules that are inside this adaptor and how they are connected together.

The TLM 2.0 DTL Target Bus adaptor operates in basically the same way as the TLM 2.0 DTL Initiator Bus adaptor, except in reverse. Incoming DTL extensions (tlm2_dtl_cmd_ext, tlm2_dtl_wr_ext and tlm2_dtl_buffer_mgmnt_ext) are read by their respective method modules (CommandMethod, WriteMethod and BufferMethod). These methods will attach the extension to a payload object and send the payload with the corresponding phase (CMD_VALID, WRITE_VALID or BUFFER_MGMNT) via the TLM 2.0 forward path interface to the target. When the target returns with the response phase (CMD_ACCEPT, WRITE_ACCEPT or TAG_ACK), the TargetBufferAdapter_Transport_Interface module places the transaction in the corresponding fifo (CommandTransactionFIFO, WriteTransactionFIFO or BufferTransactionFIFO). The methods then extract the response extension (tlm2_dtl_cmd_accepct_ext, tlm2_dtl_wr_err_ext or tlm2_dtl_tag_ack_ext) from the transaction and place them in the outgoing extension fifos, where they will be read by the router.

The ReadMethod method will first read the transaction that was placed in the ReadTransactionFIFO fifo by the TargetBufferAdapter_Transport_Interface module during an incoming READ_VALID phase. The tlm2_dtl_rd_ext extension is extracted from the transaction and placed in the outgoing read extension fifo. Immediately after the extension was successfully placed in the fifo, the method sends the READ_ACCEPT phase back to the target via the TLM 2.0 forward path interface.

If an abort transaction is received by the BufferMethod, the AbortEvent will be notified. This will cause all methods to empty the transaction fifos and incoming extension fifos.

6.3 Simple DTL Extension Router

The router that is used in the transactor testbench is a very simple router that just routes the individual DTL extensions from an initiator to the target destination. The router (test_dl_router) is a SystemC module that contains four SC_METHOD processes (one for each DTL channel). The router is constructed with two template arguments that indicate the number of initiators and targets that are connected to the router. Each initiator and target will be connected to a set of seven ports that will transfer the DTL extensions to and from the bus adapters. Figure 28 shows the diagram of the DTL extension router with a single initiator and a single target connection.

Each target must register its base address and size with the router before any routing can be done.

The routing of the extensions is done by keeping track of a number of queues. Each initiator will have its own set of four queues: Command, Write, Read and Buffer route queues. When the router receives a tlm2_dtl_cmd_ext extension from an initiator, the address is examined and the target destination is determined from the memory map. This destination is then pushed on the Command and Write or Read route queues for that
6.4 Conclusion

The TLM 2.0 DTL bus adapters that were discussed in this chapter can be used to create cycle-accurate DTL bus models. The internal interface that is used inside these bus models consists of a set of fifos that store the individual DTL payload extensions. The router model that is used to route the DTL extensions determines the actual functionality of the bus. A simple router model has been shown. This router merely passes the DTL extensions from the initiator bus adapter to the target bus adapter and vice-versa. A more complex bus model could perform additional features such as arbitration, transaction queuing and re-ordering, etc (as long as the features are supported by the protocol).

The creation of the bus adapters is relatively easy, compared to the creation of the initiator and target transactors of Chapter 5. No complex state machines are needed in the bus adapters and only a few fifos and methods that all implement similar functionality are required. Furthermore, the DTL bus adapters shown in Figure 39 and Figure 40 can be used as templates to create bus adapters for any other TLM 2.0 bus protocol that has been created with the methodology of Chapter 4, because the basic functionality of bus adapters will always be the same.
7 Test Results

In order to test if the TLM 2.0 cycle-accurate DTL interface and transactors that were discussed in Chapter 5 are functioning correctly, a number of test scenarios have been executed. The goal of these test scenarios was to verify that the communication is indeed cycle-accurate and adheres to the DTL protocol specification. Another objective of these tests was to obtain measurements on the simulation speed of these cycle-accurate TLM 2.0 models.

7.1 TLM 2.0 DTL Transactors Test

Figure 29 shows the test set-up that was used to test the TLM 2.0 DTL initiator and target transactors. A simple initiator (traffic generator) and target (memory) are connected directly together using the DTL transactors. The initiator performs a number of read or write transactions (via the SCML post interface) of varying size. The initiator transactor translates the SCML post transactions into cycle-accurate TLM 2.0 DTL transactions. The traffic monitor extracts the DTL signals from the TLM 2.0 payload extensions and saves them to a waveform file that can be evaluated after the simulation.

The target transactor receives the TLM 2.0 DTL transactions and converts them to PV transport transactions. The target receives the PV transport transactions and performs the corresponding read or write operation in the memory.

