Tesis Doctoral

Métodos y herramientas para el diseño a nivel de sistema de plataformas multiprocesador heterogéneas para multimedia

Victor Reyes Suarez
Las Palmas de Gran Canaria, 2008
Título de la Tesis

Métodos y herramientas para el diseño a nivel de sistema de plataformas multiprocesador heterogéneas para multimedia

Tesis Doctoral presentada por D. Víctor Reyes Suárez

Dirigida por el Dr. D. Tomás Bautista Delgado

El Director, El Doctorando,

Las Palmas de Gran Canaria, a 23 de octubre de 2008.
As the integration capability of silicon technologies continues growing, embedded systems on a chip are becoming more complex, adding more components and functionality into them. Moreover nowadays Consumer Electronic industry competitiveness results in shorter development times for new products trying to enter the market. This is imposing an enormous pressure on development teams, who have to cope with more complex systems in less time. The ultimate goal of any design methodology or tool is to enable designers to create more complex systems-on-chip, in less time, and with more quality. Conventional design and verification methods applied broadly in industry cannot cope any longer with the complexity of the designs and their ever-shorter time-to-market.

Electronic System Level design methods and tools are being proposed as a complement to conventional solutions, to improve the productivity of the designers and to bridge the design and verification gaps. ESL is rapidly settling down thanks to standards such as SystemC and TLM and their acceptance in Semiconductor industry. More concretely ESL is being applied on current design flows mainly via SystemC TLM Virtual Prototypes. A VP can be applied effectively to several activities (use-cases) during the design cycle. However different use-cases have different requirements in terms of accuracy, simulation speed, etc, which makes
very difficult to come up with a single modeling style that fits all these requirements. On the other hand, having to create and maintain separated models for each use-case will drastically reducing the benefits of ESL technology due to the elevated cost of creating and maintaining the models consistent with each other. Hence, model reuse and refinement is a must for the success of ESL technology.

The main contributions of this PhD thesis are innovative modeling methods, as well as supporting tools and libraries, which enable model reuse and refinement along different ESL abstraction levels relying completely on the IEEE 1666 SystemC and the new OSCI TLM 2.0 standards. More concretely the contributions can be separated on two refinement steps: “Functional to Architecture model refinement” and “Architecture model gradual refinement”. For the “Functional to Architecture model refinement” step modeling methods that enable the separation between the application functionality and the HW architecture, as well as their posterior mapping, are disclosed in this work. These concepts are implemented in the CASSE framework, which simplifies the iterative process of setting up the HW platform model, map the application model, configure the overall system, simulate and analysis the obtained results. For the “Architecture model gradual refinement” step the separation of the HW IP TLM model in two distinct parts, behavior core and interface adaptors, is the key concept that supports the methodology applied in this PhD thesis. Furthermore, advanced modeling methods and concepts, such as hierarchical modeling, dynamic layout, structural composition and advanced clock objects are presented as the way to create speed optimal refine-able architectural models with limited effort. Finally, innovative modeling methods to create bus protocol specific Cycle Accurate interfaces adaptors using the new TLM 2.0 standard are proposed in this work. The work presented on this PhD thesis demonstrates that: 1) the extension mechanisms of the TLM 2.0 standard can be used to create a protocol specific Bus Cycle-Accurate interface and 2) the simulation speed of the resulting models is two orders of magnitude better than signal-level RTL models, while retaining the same level of accuracy.
The research presented in this PhD thesis has been conducted during two different stages of my career. First part was carried out while I was a research assistant in the Institute for Applied Microelectronics (IUMA) working for the EU funded CAMELLIA project and the ARTEMI project (2002-2005). I would like to thank all the people whom shared with me those years of learning and hard work, but also years of fun (this goes for you Rias) and travels around Europe. Second part of this thesis was followed up after I joined Philips Research (now NXP Semiconductors) in The Netherlands (2005-2008). It is difficult to capture here how much I have learned and experienced during these years. I would like to thank all my colleagues from the System Design Methods and Flows groups for all the nice discussions (both in the coffee breaks and in front of the whiteboard) during this period, especial thanks to Frans Theeuwen and Marleen Boonen for believing on me and giving me the opportunities to grow professionally on my career.

I want to thank my PhD director Tomas Bautista for his support and help during this time and to explicitly mention Luz Garcia, Teresa Medina and Bert van Mol for their direct contributions to this work. I would like to show my appreciation as well to the people from the OSCI TLM Working Group for all the (lengthy but nice) debates during the hectic times before TLM 2.0 was born.
I am particularly grateful to two persons. The Professor Antonio Nuñez who always help me selflessly and motivate me to finish this PhD thesis, and my good friend Wido Kruijtzer with who I have enjoyed talking about System Level Design, brainstorming in front of a whiteboard and drinking beer in equally big proportions during all these years.

Por ultimo quiero agradecer a mis padres, Aniceto y Maria, su apoyo todos estos años, gracias por siempre estar ahí, gracias papa por nunca cansarte de recordarme que tenía que acabar la tesis. The last thanks are for you Anneke, ik hou van je.
# Index

1  Introduction .................................................................................. 1  
1.1  SOC DESIGN CHALLENGES .................................................. 3  
1.2  SYSTEM PLATFORMS ................................................................. 5  
1.2.1  HW subsystem classification .................................................. 5  
1.2.2  SW classification .................................................................. 6  
1.3  LIMITATIONS OF CONVENTIONAL DESIGN METHODS .......... 7  
1.4  ELECTRONIC SYSTEM LEVEL ................................................. 9  
1.4.1  System Level Design ............................................................... 10  
1.4.2  Open SystemC Initiative standards ........................................ 11  
1.4.3  SystemC abstraction levels .................................................... 12  
1.4.4  OSCI Transaction Level modeling styles ................................ 14  
1.4.5  Virtual prototypes ................................................................ 16  
1.5  THESIS CONTRIBUTIONS ....................................................... 19  
1.5.1  Problem statement ................................................................. 19  
1.5.2  Contributions ....................................................................... 22  
1.6  THESIS STRUCTURE ................................................................. 23  

2  Modeling methods for early architecture exploration ................... 25  
2.1  INTRODUCTION ..................................................................... 27  
2.2  RELATED WORK ...................................................................... 29  
2.3  CASSE METHODOLOGY AND TOOL STRUCTURE ................. 32  
2.3.1  Design Steps ....................................................................... 33  
2.3.2  Tool Structure ..................................................................... 36  
2.4  APPLICATION MODELING ..................................................... 38  
2.4.1  TTL Interface ...................................................................... 38  
2.4.2  TTL implementation on CASSE ............................................ 40  
2.5  ARCHITECTURE MODELING ................................................. 44  
2.5.1  Transaction Level Communication using ICCP .................... 44  
2.5.2  Processing Elements .............................................................. 45  
2.5.3  Storage Elements ................................................................. 48  
2.5.4  Network Elements ............................................................... 49  
2.5.5  External Components ........................................................... 50  
2.6  MAPPING ................................................................................ 51
2.6.1 Timing annotation.................................53
2.7 ADVANCED PERFORMANCE ANALYSIS ..... 55
2.8 MPEG-4 DECODER CASE STUDY.............57
2.8.1 Application and architecture modeling.....57
2.8.2 Mapping and analysis............................60
2.9 NETWORK-ON-CHIP CASE STUDY..............65
2.9.1 Mini-NoC platform.................................66
2.9.2 JPEG decoder application.....................66
2.9.3 CASSE Mix-Level model.........................67
2.9.4 MiniNoC RTL versus CASSE Mixed-Level comparison 69
2.10 CONCLUSIONS..................................74

3 Modeling methods for quick and reusable HW IP model creation 77
3.1 INTRODUCTION....................................79
3.2 RELATED WORK.................................80
3.3 ADVANCED MODELING CONCEPTS ...........81
3.3.1 Separation of concerns in the IP behavior core81
3.3.2 Hierarchical modeling.........................83
3.3.3 Dynamic layout..................................85
3.3.4 Timing modeling................................86
3.4 GENERIC MODELING FEATURES..............90
3.4.1 Behavior..........................................90
3.4.2 Storage and synchronization...............96
3.4.3 Generic core interfaces ...................99
3.4.4 Clock features.................................103
3.5 CONCLUSIONS..................................107

4 Modeling methods for protocol specific cycle accurate communication using TLM2.0 109
4.1 INTRODUCTION.................................111
4.2 RELATED WORK.................................112
4.3 METHODOLOGY FOR CA MODELING WITH TLM2.0 113
4.3.1 Protocol analysis...............................117
4.3.2 TLM 2.0 mapping...............................120
4.3.3 Transactor creation..............................124
4.4 ACCURACY AND SPEED RESULTS............127
4.4.1 Cycle accuracy verification approach....127
4.4.2 DTL point-to-point test........................127
4.5 CONCLUSIONS..................................131

5 Conclusions and future work..................133
5.1 CONCLUSIONS..................................135
5.2 FUTURE WORK.................................138

6 Resumen en español..............................141
6.1 Introducción......................................143
6.1.1 Diseño a nivel de sistema....................144
6.1.2 Estándares de la Open SystemC Initiative....145
6.1.3 Niveles de abstracción con SystemC..........146
6.1.4 Estilos de modelado a nivel de transacción con OSCI TLM 2.0 147
6.1.5 Prototipos Virtuales............................148
6.2 Contribuciones de la tesis doctoral...........150
6.2.1 Planteamiento del problema.................150
6.2.2 Contribuciones................................153
6.2.3 Estructura de la tesis doctoral.............155
6.3 Métodos de modelado para la exploración temprana de la arquitectura 156
Index of figures

Figure 1. System platform overview ................................................................. 6
Figure 2. SW stack .......................................................................................... 7
Figure 3. Conventional design flow ................................................................. 8
Figure 4. Abstraction levels and communication granularity ....................... 13
Figure 5. TL modeling styles ........................................................................ 14
Figure 6. Design flow using Virtual Prototyping technology ......................... 17
Figure 7. Separation of computation and communication on TLM models ...... 21
Figure 8. CASSE design flow ...................................................................... 32
Figure 9. Application, architecture and mapping description files example .... 35
Figure 10. CASSE structure ........................................................................ 37
Figure 11. TTL as parallel application model (a) and as HW/SW platform interface (b) ..... 38
Figure 12. Task example and task_if interface .............................................. 41
Figure 13. Channel implementation ............................................................... 42
Figure 14. Task state transition diagram and Suspend() primitive implementation ... 46
Figure 15. ConsumeTime method implementation ....................................... 48
Figure 16. PE, NE and SE structure ............................................................... 49
Figure 17. Architecture refinement via external components ....................... 50
Figure 18. Mapping and execution of tasks on a PE ....................................... 52
Figure 19: CTAP tool process ..................................................................... 54
Figure 20: Example of timing annotation applying CTAP ........................... 54
Figure 21: Transactions view example for a PE .......................................... 56
Figure 22. MPEG-4 application model ......................................................... 57
Figure 23. Initial architecture model ............................................................... 59
Figure 24: SRAM data load vs. time. ............................................................. 61
Figure 25: SRAM data load per task/port .................................................... 61
Figure 26. New architecture model and mapping ........................................ 63
Figure 27. Channel size vs. frame rate .......................................................... 64
Figure 28: MiniNoC Platform ...................................................................... 65
Figure 29: Compiling and executing an application in the MiniNoC platform ... 66
Figure 30: Multiprocessor implementation of the JPEG decoder ............... 67
Figure 31: CASSE Mixed-Level model ....................................................... 68
Figure 32: IDCT annotation example ........................................................... 69
Figure 33: Accuracy comparison ............................................................................................ 71
Figure 34: Simulation time comparison .................................................................................. 71
Figure 35: Simulation speed comparison ............................................................................... 71
Figure 36: Macro-blocks processing duration (latency comparison) ...................................... 73
Figure 37: Macro-block start and end processing time .......................................................... 73
Figure 38: Relationship between modeling libraries, models and VP .................................... 79
Figure 39: Layers of a SystemC HW IP model ...................................................................... 83
Figure 40: Hierarchical modeling concept .............................................................................. 84
Figure 41: Functionality refinement ....................................................................................... 86
Figure 42: Computation delays .............................................................................................. 90
Figure 43: Code example for a user defined function_thread ................................................ 92
Figure 44: Code example for a user defined function_method .............................................. 93
Figure 45: Code example for a user defined function_callback ........................................... 93
Figure 46: Code example for a user defined fsm_thread ..................................................... 95
Figure 47: Wrapper for the scml_bitfield .............................................................................. 97
Figure 48: User-defined arbiter implementation ..................................................................... 99
Figure 49: clock channel and clock_port usage and relationship ........................................ 100
Figure 50: Generic bus interfaces ......................................................................................... 101
Figure 51: Interrupt interface ............................................................................................... 102
Figure 52: Clock features example ....................................................................................... 103
Figure 53: Timing example code .......................................................................................... 106
Figure 54: IP reuse for different communication protocols ............................................... 114
Figure 55: Methodology overview ....................................................................................... 115
Figure 56: Signal attribute groups, timing points and protocol state machines ................. 116
Figure 57: Identification of timing points ............................................................................. 118
Figure 58: Timing points for the Command and Read groups of the DTL protocol .......... 119
Figure 59: Code for the DTL write channel payload extension ........................................... 122
Figure 60: DTL channels and phase flows ........................................................................... 123
Figure 61: Schematic of the initiator and target transactor templates ................................... 124
Figure 62: TLM 2.0 DTL Initiator Transactor ...................................................................... 126
Figure 63: TLM 2.0 DTL Target Transactor ......................................................................... 126
Figure 64: Testbench to test the DTL interface and transactors ......................................... 128
Figure 65: Waveform of simulation with a write and a read transaction ............................. 129
Figure 66: Speed measurements of the DTL point-to-point test setup ............................... 130
Index of Tables

Table 1. VP use-cases and requirements ................................................................. 18
Table 2: Maximum channel size ............................................................................. 58
Table 3. ITCP calls on Frontend/bits and Backend/frame_inf ports .................. 62
Table 4. ARM computational load per task ............................................................ 62
Table 5: DTL Protocol Groups .............................................................................. 117
Table 6: DTL Transaction Phases ......................................................................... 121
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>AHB</td>
<td>Advanced High-performance Bus</td>
</tr>
<tr>
<td>APB</td>
<td>Advanced Peripheral Bus</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application Specific Instruction-set Processor</td>
</tr>
<tr>
<td>AT</td>
<td>Approximately Timed</td>
</tr>
<tr>
<td>AXI</td>
<td>Advanced eXtensible Interface</td>
</tr>
<tr>
<td>BCA</td>
<td>Bus Cycle Accurate</td>
</tr>
<tr>
<td>CA</td>
<td>Cycle Accurate</td>
</tr>
<tr>
<td>CBD</td>
<td>Component Based Design</td>
</tr>
<tr>
<td>CCATB</td>
<td>Cycle Count Accurate at the Transaction Boundaries</td>
</tr>
<tr>
<td>CCM</td>
<td>Cycle Callable Models</td>
</tr>
<tr>
<td>CE</td>
<td>Consumer Electronics</td>
</tr>
<tr>
<td>CP</td>
<td>Communicating Processes</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycles Per Instruction</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAB</td>
<td>Digital Audio Broadcasting</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DTL</td>
<td>Device Transaction Level</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>PV</td>
<td>Programmer's View</td>
</tr>
<tr>
<td>PVT</td>
<td>Programmer's View with Time</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SCML</td>
<td>SystemC Modeling Library</td>
</tr>
<tr>
<td>SLD</td>
<td>System Level Design</td>
</tr>
<tr>
<td>SLS</td>
<td>System Level Synthesis</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TAC</td>
<td>Transaction Accurate Communication</td>
</tr>
<tr>
<td>TDMA</td>
<td>Time Division Multiplexing Algorithm</td>
</tr>
<tr>
<td>TL</td>
<td>Transaction Level</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Modeling</td>
</tr>
<tr>
<td>TTL</td>
<td>Task Transaction Level</td>
</tr>
<tr>
<td>TtM</td>
<td>Time-to-Market</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications Systems</td>
</tr>
<tr>
<td>UT</td>
<td>Untimed</td>
</tr>
<tr>
<td>VLD</td>
<td>Variable Length Decoder</td>
</tr>
<tr>
<td>VP</td>
<td>Virtual Prototyping / Virtual Prototype</td>
</tr>
</tbody>
</table>
As the integration capability of silicon technologies continues growing, embedded systems on a chip (SoC) are becoming more complex, adding more components and functionality into them. Moreover, nowadays Consumer Electronic industry competitiveness demands shorter development times for new products trying to enter the market. This is imposing an enormous pressure on development teams, who have to cope with more complex systems in less time. The ultimate goal of any design methodology or tool is to enable designers to create more complex systems-on-chip, in less time, and with more quality. Conventional design and verification methods applied broadly in industry can not cope any longer with the complexity of the designs and their ever-shorter time-to-market.

This chapter introduces how Electronic System Level (ESL) design methods and tools are being proposed as a complement to conventional solutions, to improve the productivity of the designers and to bridge the design and verification gaps. ESL is rapidly settling down thanks to standards such as
SystemC and TLM and their acceptance in EDA vendors and manufacturers. More concretely ESL is being applied on current design flows mainly via SystemC TLM Virtual Prototypes (VP). A VP can be applied effectively to several activities (use-cases) during the design cycle. However different use-cases have different requirements in terms of accuracy, simulation speed, etc., which makes very difficult to come up with a single modeling style that fits all these requirements.
1.1 SOC DESIGN CHALLENGES

**Increasing SoC complexity** – The International Roadmap for Semiconductors (ITRS) predicts that a single IC will be able to contain a billion of transistors by the end of the decade [1]. As the room available in the chip continues growing, SoC’s tend to be more complex adding more components and functionality into them. The complexity and requirements of multimedia, wireless, and automotive applications has been exploding over the last years and the productivity of design teams has difficulties to keep up the pace. Now this trend is even further accelerated by the demand for convergence products combining the disciplines of several application domains. Typical examples are the convergence of wireless communication and video and audio processing capabilities in cellular phones, and the entry of wireless and multimedia applications into the car.

As the performance demanded by new applications continues growing, SoC are becoming multiprocessor systems (MPSoC) adding more and more processing units that execute concurrently. Since different application domains and market segments have different requirements in terms of performance, cost, power consumption, etcetera, trading such requirements require different components with different properties (i.e. CPU, DSP, and dedicated HW). All these requirements lead to today’s heterogeneous multiprocessor platforms.

**Shorter Time-to-Market** - Nowadays electronic consumer market competitiveness and time-to-market (TtM) imposes that no matter how complex the design problem is, design time budget has to be kept as short as six months from initial specification to a final and correct implementation. However, shortening the time-to-market and at the same time meeting the requirements of the new application domains - such as multimedia, wireless, and automotive - require tremendous changes in design methodology. This is imposing an enormous pressure on development teams, which have to cope with more complex systems in less time. According to the Embedded Market Forecast, more than 50% of the new embedded system developments run late, whereas the 20% do not achieve the initial specification requirements or even are completely cancelled [2].
With the aim of reducing the development time, SoC design is moving in two complementary directions. First, design-reuse and Intellectual Property (IP) trading is considered a must when the components to be shared are close to the final implementation. Secondly, programmable processing units are being preferred instead of ad-hoc hardware blocks. As a consequence of this seek for programmable solutions the size of the embedded software available in the SoC is exponentially growing. This is happening to an extent that embedded software development is requiring more engineers than hardware design [3]. For instance, a typical multimedia device like a cell phone contains up to 2 millions of code, which requires nearly 70% of the development effort. This represents a dramatic increment since only a few years and the trend continues growing. Software content in multimedia mobile phones is expected to rise up to the 20 millions lines of code at the end of the decade.

**First time right silicon** - Another key factor in SoC design is the rapidly growing of the Non-Recurrent Engineering (NRE) and production costs. These increasing costs are moving manufactures towards parts that have guaranteed high-volume production (in the order of millions of units) from a single mask set. However, the greater cost of redesign (i.e. re-spin) is the increase of development time and the risk of missing the market window.