Figure 30 shows an example waveform that was observed by the traffic monitor when the initiator performs a single (tagged) write transaction, followed by a single read transaction. The clock signal is only generated by the traffic monitor as a reference signal and is not part of the TLM 2.0 DTL interface. In order to clearly show when each DTL phase is being send, processing delays have been modeled in the transactors. The command phase is accepted by the target transactor with a 1 cycle delay, the read and write data phases are accepted with a 2 cycle delay and the delay between the transactions is 1 cycle. The phase flows for the transactions are also indicated in the figure. For the tagged write transaction, the following phase flow is observed:

\[\text{Tagged Write Transaction Phase Flow:}\]

- Command: 1 cycle delay
- Read Data: 2 cycle delay
- Write Data: 2 cycle delay
- Command Acceptance: 1 cycle delay
- Read Data Acceptance: 1 cycle delay
- Write Data Acceptance: 1 cycle delay
- Delay Between Transactions: 1 cycle
CMD_VALID \rightarrow WRITE_VALID \rightarrow BUFFER_MGMNT \rightarrow

CMD_ACCEPT \rightarrow WRITE ACCEPT \rightarrow TAG_ACK

Note: Phases that are printed like this are send by the initiator transactor and phases that are printed like this are send by the target transactor.

The read transaction goes through the DTL transaction phases as follows:

CMD_VALID \rightarrow CMD_ACCEPT \rightarrow READ_VALID \rightarrow READ ACCEPT

Observations of the waveforms that are produced by the traffic monitors indicate that the TLM 2.0 DTL interface is indeed accurate at the clock-cycle level, which was the objective of the methodology.

![Waveform of simulation with a write and a read transaction](image)

Figure 30: Waveform of simulation with a write and a read transaction. Delays are modeled to simulate processing delays.

The simulation speed of the testbench system is measured by calculating how many clock-cycles are simulated per second. The cycles per second (CPS) of a simulation is calculated as follows:

\[
CPS = \frac{t_{\text{sim}}}{t_{\text{CPU}} \cdot d_{\text{cycle}}}
\]

Where \(t_{\text{sim}}\) is the elapsed SystemC simulation time, \(t_{\text{CPU}}\) is the physical time it took to simulate the testbench and \(d_{\text{cycle}}\) is the reference clock period.

For the speed measurement test, the traffic generator issues 2 million alternating read/write transactions. The rest is done for burst sizes of 1, 4, 8, 16, 32 and 64 words (a word equals 4-bytes). The testbench is simulated with two different configurations.
Configuration 1 models no delays and every clock-cycle a data item is transferred. Speed results for this configuration are shown in Figure 31. Configuration 2 simulates processing delays by introducing a 2 cycle delay for every command and data phase. Results are shown in Figure 32. From these speed figures, it can be clearly seen that the number of cycles per second increases with bigger burst sizes. This is because although both \( t_{\text{cm}} \) and \( t_{\text{cpu}} \) increase with bigger burst sizes, they do not increase at the same rate.

Furthermore, it can be clearly seen that while the introduction of computational (processing) delays in configuration 2 increases the simulation time, the CPU time stays about the same. This means that the effective number of simulated cycles per second increases when delays are added to the communication. The reason for this is that the TLM 2.0 DTL interface and transactors use timing annotation to consume the delays and do not use clock sensitive processes. Therefore the signals are not evaluated every clock-cycle (as is usually the case in traditional clock-driven signal-level models), but only when there is actually a transaction taking place. The number of function calls needed for a transaction remains the same, whether the transaction takes one cycle or many cycles. For burst sizes of 64 words, a simulation speed of 238 kcycles/second can be achieved when no processing delays are modeled. Adding processing delays, which is useful for modeling memory access latencies and communication delays gives an effective simulation speed-up that is proportional to the amount of delay that is added. In configuration 2, where the DTL command and data phases take three times as long as in
configuration 1, the simulation speed for 64 word bursts is also about three times as high (708 kcycles/second).

### 7.2 TLM 2.0 DTL Interconnect Test

In order to test the TLM 2.0 DTL interconnect model that was discussed in Chapter 6, the testbench set-up of Figure 29 is modified so that it includes the DTL interconnect. Figure 33 shows the test scenario that is used to test the TLM 2.0 DTL bus adapters and the extension router. The initiator, target and the TLM 2.0 DTL transactors remain the same as in the previous testbench. Two traffic monitors are used so that both the signals on the initiator side and the target side of the interconnect can be monitored.

The exact same tests that were performed with the previous set-up are also performed for this testbench.

The speed figures for the testbench of Figure 33 for configuration 1 and configuration 2 are shown in Figure 34 and Figure 35, respectively.

For the test scenario with the interconnect, simulation speeds of 122 kcycles/second for configuration 1 and 358 kcycles/second for configuration 2 can be obtained for 64 word burst transactions. These figures show that the DTL interconnect (the router and the two bus adapters) has a considerable impact on the simulation speed. A speed reduction of about 50% is observed when the DTL interconnect is added to the system. This is not surprising, considering the amount of extra methods and fifos that are introduced in the
Figure 33: Testbench to test the DTL interconnect, which is used to connect the initiator and target sides. Traffic monitors are used to trace the DTL signals.

interconnect. The important part, however is that the interconnect does not introduce any clock-sensitive processes, which means that the entire system remains clock free.