Extensive verification is one way to reduce the number of spins of a design. Typically, verification is accomplished with a whole suite of tools and teams of people at the end of the design cycle. Verification teams must have a complete knowledge of the system behavior at the cycle/bit level and to define a great number of scenarios and stimuli that reproduces such behavior. Normally, a written specification containing these scenarios and verification stimuli is created, which can take up to 50% of the design cycle. The cost and time spent in verification is expected to increase as SoCs integrate more components and functionality into them. Moreover, some studies show that up to 70% of the re-spins are due to functional bugs (i.e. differences between the specified functionality and the implemented functionality) [4].

**Design productivity and verification gaps** – Conventional design and verification methods and tools can not fully exploit the amount of transistor per chip available
today with the current fabrication technologies. According to Gartner Dataquest, when the initial 90-nanometer designs came out, none of the designs exceeded 50 millions gates – that is, none of the designs utilized even half of the number of gates that could be integrated by the new 90nm silicon process [5]. Thereby, there is an ever-increasing difference between the capabilities offered by the technology and the ability of the designers to fully exploit such capabilities. This effect is known as “design productivity gap”. Likewise, Dutt et al. [6] highlight the tremendous complexity faced by simulation-based validation of complex SoCs, estimating that by 2007, a complex SoC will need 2000 engineer years to write 25 million lines of RTL code and one trillion simulation vectors for functional verification. Thus, as SoC complexity grows there is an even bigger difference between the capabilities of the technology and the ability to verify such systems. This effect is known as “verification gap”.

1.2 SYSTEM PLATFORMS

A HW platform can be defined as a flexible architecture that can be configured to execute diverse applications within an application domain. As application domains are converging, HW platforms are not anymore a matter of IP integration but subsystem integration. This is depicted in Figure 1. Every subsystem can contain in turn tens of IP blocks and requires specific SW to provide its specific functionality. Therefore platforms are not only about HW components but they are more and more about the combination of HW IP and specific function SW. Hence a more appropriated definition is system platform.

1.2.1 HW subsystem classification

There are several sorts of subsystems that can be found in a system platform:

- **Peripheral subsystems**, which basically gather together a bunch of low-rate communication peripherals (e.g. Timers, UARTs, etc) on a low-bandwidth second-level shared interconnect (e.g. APB bus).

- **Memory subsystems**, which provide access to shared on-chip memory and/or off-chip memory controllers. Typically those subsystems are the most
critical to balance in terms of cost versus performance and use to be the bottleneck of the platform performance.

• **Specific subsystems**, which can be very heterogeneous and are specialized on performing specific functions like audio, video, modem, security, gaming, etc. Typically those specific subsystems contain programmable DSP’s, ASIP’s and/or ad-hoc HW accelerators. Moreover, they may have dedicated high bit-rate interfaces to feed data in the subsystem.

• **Host subsystems**, which basically contain a general purpose CPU and typically control the execution of the user applications running on the platform. Host subsystems allow the platform to be flexible enough to map diverse applications onto it.

![Diagram](image)

**Figure 1. System platform overview**

### 1.2.2 SW classification

Several levels of SW can be found in a system platform:

• **HW dependent SW (HdS)**. This level of SW typically refers to low-level drivers, HW abstraction layers (HAL), etc., which is used for a general purpose CPU to control the HW IP in the system.

• **Operating System (OS)**. Typically this kind of SW is built on top of the HdS and provides services to build the higher level SW layers.
- **Function SW.** This level of SW refers to more complex algorithms and codecs, such as MP3 for audio, MPEG4 for video, UMTS for modem, AES for security, etc, that enable end user applications to be built on top. Typically this SW uses the services provided by the OS and HdS layers.
- **Application SW.** This level is the end-user application. An example is the complete graphical interface and applications (games, browser, etc.) of a mobile phone. This SW is built on top of the services provided by the previous layers.

![Figure 2. SW stack](image)

### 1.3 LIMITATIONS OF CONVENTIONAL DESIGN METHODS

As depicted in Figure 3, a typical design flow starts with the specification phase where the customer/functional requirements are analyzed (typically by a system architect) and different alternatives for their implementation explored. At the end of this phase a paper architecture specification written in a natural language is created and passed to the implementation teams. The HW implementation team receives the paper spec and starts creating the HW platform. This comprises both the integration of existing HW IP and the creation of new HW IP. Typically HW implementation is carried out at the RTL level using HDL languages such as Verilog or VHDL. IP blocks are individually verified and once the platform is completely integrated its verification also starts at the RTL level using conventional techniques. After verification, the RTL code can be synthesized and goes through the backend
process until the final chip tape out is obtained. In the meantime, the SW development team also received the paper spec. Application SW and partially Function SW is independent of the new design and either it is already available or can be developed in parallel with the HW. HW dependent SW and parts of the Function SW is platform dependent and can only be developed once the HW platform (with all the new details) is available. Finally, both HW and SW are integrated together and the final design is verified. HW/SW verification comprises both the validation of the functional and non-functional requirements (i.e. performance, power-consumption, etc).

Unfortunately, this conventional design flow can not cope with the complexity of new SoC designs and their associated short design times:

- **Static analysis is not enough.** Most of the design decisions are made very early during the specification phase and, since they can not be validated until very late, wrong decisions have an enormous impact. Architecture decisions, as well as the best partition of the application functionality, are taken during this phase using static techniques (e.g. spreadsheets) or just based on the knowledge/intuition of the system architect. Unfortunately, these techniques are not suitable enough to deal with the complexity of new designs. Static techniques can not capture the complex use cases and interactions of the applications running on the designs nor their dynamic behavior.

- **Ambiguity of paper spec.** The paper spec resulting of the specification creates a gap between this phase and the rest of the design flow. Typically, a paper spec written in natural language is ambiguous and can be misinterpreted for the development and verification teams. Such ambiguity can produce functional bugs that, again, can be detected only at the end of the design flow.
• **Late HW/SW integration.** Since HW/SW integration can not be performed until the HW platform is available, this happens too late in the design flow. The main problem is that functional and non-functional mismatches with the specification can not be detected until the complete system is not executed together. Moreover, as new designs become heterogeneous multiprocessors with an important amount of SW running on them, the HW/SW interactions are also more complex and the chance for a first-time working design is very low.

• **First solution that works instead of best solution.** Typical HW/SW co-design tools try to verify both the HW and SW together before the chip tape out is performed. Such tools work at the RTL level for HW and at the C and/or assembly code level for SW. These co-design tools limit the size and complexity of the systems due to the enormous amount of details the designer has to handle and its tremendously slow simulations (in the order of few hundred cycles per second). Using these tools the designers selects the first system instance that works, instead of trying to find out the best implementation that fulfils the specified requirements.

Summarizing, the distance between the moment when the design decisions are taken and the moment when these decisions are verified is too long. Hence, iterations are not feasible and the risk of missing the design deadline and the market window very important.

### 1.4 ELECTRONIC SYSTEM LEVEL

Electronic System Level (ESL) design methods and tools are being proposed as a complement to conventional solutions, to improve the productivity of the designers and to bridge the design and verification gaps. A complete overview of the ESL field can be seen in [84] and [85]. ESL is rapidly settling down thanks to standards such as SystemC and TLM and their acceptance in EDA vendors and manufacturer industry [15]. More concretely ESL is being applied on current design flows mainly via SystemC TLM Virtual Prototypes (VP).
1.4.1 System Level Design

In order to deal with SoC complexity and to overcome the limitations of conventional methods, innovative system level design (SLD) and verification methods have been proposed by the research community for years. SLD methods can be grouped in three main categories:

- **Component-based design** (CBD) follows a bottom-up design approach, where complete architectures are built up assembling together pre-designed components. These components are interconnected and communicate with each other by automatically inserting wrappers among them, as in [7], or by using standard interfaces and bus protocols.

- **System-level synthesis** (SLS) methods, such as [9] and [14], move the designers towards working at a higher abstraction level starting from a behavioral description of the system. SLS follows a top-down design flow, where the architecture is generated from that behavioral description by gradually adding implementation details until the final implementation is reached.

- Finally, **platform-based design** (PBD) [10] is a meet-in-the-middle design approach. PBD combines a bottom-up approach for creating a predefined SoC architecture (a.k.a. HW platform) and a top-down approach to map the system behavior to the components of the architecture.

Roughly speaking, SLD methods have been focused mainly in three key aspects: moving designers towards working at higher abstraction levels above RTL, separating the various aspects of the design to allow a more effective exploration of alternative solutions, and allowing the progressive refinement of the system from abstract descriptions down to implementation. More in detail:

- **Abstraction.** As the designer’s abstraction level moved from standard-cells design using full-custom techniques to the RTL using hardware description languages in the 90’s, new design methods propose to elevate the designer’s abstraction level from RTL to a system-level. At the system-level basic elements are of a bigger granularity (e.g. IP blocks or complete subsystems) and they are described using high-level modeling languages.
• **Separation.** Separating the various aspects of the design problem beyond hardware and software is a key issue in SLD in order to overcome the limitations of conventional design methods. The concepts of orthogonalization of concerns [10], interface-based design [11] and the Y-chart approach [12] have settled down the basis to provide such separation in SoC design.

• **Refinement.** Progressively refining a design from an abstract description down to an implementation model is a clear strategy to reduce the design complexity [13]. Although such refinement process can be performed manually, real benefit comes when the process is automated via tools. Examples are the high-level synthesis tools that automatically transform a C description to a RTL implementation [83] or the code generation tools that create C or C++ code from UML descriptions.

### 1.4.2 Open SystemC Initiative standards

Standards are the key differentiator between academic SLD and the ESL concept adopted by industry. It has been foreseen that ESL design tools will drive the EDA market in the coming years, and next generation ESL tools will be supported by the continuously deployment of SystemC [18] and TLM [19] in both industry and academy. The Open SystemC Initiative (OSCI), controlled by a steering group composed of thirteen major companies in the EDA and electronics industry, promotes the development and standardization of both SystemC and TLM.

**IEEE 1666 SystemC standard**

SystemC is a modeling language that is intended to enable system-level design and IP exchange at multiple abstraction levels, for systems containing both hardware and software components. SystemC is based on the C++ programming language and extends its capabilities to enable hardware description. These extensions are achieved via a class library that provided powerful new mechanisms to model hardware elements, concurrency, time and reactive behavior. SystemC also provides a simulation kernel that allows simulating an executable specification of the design.
**OSCI TLM 2.0 standard**

In TLM the details of communication among computation components are separated from the details of their implementation. Communication is modeled as channels and transaction requests take place by calling interface functions of these channel models. The primary goal of TLM is to dramatically increase simulation speeds, while still offering enough accuracy for exploring and validating implementation alternatives at the higher levels of abstraction. The increase in speed is achieved by abstracting the number of events and amount of information that have to be processed during simulation to the minimum required. Besides this increase in speed, TLM reduces the amount of detail the designer must handle, therefore making modeling easier. TLM 2.0 proposes a standardized set of Interface Method Calls (IMC) to create TL models, a standard generic payload and protocol (GP) for modeling memory-mapped bus communication and mechanism to extend the standard payload with protocol specific information [20].

**1.4.3 SystemC abstraction levels**

As the designer's abstraction level moved from standard-cells design using full-custom techniques to the Register Transfer Level using hardware description languages in the 90’s, nowadays SoC complexity requires to elevate the designer’s abstraction level from RTL to a system-level. RTL simulations limit the size and complexity of the systems due to the enormous amount of details the designer has to handle and its tremendously slow performances (in the order of few thousand cycles per second). Several abstraction levels are needed to represent the SoC at the right level of detail required for different activities along the design cycle (e.g. if the level of detail is accurate enough for architecture exploration, then the simulation speed is too low for algorithmic analysis).

Three main abstraction levels can be modeled with SystemC:

- **Functional level (FL).** This abstraction level only cares about the functional blocks that compose an algorithm and their relationships. Communication among functional blocks is performed at the message level (ML). The
functional abstraction level is also known as algorithmic level or communicating processes (CP). There is not a standardized set of SystemC interface method calls to create models at this level.

- **Architecture level (AL).** The architecture level aims to capture both the functionality and the structure of the HW IP blocks in a HW platform. Models have to be bit-true and functionally correct, but micro-architectural details (e.g. control logic, data-path registers, etc) do not need to be considered at this level. Communication among HW IP blocks is performed at the transaction level (TL), where the interfaces represent the actual interfaces of the IP blocks.

- **Implementation level (IL).** This abstraction level aims to capture the implementation details of both the interfaces and internals of the HW IP blocks. Communication among HW IP blocks is carried out at the signal-level (SL). IEEE 1666 SystemC standard and its synthesizable subset has standardized a set of ports and channels to create signal-level models (e.g. sc_in, sc_out, sc_signal).

![Diagram of abstraction levels and communication granularity](image)

*Figure 4. Abstraction levels and communication granularity*
1.4.4 OSCI Transaction Level modeling styles

Transaction Level models belong to the AL SystemC abstraction level. TL models can be categorized as models where the communication between blocks is abstracted away from the RTL style (of setting events on individual pins or nets) to using function calls. However the term TLM does not imply any particular level of granularity with respect to the abstraction of time, structure or behavior, and therefore there is not a unique manner of creating models.

In order to tackle this problem OSCI has proposed a group of modeling styles to create TL models. These modeling styles differ from each other mainly on the time granularity for the communication. However adding more time information in a TL model implies more effort to create it and slower simulation speed. Therefore different modeling styles are appropriated for different use cases depending on their accuracy and speed requirements.

The modeling styles and definitions proposed for OSCI are the following:

- **Untimed (UT).** A modeling style in which there is no explicit mention of time or cycles, but which includes concurrence and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs.
Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style. As an example, Figure 5 shows how in UT style two consecutive transactions (T1, T2) occur in an ordered way but without consuming any time.

- **Loosely timed (LT).** A modeling style in which it is possible to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model as sampled at the timing points marking the boundaries of a transaction. A LT model should at least represent the minimal timing information to manage multiple threads in the absence of explicit synchronization between those threads and to support features necessary to boot an operating system (if applicable). In case a LT model contains enough timing information to accurately predict the functional state and the number of cycles (at certain key timing points as defined by the boundaries of the transactions), its modeling style can be also named as Cycle Count Accurate at the Transaction Boundaries (CCATB) [27]. As an example, Figure 5 shows how in LT style the same two transactions (T1, T2) now consume certain time. A side effect of this modeling style is that transactions occur sequentially, i.e. T2 can only starts once T1 has finished.

- **Approximately timed (AT).** A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. As an example, Figure 5 shows how in the AT style the original T1 and T2 transactions are now composed of two independent states (phases): request and response transactions. Using these two independent phases, multiple outstanding transactions can now be modeled (see overlapping between request2 and response1), overcoming the sequential nature of the LT style. Notice that in LT style transactions cannot overlap and therefore the absolute timing of executing T1 and T2 differs from the time obtained using AT. AT models enable effects such as pipelining and out of order communication to be analyzed.
• **Cycle accurate (CA).** A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to explicitly re-evaluate the state of the entire model in every cycle or to explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles. In Figure 5, it is depicted how in a CA style the transaction is composed of multiple states (or phases), which depends on the protocol being modeled. In this style individual protocol phases occurs in specific timing points and, for example, the time difference between the address and data phases, as well as idle cycles can be observed during the communication.

### 1.4.5 Virtual prototypes

A Virtual Prototype (VP) is a software model (typically written in SystemC) that emulates the functionality of the HW platform and to some extent (depending of the abstraction level) its timing. VPs are typically modeled at the Transaction Level in order to achieve the high speed required for the simulation of complex designs. Virtual Prototypes have several benefits when compare to other prototyping solutions (e.g. emulators, FPGA or RTL simulations). A VP can be *earlier available*. This is because SystemC TLM models are created at higher abstraction levels than RTL, which implies fewer implementation details and therefore can be created with less effort. Moreover since these models contain fewer details, changes can be done quicker and new design instances set up in shorter time than with other prototype solutions. This makes VP the more *flexible* solution. Finally being a SW solution a VP offers a better visibility on what is happening in the system than HW emulators or FPGAs. Furthermore since the level of detail is higher using a VP is easier to pinpoint a certain error (bug) than using RTL simulations. The only drawback of VP is their lower execution speed, for a same degree of accuracy, when compared to FPGA solutions. However VP simulation speed is typically two or
three orders of magnitude faster than RTL simulations (depending on the modeling style).

With a VP solution HW and SW can be integrated and verified together much earlier than with the other solutions. Functional and non-functional requirements can be validated, and bugs and bottlenecks detected and solved following much shorter iterations. This approach saves a considerable amount of time, reduces the risk of redesign and, therefore, of missing the design deadline and market window. Furthermore VP serves as an *executable representation* of the HW platform, which can complement the ambiguous paper spec. This reduces the risk of introducing errors during the implementation phase due to misinterpretation of the specification. VP bridges the gap between the specification phase and the rest of the design flow, since it can be used as a reference for the HW and SW development. New IP blocks can be developed at the RTL level using the VP as a system-level test-bench. This reduces considerably the number of stimuli and new scenarios that have to be created to test the IP block and, therefore, reduces the time and effort required for the RTL verification. Moreover the VP can contains enough details of the HW platform to allow the development of the HW dependent SW in parallel with the RTL development (i.e. a register-accurate memory map view).

![Figure 6. Design flow using Virtual Prototyping technology](image)

The following table summarizes the most important VP use-cases and main requirements in terms of availability, simulation speed and accuracy.
<table>
<thead>
<tr>
<th>Use-case</th>
<th>Availability</th>
<th>Simulation speed</th>
<th>Timing accuracy</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early architecture exploration</td>
<td>Specification phase</td>
<td>RTL x100</td>
<td>Approximately timed</td>
<td>Traffic generators, application mapping</td>
</tr>
<tr>
<td>Software development and optimization</td>
<td>Implementation phase</td>
<td>RTL x1000</td>
<td>Loosely timed</td>
<td>Fast Instruction Set Simulators, debuggers</td>
</tr>
<tr>
<td>HW development and optimization</td>
<td>Implementation phase</td>
<td>RTL x100</td>
<td>Approximately timed</td>
<td>HDL co-simulation</td>
</tr>
<tr>
<td>HW/SW functional verification</td>
<td>Pre-silicon verification phase</td>
<td>RTL x1000</td>
<td>Loosely timed</td>
<td>SW platform sign-off</td>
</tr>
<tr>
<td>HW/SW performance verification</td>
<td>Pre-silicon verification phase</td>
<td>RTL x10</td>
<td>Cycle-Accurate</td>
<td>HW platform sign-off</td>
</tr>
</tbody>
</table>
1.5 THESIS CONTRIBUTIONS

1.5.1 Problem statement

As described in the previous section, different VP use-cases have different requirements in terms of availability, simulation speed, and accuracy. A VP is mainly a composition of SystemC TLM models and therefore those requirements are heavily influenced by the characteristics of the models that are part of it. Ideally models must be accurate enough, fast enough and simple enough to create in order to fit all VP use-cases. However, reality shows that different simulation requirements are achieved only by using different type of models, that is, the right model for the right use-case. On the other hand, having to create and maintain a separated model for each use-case (with the elevated cost of having to keep all models consistent with each other) is obviously an overkill and would drastically reduce the benefits of VP technology. Therefore model reuse and refinement is a must.

A model can be represented in three distinct abstraction levels, as explained in Section 1.4.3: Functional Level, Architectural Level and Implementation Level. Moreover within the Architectural Level there are several modeling styles possible with TLM (LT, AT and CA), each fitting better a specific VP use-case.

Enabling a seamless path that allows refining models along the different abstraction levels and modeling styles, should provide the following steps:

1. Refining from a Functional Level model to an Architecture Level model.
2. Refining along the different modeling styles of the Architecture Level. This comprise the following sub-steps:
   - LT to AT model refinement
   - LT/AT to CA model refinement
3. Refining to an Implementation Level model from any of the levels above.
Function to Architecture model refinement

Typically, application functionality is refined into architecture models by means of embedded software that runs on instruction set simulators (ISS) or specific hardware IP blocks that have to be integrated into the system architecture. Creating embedded software and specific hardware models for a new design still requires a considerable effort that only makes sense if the right HW/SW partition is known beforehand. Hence, due to the tight coupling of functionality and architecture they lack flexibility with respect to the exploration possibilities. Having the capability to analyze and explore different options is very important at the beginning of the design process, especially for those designs where most of the architectural and mapping decisions still have to be taken. For that reason, modeling methods that allow separating application functionality from architectural descriptions, and their posterior mapping, are vital to improve the decision making process and the early exploration of what-if scenarios.