The absence of clock sensitive processes, also from the interconnect, means that the TLM 2.0 cycle-accurate modeling style will scale very well. When a more complex system is modeled that also simulates computation in the initiator and target modules and a more complex bus model that also models bus latencies is used, the impact of the DTL interconnect on the overall system simulation speed will be further reduced.
Figure 34: Speed measurements of the TLM 2.0 DTL interconnect test setup (Figure 33), using the configuration with no processing delays (worst case scenario).

Figure 35: Speed measurements of the TLM 2.0 DTL interconnect test setup (Figure 33), using the configuration which has a 2 cycle processing delay for each command and data phase.
8 Conclusion

The semiconductor and EDA industries are adopting new TLM techniques in order to handle the increasing complexity of modern Systems-on-Chip. In an effort to reduce the design time (time-to-market) of SoCs, virtual prototypes of systems are used to perform a wide range of design tasks. TLM models have proven themselves to be valuable for use-cases such as software development that require high simulation speeds but do not need to be very accurate. However, one of the problems that current TLM techniques have is that models that have a high degree of accuracy (clock cycle-accurate) can only run at low simulation speeds.

The TLM 2.0 standard has been introduced by OSCI to provide an inter-operability standard for the creation of TLM models. This standard enables the re-use and exchange of IP models. In addition to this re-usability, the TLM 2.0 standard also focuses on improving the simulation speed of models by introducing mechanisms such as DMI, temporal decoupling and the non-blocking transport interface. The TLM 2.0 standard introduces coding styles for the creation of loosely-timed and approximately-timed modes. Furthermore, an extension mechanism is provided that enables users to increase the accuracy of their models.

A methodology to create protocol specific, bus-cycle accurate interfaces and transactors using the TLM 2.0 standard has been developed as part of this thesis research. The methodology has been demonstrated for a use-case example using the OTL protocol. The results of the DTL example show that the methodology can be used to create cycle accurate interfaces by using the extension mechanisms of the TLM 2.0 standard.

While the initial design for the DTL transactors was quite complex, design of transactors for other protocols in the future will be easier since many parts of the DTL transactors can be re-used due to its modular design. The most difficult part of creating the transactors is the design of the finite state machines. Future research may lead to automating (parts of) the transactor creation process.

In addition to the TLM 2.0 DTL initiator and target transactors a set of TLM 2.0 DTL bus adapters have also been created. These bus adapters make it possible to create cycle-accurate DTL interconnect components. An example of such an interconnect model has also been shown in this thesis work.

The test scenarios that have been executed show that a simple TLM 2.0 DTL testbench model the example interconnect can run at a speed of around 122 kcycles/second for burst sizes of 64 words and with no processing delays (worst case). This is at least an order of magnitude better than a traditional signal-level RTL model. The real benefit of this modeling proposal is, however, that the effective simulation speed improves significantly (up to 359 kcycles/second) when a more realistic scenario with processing delays is simulated, retaining the same level of accuracy. This is because the number of transactions and function calls remain the same. This would not be the case in traditional clock-driven RTL models, where synchronization occurs every clock-cycle.
The work presented in this thesis can be seen as a proof of concept, showing that it is possible to use the new TLM 2.0 standard to create cycle-accurate models.
9 References

[2] SystemC modeling language (www.systemc.org)
[14] TLM 2.0 User Manual, included in the TLM 2.0 library kit
[17] CoWare. www.coware.com
[23] Generic Modeling Features Library (GMFL), NXP internal library
### DTL Signal Mapping to TLM 2.0 DTL Extensions

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Driver</th>
<th>Command Channel</th>
<th>Write Channel</th>
<th>Read Channel</th>
<th>Buffer Management Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd_valid</td>
<td>Initiator</td>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_accept</td>
<td>Target</td>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_trans</td>
<td>Initiator</td>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_addr1[0]</td>
<td>Initiator</td>
<td>uint64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_read</td>
<td>Initiator</td>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_block_size[0]</td>
<td>Initiator</td>
<td>uint32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_data_size[0]</td>
<td>Initiator</td>
<td>uint32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_wrap</td>
<td>Initiator</td>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_reverse</td>
<td>Initiator</td>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_start[0]</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd_secure</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr_valid</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr_accept</td>
<td>Target</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr_desig[0]</td>
<td>Initiator</td>
<td>uint64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wr_last</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd_valid</td>
<td>Target</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd_accept</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd_size[0]</td>
<td>Target</td>
<td>uint64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd_test</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>flush</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag_ack</td>
<td>Target</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>err_rd</td>
<td>Target</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>err_wr</td>
<td>Target</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort_all</td>
<td>Initiator</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **a** = address width
- **b** = rd mask width
- **c** = data size width
- **e** = block size width
- **n** = data width
- **r** = number of lines width
- **s** = the address offset width

Figure 36: DTL Signal to TLM 2.0 DTL extension mappings.
Figure 37: SCML post to TLM 2.0 DTL Transactor (Initiator DTL Transactor).
Figure 38: TLM 2.0 DTL post to PV Transactor (Target DTL Transactor).
Figure 39: TLM 2.0 DTL to DTL Extensions Transactor (Initiator DTL Bus Adaptor).
Figure 40: DTL Extensions to TLM 2.0 DTL Transactor (Target DTL Bus Adaptor).