Architecture model refinement (LT, AT, CA)

Because of their different requirements, models created for a specific VP use-case cannot be usually reused for a different use-case. For example, TLM models created for “HW/SW performance verification” are to slow to be used for “SW development and optimization”. Likewise models created for “HW/SW functional verification” are not available soon enough (and lack flexibility) to be used in “Early Architectural Exploration”. The key concept to refine an architecture model along the different modeling styles (LT, AT, CA) and, hence, enable its reuse for different use-cases is to separate the IP behavior (computation) from the details of its (communication) interfaces. As a result of this split the resulting model is composed of two separated parts connected together by a generic interface. The two parts (see Figure 7) are:

- **IP behavior core**, which represents the computation part and contains the main functions/operations that the IP block has to perform.
- **IP interface adaptor** (a.k.a. transactor), which represents the communication part and contains all the protocol details that the IP block has to play.

The generic interface that connects together the IP behavior core with the IP interface adaptor can be considered as a modeling artifact. It is an element whose presence is only justified to allow this separation. Such generic interface has to be
abstract (i.e. it should not reflect any specific implementation details) and has to be neutral (i.e. it should not influence neither the functionality nor the timing of the two separated parts).

Both the computation and communication parts have a different cost (timing) associated to it. Refining each of these two parts separately with more timing information is the way forward to obtain more accurate models with a limited effort.

- Communication refinement can be achieved just by replacing the IP interface adaptor (transactors) with versions that contains more time information. Creating transactors with the LT modeling style for a certain protocol would be the first step in the refinement process. LT modeling is very simple and enough information can be found in the TLM 2.0 manual [20]. Next step it would be to refine to the AT modeling style. Again this step is straightforward following the TLM 2.0 rules. It is even possible to have a single LT/AT transactor since the interface in terms of protocol payload and number of phases is the same. The following refinement step would be towards CA style, desirably using a standard such as TLM 2.0. However this modeling style is not properly addressed for the standard, which breaks the proposed refinement strategy.
• **Computation refinement** can be achieved by replacing internal blocks described at a coarse algorithmic level granularity by behavior blocks modeled at a finer operation granularity, that is, blocks representing the micro-architecture details (e.g. control state machines, data-path, etc). Unlike for the communication refinement, there are not standard definitions or special modeling methods that can help to implement the refinement steps in the *IP behavior core*.

### 1.5.2 Contributions

This PhD thesis has the following contributions:

1. Modeling methods that allow the separation between functional and architecture models, enabling their posterior mapping using SystemC and TLM. This permits to evolve the typical HW/SW co-design approach with more sophisticated methodologies that fit better with the requirements of the *“early architecture exploration”* VP use-case.

2. Modeling methods that allow the separation between computation and communication in a SystemC TLM architecture model, as well as its further refinement along different modeling styles and VP use-cases. Sub-contributions of these modeling methods are:
   a. Provide innovative structured methods for behavior (computation) modeling and refinement with TLM. Moreover such methods reduce the effort to create speed optimal TLM models.
   b. Provide innovative methods for Cycle-Accurate (communication) interface modeling and protocol specific transactors using the TLM 2.0 standard.
1.6 THESIS STRUCTURE

This PhD thesis is structured as follows:

Chapter 1  is this introduction.

In Chapter 2  the modeling methods that allow separating and mapping functional and architecture models are described. This chapter also covers the requirements and techniques needed to support the “early architecture exploration” use-cases and presents a tool named CASSE that implement them. Finally two examples that prove the exploration capabilities and the accuracy of the results are presented.

In Chapter 3  the advanced modeling concepts to create speed optimal TLM models that can be refined for different VP use-cases are introduced. This chapter focuses on the techniques that allow modeling and refining of the computation part (IP behavior core) with low effort. Moreover an implementation example of the concepts (as a generic modeling feature library) is also disclosed.

In Chapter 4  the modeling methods to create protocol specific Cycle-Accurate communication interfaces and transactors (IP interface adaptor) using the TLM 2.0 standard are disclosed. This methodology is proven through an example for a complex bus protocol. Simulation speed and accuracy results are given for this example.

In Chapter 5  the main conclusions of this work and future work are presented.
During the SoC specification phase most of the design decisions are taken by the System Architect. The success of this phase depends on how well the decision taking process addresses the functional product feature description, within limits set by cost and schedule constraints. Wrong decisions can have an enormous impact in terms of Time-to-Market and therefore decisions need to be validated as soon as possible. As explained before, Virtual Prototypes can help to speed up and improve the quality of this validation. However, the typical concept of HW/SW co-design is not enough to achieve results early enough during the specification phase. This chapter aims to provide innovative modeling methods that improve the decision taking process early during the specification phase of a design.
2.1 INTRODUCTION

The modeling methods proposed in this chapter aim to help System Architects to explore and analyze application models running on an early available model of the HW platform. The main requirements to consider are:

- **Application modeling and mapping.** During the specification phase it is very unlikely that the application in its final stage (e.g. embedded SW and/or HW implementations) is available. Therefore, other techniques should be applied that allow modeling the application requirements (and desirable its functionality) in a more abstract way. Moreover, to allow an extensive exploration of scenarios, separation of concerns has to be applied. This means that the behavior, interfaces and cost (e.g. timing information) of the applications are modeled in an orthogonal way and are merged once the application is mapped on the targeted architecture.

- **A library of highly configurable generic architecture models.** During the specification phase, different HW architectures with different properties need to be explored. In order to have VP models early during the specification phase, IP models have to be quickly available and therefore the effort to create and assemble them has to be minimal. As a solution to this limitation a library of highly configurable generic models that emulate the common functionality and timing of several IP types has to be put in place. Such generic IP model can be used when the model for the specific IP is not available yet. Moreover these generic architecture models have to enable/support the mapping of application models onto them.

- **Advance performance analysis.** During simulations tons of different performance data that has to be interpreted and analyzed before deciding whether the system instance fulfill the requirements or not. Pruning this big amount of information in order to find performance bottlenecks and possible optimizations is not obvious for a multiprocessor system where multiple tasks are executing together. Therefore, new performance analysis techniques should allow identifying hot spots and bottlenecks in a complex design and
easily correlate them with the application model. This enables an easier and faster way to isolate and hence optimize the designs.

- **Short and fast iterations.** For exploration a VP needs to be created, configured, simulated, and the obtained results analyzed in a very short time. The shorter the iterations, the bigger number of different scenarios can be explored. More specifically the requirements are:
  - Small effort to change/configure the VP. User-friendly front-end tools and possibly some scripting capabilities are required to enable quick modifications in the VP architecture and configuration.
  - Fast simulations. At least an improvement factor of x1000 compared with RTL simulations is required to perform meaningful simulations during the exploration process. This leads us to simulation speeds in the range of several MHz.
  - Relative accuracy. Although absolute accuracy (compared with the RTL counterpart) is always wanted, most of the times certain level of accuracy can be traded in order to reduce the modeling effort and improve the simulation speed. For exploration purposes the concept of relative accuracy or fidelity is introduced, such as the VP can be used to evaluate, for instance, whether architecture A is better or worse than architecture B for certain aspects.

The rest of this chapter is structured as follows. Section 2.2 discusses the state of the art on methods and tools that aim to solve the problems mentioned above and their limitations. Section 2.3 gives an overview of the CASSE simulation environment [43][45]. CASSE is a SystemC-based system-level simulation environment that aims to ease and speed up the modeling and analysis of complex SoCs during the specification phase. Sections 2.4, 2.5, 2.6, and 2.7 explain in more detail the different aspects that compose the CASSE tool: application modeling, architecture modeling, mapping and analysis, respectively. Sections 2.8 and 2.9 present two case studies where the tool is used.
2.2 RELATED WORK

From the methodological point of view, current ESL tools, such as [52], [53], or [54], follow a component-based design approach, separating computation from communication in the implemented models, and are able to refine them from more abstract to more accurate descriptions. However, these tools cannot be applied before the partitioning of the application functionality into HW and SW modules is decided. In this regard, application functionality is added to the architectural models by means of embedded SW that runs on instruction set simulators (ISS) or cycle-callable models (CCM) of the embedded processor, or specific HW modules that have to be integrated into the system architecture. Creating embedded SW and specific HW models for a new instance still requires a considerable effort that only makes sense if the right HW/SW partition is known beforehand. Hence, due to the tight coupling of functionality and architecture they lack flexibility with respect to design space exploration (DSE) possibilities. DSE is very important at the beginning of the design process, especially for those designs where most of the architectural and mapping decisions still have to be taken. For that reason, SystemC-based methods and tools that elevate the abstraction level above HW/SW and allow separating application functionality from architectural descriptions, as well as their mapping and analysis, are vital to improve the decision making process and the early exploration of the design space.

Furthermore, current ESL tools lack important features for SW task modeling and scheduling capabilities required for modeling real-time operating systems (RTOS). This is mainly because of the limitations of the SystemC language. The modeling and analysis of RTOS parameters early in the design process is an extremely important part of the exploration phase, since its correct configuration might greatly affect global system performance. Currently, these tools deal with RTOS configuration and low-level SW development by using ISS models of the embedded processor, which reduces the gains in terms of modeling effort and simulation speed obtained through the usage of high level models. First, still a considerable effort has to be spent in porting the RTOS and the application SW to the embedded architecture. And second, ISS execute the SW in an instruction-basis that increases again the number of events that the simulation kernel has to handle, resulting in
slower simulations. Therefore, new modeling techniques and methods that allow modeling SW tasks and RTOS at higher abstraction levels are required.

The Y-chart scheme is a typical example of a methodology that applies separation of concerns [33]. The Y-chart eases the DSE process by modeling independently functionality and architecture, and later on combining them in a separated mapping phase. CASSE follows a Y-chart methodology where application functionality is separated from architecture, but also communication is separated from computation by means of a task level interface [34]. Similar Y-chart frameworks are Spade [13], Sesame [32] and VCC [48]. Spade and Sesame start with functional simulations of the application that is described in the form of a Kahn Process Network [49]. However, they apply trace-driven architectural simulations where the architectural models (annotated with timing and performance figures) are fed with traces obtained during functional simulations. VCC, although no longer maintained, was one of the first commercial examples of a system-level design environment. VCC is based on the Polis framework [36]. VCC uses different abstraction levels and successive refinement to allow fast DSE, including performance analysis in the first stages of the design cycle.

More novel frameworks are Metropolis and Daedalus. In [37] the Metropolis framework is presented. Metropolis provides a meta-model of computation that offers syntactic and semantics mechanisms to support functionality capture and analysis, as well as architecture description and mapping of functionality to architectural elements. Besides the function/architecture separation, Metropolis also proposes the separation between the capabilities of an architectural model versus the cost it bears when it implements a given behavior. Although quite valuable and innovative from the methodological point of view, all those frameworks are not based on SystemC. This is an important drawback for their interoperability and integration with typical ESL design flows adopted in the industry. The Daedalus framework [38] extends the work from Sesame [32] by adding automatic application parallelization from sequential C/C++ description [50] and automated synthesis and RTL integration towards FPGA technology.
Existing SystemC-based frameworks that apply similar techniques than CASSE are presented in [51], [39], and [40]. **CoFluent** [51] is a commercial tool that applies function to architecture separation and mapping. Similar to CASSE, this tool aims to help in the DSE phase early in the design process. Likewise, this tool covers abstract RTOS modeling including important properties such as scheduling policy, context switching delay, etc. This work is further discussed in [42]. CoFluent is based on the MCSE (Co-design Methodology for Electronic Systems) methodology. Its functional model describes a system by a set of functions (tasks) and relations between them. Unlike our work that uses an architecture-independent streaming-wise parallel programming model, CoFluent models tasks communication and synchronization using elements closer to final implementation (e.g. events, message queues and shared variables). Kogel et al. [39] presents the Virtual Architecture Mapping methodology that enables the quantitative evaluation of an application-to-architecture mapping by means of an executable performance model. Shared processing resources are modeled via the Virtual Processing Unit that allows the spatial mapping and execution of multiple tasks on a single element. Similar to our approach, this framework accelerates the exploration of a large design space by means of description files where individual timing annotations as well as the mapping are specified. This framework uses a general timed Communicating Extended Finite State Machine (tCEFSM) programming model. Finally, Madsen et al. [40] proposes a system-level modeling framework to model SoCs consisting of heterogeneous multiprocessors and network-on-chip structures. Similar to CASSE, this framework covers abstract RTOS modeling, but unlike our approach that also includes functional information, the application tasks are only represented as a set of timing budgets for processing and communication.
2.3 CASSE METHODOLOGY AND TOOL STRUCTURE

CASSE follows a typical Y-chart methodology where application functionality and architecture are independently modeled and combined in a separate mapping phase, see Figure 8. Quantitative information about the system execution is then obtained by means of simulations. The tool can be used to perform functional simulations of only the application model or performance simulations of the mapped application executing on the architecture model. After simulations the obtained information can be visualized and analyzed in order to guide further optimizations in architecture, application and/or mapping structure.

CASSE controls all stages in the design flow by means of textual description files. These description files are read and parsed by the tool during elaboration time in order to create and properly configure the desired system model. The result is an executable model that is simulated using the SystemC kernel. Hence, the tool simplifies the exploration of several scenarios by means of these description files that can be easily modified in order to create a new system instance. Since changing the description files does not require recompiling the existing models, extensive parameter sweeps can be performed easily using scripts. The internal structure of the tool is depicted in Figure 10.

![Figure 8. CASSE design flow](image-url)
2.3.1 Design Steps

**Application modeling** - CASSE applies a parallel programming model based on the Task Transaction Level (TTL) interface [34]. According to TTL, an application is described as a task graph where parallel tasks communicate with each other by means of unidirectional channels. A task is an entity that performs computations. Tasks are connected to the channels via ports, and they communicate and synchronize with each other by calling TTL interface primitives on their ports. Hence, TTL strictly separates computation and communication at the application level. TTL can be used both for developing parallel application models and as a platform interface for integrating HW and SW modules on a platform infrastructure. TTL only defines the interface primitives and their functionality, but leaves their implementation open to the designer. More details about TTL and the TTL implementation used in CASSE are provided in Section 2.4.

In the tool, tasks containing the application functionality are written in C/C++ (i.e. the functionality per task is fixed at compile time). However, the task graph structure (i.e. port to channel connections) and its configuration (e.g. channel size) are described in a separate text file. An example of the syntax is shown in Figure 9 for a simple example application. The tool uses this description file to instantiate and bind together tasks and channels creating an architecture-independent executable model of the application. Such application model can be simulated using CASSE in order to validate its functional correctness.

**Architectural modeling** - CASSE achieves easy and fast architecture modeling by means of a group of highly configurable predefined elements. Those elements provide generic functionality that can be parameterized in order to cover either individually or combined a broader range of architectural components. These predefined elements are: processing elements (PE), storage elements (SE), and network elements (NE). A PE models generic multitasking computational unit. A PE includes an abstract task scheduler model that supports different arbitration schemes and advanced features like interrupts and pre-emption. A SE models generic random access memory elements, such as register files or static RAM memories. Finally, a NE models generic shared bus interconnections including
programmable arbiter, address decoder, and optional input buffers. All predefined elements can be connected together in a ‘plug and play’ fashion by means of a TLM interface called ICCP (Inter-Component Communication Protocol). ICCP defines an abstract device level communication protocol, including a point-to-point TLM interface and a group of communication primitives between two entities named Initiator and Target. Both the ICCP interface and the library of predefined elements have been developed using the IEEE 1666 SystemC and the OSCI TLM 2.0 standards (see Section 1.4.2). Besides these predefined elements the architecture models can be extended with new functionality by means of external components (EC). These EC can be described at any abstraction level using SystemC and they can be seamlessly added to the architectural models as long as their interfaces are TLM2.0 GP compliant. More details about ICCP and the predefined elements are given in Section 2.5.

A separate description file is used in order to specify the architectural composition of the system (i.e. number of elements of each type, number of interfaces per element, and their interconnection), and its configuration (e.g. memory map, memory sizes, communication latencies per interface, task scheduler policy, etcetera). An example of such architectural description file is shown in Figure 9.

**Mapping and execution** - One of the main advantages of the tool as a unified environment is the straightforward mapping support of the application functionality onto the modeled architecture. That is, CASSE supports the direct mapping and execution of the TTL applications (i.e. tasks and channels) on the architecture models created with the predefined elements. Original source code of the tasks can be executed directly in the PE available in the architecture model. This technique is frequently called Host Code Emulation. HCE avoids the usage of accurate HW models and/or ISS models and, therefore, reduces the modeling effort and allows faster simulations. Nevertheless, timing delays reflecting the computational costs of the functionality has to be annotated into the tasks. This can be performed by either automatic methods like described in [46][47] or the timing information has to be extracted and annotated manually. More information about the mapping is described in Section 2.6.
As for the previous steps another description file is used for the tool in order to control the mapping procedure. An example of such mapping description file is also shown in Figure 9. The outcome of the mapping stage is an executable model containing the selected application/architecture instance. This executable model is executed using the SystemC kernel in order to validate both the functional correctness and the performance of the system.

Performance analysis - During simulations the tool obtains and records information about the system execution. This information allows the user to analyze and identify architectural bottlenecks and possible system optimizations at different levels. Based on this analysis further iterations might be carried out where both the...
application and the architecture models might be tuned or a new mapping selected. CASSE predefined elements are able to trace and record two different kind of information: performance metrics (statistics) and transactions (transaction recording is based on the SystemC Verification Library [55]). Such tracing support is possible since all predefined elements incorporate built-in monitors that can be individually enabled to automatically gather statistics and record transactions during their execution. Monitoring is also controlled via a separated CASSE description file. More details about performance metrics gathering and transaction recording are given in Section 2.7.

Refinement - Finally, once the expected requirements are fulfilled with a specific application/architecture/mapping instance the system is ready for implementation. HW modules can be progressively refined from more abstract to more accurate (even synthesizable) descriptions in SystemC, and verified within the architecture model just by replacing predefined elements of the tool libraries with ECs containing the accurate model. Likewise, SW modules might be directly taken into an embedded compiler and later integrated again in the system by means of an EC that integrates an ISS.

2.3.2 Tool Structure

As shown in Figure 10, CASSE is structured in three layers:

Front-end layer: the front-end layer serves as a user interface that controls the tool. This layer is composed of the user libraries and the description files. There are two user libraries: the tasks library contains TTL-compliant tasks composing the application and the external components library that contains user specific SystemC models to be added on the architecture model. As explained before there are three description files: the task-graph file, the architectural file, and the mapping file.

Back-end layer: the back-end layer implements the core functionality of the tool. Besides a parser that reads and interprets the description files, this layer contains also two specific libraries. The application library (APP) where the TTL interface is implemented and the architecture library (ARCH) where the group of predefined
elements are implemented. The tool is able to carry out two kinds of simulations: functional simulations and performance simulations. During functional simulations the tool only requires the task-graph file. Based on the information of that file the tool automatically instantiates and binds together tasks and channels (from the user and tool libraries, respectively), creating an executable model of the application. During performance simulations the tool reads and parses the task-graph, architectural and mapping files. Predefined elements (i.e. PE, NE and SE) and external components (EC) are automatically instantiated (from the respective libraries) and connected together following a modular approach according to the architectural file. Tasks and channels are allocated on specific PE and SE elements according to the mapping file. All elements are configured according to the task graph structure and the parameters specified in the description files. The outcome of this process is an executable model of the system instance.

**Kernel layer:** these executable models are then run using the SystemC kernel, which constitutes the third layer of the tool. During SystemC simulations execution traces and statistics can be recorded and dumped to output files for later inspection and analysis.
2.4 APPLICATION MODELING

2.4.1 TTL Interface

Efficient MPSoC design requires the usage of advanced integration technology and parallel programming models in order to ease the integration of HW and SW modules. For this purpose, a new programming model and platform interface for MPSoC design and integration called TTL has been proposed in [34]. As shown in Figure 11, TTL is a task level interface that can be used both for developing parallel application models and as a platform interface for integrating HW and SW modules on a platform infrastructure.

![Figure 11. TTL as parallel application model (a) and as HW/SW platform interface (b)](image)

The TTL interface is an abstract interface that offers well-defined semantics for modeling media processing applications. The basic elements in the TTL logical model are tasks, ports, channels, task graphs and schedulers:

Task graphs group multiple tasks into larger entities. Although execution inside every task is sequential, all tasks in a task graph might execute concurrently. The execution of the tasks is controlled by the scheduler entity. There may be more than one scheduler in a system, e.g., one scheduler per processor.
A task is a unit of work, typically consisting of computation and communication actions. A computation action is performed on private data such that it does not interfere with other tasks. In TTL three types of tasks are distinguished: processes, coroutines, and actors. A process is a task that has a main method that is invoked exactly once. Processes are preemptive tasks where all private variables and the point of execution are preserved upon implicit task switching (i.e. process cannot explicitly suspend itself). A coroutine is a cooperative task that, similar to a process, has a main function that is invoked only once. The difference with a process is that a coroutine can only suspend its execution via an explicit call to the scheduler. An actor is a fire-exit task that has a main method that can be invoked multiple times. Actors only support explicit task switching by returning from its main method.

Tasks communicate with each other by means of unidirectional channels. Channels contain tokens that are produced/consumed by the tasks. Communication among tasks has to be synchronized to inform them about the presence/absence of tokens (data) and/or token space (room) in the channels. Tasks are connected to channels via ports. Ports implement TTL interfaces and tasks communicate and synchronize with other tasks by calling TTL primitives on their ports. Since different applications and platform infrastructures may have different communication requirements, it is not likely that a single interface can satisfy all requirements. Hence, the TTL specification offers support for different communication styles by providing a set of different interface types. All interfaces are based on the same logical model, which enables interoperability across interface types. There are seven different TTL interfaces named: CB (Combined, Blocking), RB/RN (Remote, Blocking/Non-blocking), DBI/DNI (Direct, Blocking/Non-blocking, In-order) and DBO/DNO (Direct, Blocking/Non-blocking, Out-of-order). More details about the different TTL interfaces can be found in [34].

As defined by TTL, task graphs are created and configured by a global entity called the configuration manager (CM). The CM is in charge of creating/destroying tasks and channels, connecting/disconnecting ports to channels, and controlling the execution of the tasks. In this regard tasks can be started, paused, continued and stopped. The CM may dynamically reconfigure the task graph based on the occurrence of certain events. Each possible task graph instance is considered a
separated use case. More details about the TTL dynamic reconfiguration capabilities are disclosed in [35].

Summarizing, TTL specifies services for inter-task communication, multitasking and task graph reconfiguration. However, TTL does not specify how these services should be implemented, but leaves their implementation open to be optimized in each particular platform.

2.4.2 TTL implementation on CASSE

CASSE provides a particular (but configurable) implementation for the services and basic elements specified by TTL. These services and basic elements are implemented in such a way that TTL tasks can be re-used without changes both for application modeling and for integration on an architecture model. Application models can be functionally simulated and their correctness validated using the tool. Tasks can be seamlessly integrated on the architecture models through TTL shells included on the PE. Next subsections discuss more details about this implementation.

2.4.2.1 Task implementation

As shown in Figure 12, TTL tasks are implemented as a C++ class inherited from the task_if class interface. Tasks implement their functionality in the pure virtual main method of that interface. This main method is invoked automatically by the tool via the Run method, which is connected to a SystemC thread. As depicted, the three TTL task types (i.e. process, coroutine and actor) are supported in this implementation.

Tasks contain TTL ports (e.g. CinP and CoutP) through which they communicate and synchronize with other tasks (i.e. inter-task communication services). Ports invoke TTL primitives via the ttl_if interface registered to the task. Such ttl_if interface provides access to a TTL shell where the primitives for the different TTL interfaces are implemented. There exist two different implementations of the TTL
shell: one used during the functional simulations and another that is integrated in the PE.

Tasks provide explicit access to the multitasking services via the `suspend()` and `yield()` methods of the `task_if` interface. Those methods access a scheduler implementation via the `scheduler_if` interface. Such scheduler implementations also implicitly access services such as preemption and task switching. As for the TTL shell there exist two implementations of the scheduler: one for the functional simulator and another that is integrated in the PE. The scheduler implementation (based on the standard SystemC kernel) is further discussed in Section 2.5.2.

Furthermore, CASSE partially supports the dynamic reconfiguration services specified in TTL. Currently, task execution can be dynamically started, paused, continued and stopped from a CM unit integrated in the tool. Support for such dynamic execution is also part of the scheduler implementation, which is build as an intermediate layer between the tasks and the standard SystemC kernel. This implementation adds functionality that is lacking in the current SystemC 2.2 version, especially for the dynamic stopping and resetting of tasks during run time.

```cpp
class IZZ : public task_if {
public:
    int mapPorts() {
        BIND_PORT( CinP, sizeof (pixel) );
        BIND_PORT( CoutP, sizeof (pixel) );
    }
    void main() {
        pixel Cin[64];
        pixel Cout[64];
        while (true) {
            read (CinP, Cin, 64);
            for (unsigned int i=0; i<64; i++)  {
                Cout[zigzag[i]] = Cin[i];
            }
        }
        write (CoutP, Cout, 64);
    }
private:
    InputPortCB CinP;
    OutputPortCB CoutP;
};
```

```cpp
class task_if {
public:
    // Called by the SystemC Thread
    void Run() {
        switch (task_type()) {
        case PROCESS || COROUTINE:
            while(…) {
                try {
                    main();
                } catch(ts_stop_exception&) {
                    // Task is forced to stop, stop();
                    // when started again, reset the task
                }
            }
            break;
        case ACTOR:
            while(true) {
                try {
                    main();
                } catch(ts_stop_exception&) {
                    // Resume another task
                    yield();
                }
            }
            break;
        }
    }
    // Mandatory methods (have to be implemented in the task)
    virtual void main() = 0; // Task functionality
    // Defaults methods (can be override by the task)
    virtual void init() {
        virtual void end() {
            virtual void reset() {
                virtual int mapPorts() {
                // Methods available during task execution
                void yield();
                void suspend();
                void consumeTime(int cycles);
                void consumeTime(sc_time t);
                // Registration methods
                void register_port (PortBase* p);
                void register_scheduler (scheduler_if* sch);
                void register_ttl (ttl_if* ttl);
                // …
            }
        }
    }
    // Registration methods
    void register_port (PortBase* p);
    void register_scheduler (scheduler_if* sch);
    void register_ttl (ttl_if* ttl);
};
```

Figure 12. Task example and `task_if` interface.
2.4.2.2 Channel implementation

As explained, tasks are connected to channels via ports, which implement TTL primitives. Since all TTL interface types are based on the same logical model and different port types should be able to interoperate with each other, we choose an implementation in which a single channel administration scheme suits all TTL interfaces types. Channels are implemented as circular buffers in memory. These buffers are either allocated in the host PC memory for the functional simulator or in an SE for architecture modeling. In order to manage these circular buffers, a channel consists of two parts: the channel buffer (CHB) and the channel administration tables (CHAT). This is depicted in Figure 13.

The CHB, where the tokens are stored, is unique for the producer and for all the consumer tasks connected to it. Each channel keeps associated a producer CHAT for the port producing tokens into the channel and a consumer CHAT for every port consuming tokens from the channel. Channels can be of two types: unicast and multicast. Unicast channels connect a producer task to a single consumer task. Multicast channels connect a producer task to an arbitrary number of consumer tasks. Although by using multiple CHATs the administrative information of the channel is duplicated, this implementation allows choosing different mapping schemes in both shared and distributed architectures. CHATs contain static information about how to access the tokens in the CHB (reference), the size of the CHB, etc, and dynamic information to derive the status of the channel. Finally,
CHATs also contains cross-references to the CHAT in the other side of the channel, in order to update (i.e. synchronize) its values.

2.4.2.3 Ports implementation

In order to connect a port to a channel a reference to a CHAT has to be assigned to the port. TTL primitives use this reference to obtain the CHAT information in order to access the tokens in the channel and to update the channel status. However, ports do not implement that functionality themselves but rely on a TTL shell to manage the channels. Access to the TTL shell is provided via the \textit{ttl\_if} registered in the tasks. TTL shells are implemented following a layered approach, where different interface types are implemented separately from the channel administration. This separation is provided by means of a group of low-level basic primitives that contains common functionality to all interfaces types. These basic primitives are divided in four classes: test, transport, reference, and update primitives. The \textit{test} primitives are non-blocking methods used to check the fullness and emptiness status of the channels. The \textit{transport} primitives are used to transfer data from the tasks to the channels and vice versa. The \textit{reference} primitives are used to retrieve the current pointer to the channel. The \textit{update} primitives are in charge of bringing up to date the status of the channel. Finally, low-level basic primitives access to the channels depending on the platform infrastructure where the tasks are executed. This means either regular pointer addressing for the functional simulator or ICCP primitives for the architecture models.

Furthermore, the TTL shell is implemented in such a way that they support multiple tasks and ports in a single module. A taskport2chat table is available in the TTL shell containing an entry for each task/port pair assigned to it. Every entry stores a reference to the CHAT associated with the port.
2.5 ARCHITECTURE MODELING

2.5.1 Transaction Level Communication using ICCP

ICCP is an abstract device level communication protocol, implemented as part of the tool libraries, which defines a point-to-point TLM interface and a group of communication primitives between two entities named *Initiator* and *Target*. Basically, ICCP is used to interconnect all architectural components used in CASSE (i.e. PE, NE, SE, and EC) and allows the communication among them. ICCP is not a new device level communication protocol, but its aim is to model a generic transaction-level communication protocol that can be parameterized to emulate the timing and basic functionality of standard protocols such as AHB, OCP or AXI. As an example of its configurability, both the ICCP *Initiator* and *Target* entities can individually be configured with parameters such as interconnection width, latencies, clock frequency, and so on. The tool checks automatically the compatibility of the parameters of the *Initiator* and *Target* when two instances are connected together.

ICCP uses the AT modeling style as defined for the TLM 2.0 standard. Basically ICCP has extended the standard Generic Payload (GP) with specific extensions using the extension mechanism proposed by TLM2.0. These extensions are ignorable such as standard TLM 2.0 GP components can interact with the CASSE predefined elements. In addition to the *nb_transport_fw* standard methods, ICCP provides also convenient communication primitives that simplify and abstract the usage of the interface for the tasks running on the architecture models. These convenient communication primitives are used to transfer data structures of any type between predefined elements and/or external components. Moreover, there are two different kinds of primitives that differ in the accuracy of the communication. In the first group, called *single request, multiple data transfer (SRMD)* primitives, the transaction transfers the complete burst in one pair of request/response phases. In the second group, called *multiple requests, multiple data transfer (MRMD)* primitives, the transfer is composed as a chain of multiple request/response transactions with burst length equal to one. This derives in a more accurate cycle-by-cycle communication, but on the other hand decreases the simulation speed.
2.5.2 Processing Elements

Processing Elements model the generic computational units existing in a system, emulating their functionality and timing. By default, PEs do not contain any functionality, but they are simply placeholders where the task’s functionality is executed. An arbitrary number of tasks can be assigned (mapped) to a single PE, but only one task can be active at the same time. In this way, a PE can model both single-task computational units (e.g. dedicated HW coprocessor) and sequential multitasking computational units (e.g. embedded processor with an operating system).

As shown in Figure 16, PEs are composed in turn of several modules. These modules are the multi-task container (MTC), the task scheduler (TS), the interrupt controller (IC), the TTL shell and the virtual abstraction layer (VAL). Moreover, a PE contains a configurable number of Initiator ICCP interfaces and requires a single clock. Tasks are mapped into the multi-task container that is just a C++ standard vector of SystemC modules. Each SC_MODULE contains a single SC_THREAD and a pointer to a task interface. As mention in Section 2.4.2.1 the task_if interface contains methods to bind the tasks the different modules of the PE and to run the task. As stated before, more than one task can be mapped into the same PE, therefore, tasks execution and their access to the shared resources has to be scheduled. This is performed by the TS that provides methods to suspend and resume tasks based on SystemC events and the SystemC kernel. Basically, the TS assure that only a single task is running at the same time in a given PE. More details about the task scheduler are disclosed in Section 2.5.2.1. The IC activates an interrupt service routine (ISR) every time a write event happens on its interface. The ISR has to be modeled just as another task composing the application, but mapping a task on a PE as an ISR has to be explicitly indicated via the description files. Tasks running on the PE communicate with each other (i.e. with other tasks in the same or in a different PE) invoking TTL primitives at their ports. Therefore, when a task is mapped on a PE all its ports are bound to the TTL shell contained in that specific PE. Basically, this module transforms the logical TTL communication to the device level ICCP communication and handles all necessary data structures for communication and synchronization via TTL channels. The TTL shell is highly configurable to enable multiple data transport and synchronization schemes. Finally,
the VAL module enables other modules in the PE and all tasks running on it accessing any of the available *Initiator* ICCP interfaces existing in the PE.

### 2.5.2.1 Abstract RTOS modeling

As before mentioned, every PE contains an abstract TS model that can be configured to emulate the functionality and timing of a simple embedded RTOS. This TS assures that only a single task is running simultaneously on a PE by means of SystemC events. Basically, tasks are suspended by executing the SystemC `wait(task_event)` primitive on their associated SystemC event. Likewise, tasks are resumed by the scheduler using the `task_event.notify(SC_ZERO_TIME)` primitive. It is important to mention that the TS does not have a private SystemC thread, but it runs always in the thread of the active (awake) task. On the one hand, this improves the simulation performance by reducing the number of threads of the SystemC model [42]. But, on the other hand, switching between tasks requires that the active thread resumes the next task to be executed on the PE before suspending itself. Such task switching happens in the same delta cycle (i.e. in zero time). Therefore, in order to measure the cost in terms of time delay that might exist when swapping tasks, the TS can be configured with a *context switching delay* parameter. This parameter is specified in number of clock cycles and it is executed before the `notify/wait` methods. This is shown in Figure 14.

![Diagram](image)

**Figure 14. Task state transition diagram and Suspend() primitive implementation**

Moreover, in order to select which task has to be resumed next, different policies can be used in the TS. Besides the scheduler policy, other parameters such as task state, priority, enable/disable preemption, etc, influence the task selection. Task
states and the methods implemented on the TS for changing among states are shown in the transition diagram of Figure 14.

The different scheduler policies that can be configured in the TS are:

- **Multi Level Queue (MLQ), which** implements a configurable number of queues each with a different priority level. An important feature of MLQ is that it can be configured to implement different scheduling policies. Priority-based scheduling can be achieved by assigning to each task an individual priority level. Round-Robin scheduling can be achieved by assigning to all tasks the same priority level and enabling time slicing (number of cycles for the time slice has to be configured first). Finally, cooperative multitasking scheduling can be achieved by assigning to all tasks the same priority level and disabling time slicing. With cooperative scheduling tasks have to explicitly suspend themselves using the methods provided by the TS (i.e. `Suspend()` or `Yield()`).

- **Time Division Multiplexing Algorithm (TDMA), which** implements a slot table where both the number of slots and the slot size are configurable. Slot size is specified in number of cycles. Tasks can be arbitrarily assigned to the slot table in any combination, the TS is implicitly in charge of suspending the current task and resuming the next task when the slot time has been consumed.

The TS also provides methods to dynamically enable and disable both interrupts and preemption. Interrupts can be accurately detected and its associated ISR task resumed (if preemption is enabled) during the execution of others tasks. This is possible, since tasks have to invoke a special method of the TS to compute their computational delays. That is, the cost of the computational operations of the tasks is modeled by means of delays annotated in the task source code. This method is called `ConsumeTime` and its implementation is partially shown in Figure 15. The `ConsumeTime` method can take as argument either the number of clock cycles to compute or absolute time (sc_time). Underneath this method the actual time consumption is carried out using the `wait(sc_time)` primitive. An important feature of the TS is that the time consumption can be controlled by means of the `resolution time` parameter. This parameter indicates the granularity at which the `ConsumeTime`
method is executed. For instance, if \textit{resolution time} is set to 10 cycles and \textit{ConsumeTime(100)} is executed, then the delay is performed in 10 iterations of 10 cycles. During such iterations, the TS checks whether an interrupt has happened or whether the time slice of the task has expired. Of course a smaller \textit{resolution time} results in more accurate timing (e.g. for interrupt detection), but on the other hand simulation speed will decrease.

```cpp
void task_scheduler::ConsumeTime(int task_id, int ticks)
{
    setPendingTime(task_id, ticks);
    do {
        // Delay the task (max. resolution time)
        delay_ticks = min(getPendingTime(task_id), resolution_ticks);
        DELAY(delay_ticks);
        remaining_ticks = getPendingTime(task_id) - delay_ticks;
        setPendingTime(task_id, remaining_ticks);

        // Check if an interrupt has happened (if enabled)
        if(irq_is_enabled && preemption_is_enabled) {
            if(interrupt) {
                changeTaskState(task_id, Runnable);
                preempted_task = task_id;
                contextSwitchingDelay();
                isr_event.notify(SC_ZERO_TIME);
                wait(task_event[task_id]);
            }
        }
    } while(remaining_ticks > 0);
    setPendingTime(task_id, 0);
}
```

\textbf{Figure 15. ConsumeTime method implementation}

\subsection*{2.5.3 Storage Elements}

Storage Elements model generic random access memory elements, such as register files or static RAM memories. SEs can be configured with an arbitrary number of Target ICCP interfaces. This allows emulating the behavior of single, dual or multi-port memories existing in the system architecture. In the latter case, all Target interfaces can access simultaneously the memory data. Moreover, each
**Target** interface can be individually parameterized with its own clock frequency, latencies, interconnection width, etc. The SE structure is depicted in Figure 16.

![Figure 16. PE, NE and SE structure](image)

### 2.5.4 Network Elements

Network Element model generic shared interconnections, such as on-chip shared busses. NEs can be configured with an arbitrary number of **Target** (input) and **Initiator** (output) interfaces. Moreover, as depicted in Figure 16, an NE includes configurable input buffers, an arbiter module, an address decoder module and a controller module. The main functionality of an NE is to interconnect architecture elements. Basically, the controller module routes transactions from the **Target** interfaces to the **Initiator** interfaces. The controller module selects a specific input interface among all inputs with a pending request operation, based on the arbitration policy configured in the arbiter module. Afterwards, the controller selects an output interface through which to route the transfer, based on the transaction destination address and the information programmed in the address decoder module. Global timing of the transfer depends on the NE configuration, as well as the latencies programmed on both source and destination elements. Furthermore, when input
buffers are enabled multiple transfers might happen in parallel. That is, simultaneous transfers might happen from the source elements to the input buffers and from one of the input buffers to the destination element. Source elements are the ones that initiate the transfer (i.e. PE or EC). Destination elements are the ones where the transfer ends (i.e. SE or NE). Likewise, since an NE can be configured with two different clocks (one for all Target interfaces and another for all Initiator interfaces), they can be used to adapt two different clock domains. This requires that the input buffers are enabled.

2.5.5 External Components

External Components are simply placeholders that can include any SystemC model ranging from simple elements to complete sub-systems (e.g. a new interconnection network). Models integrated on an EC can be developed at any abstraction level from completely untimed models to cycle/bit-true RTL architectures. However, to be able to directly connect an EC with other predefined elements, they have to use ICCP or TLM 2.0 GP interfaces. Hence, adaptors might be required to adapt cycle/bit-true interfaces (using SystemC signals) to the TLM interfaces used by ICCP. According to the CASSE methodology, ECs are key to achieve a progressive refinement of the architecture models and hence to cover also the path to implementation. Basically, predefined elements can be progressively replaced with more accurate descriptions and added to the architectural model by means of an EC. This is shown in Figure 17.

![Figure 17. Architecture refinement via external components](image-url)
2.6 MAPPING

An important contribution of the tool is the straightforward mapping support of the applications onto the architectural models. Such feature eases the exploration of different partitioning alternatives with a minimum effort. Mapping is divided in four steps:

- **First, TTL tasks are mapped on processing elements.** Mapping a task on a PE means that the task is linked to one of the SystemC threads available in the MTC module and access to all methods in the TS, TTL shell and VAL modules is provided. As mentioned before, PEs provide a simple placeholder where tasks are executed. Tasks source code is directly executed in the PE. This reduces the modeling effort and allows faster simulations. As shown in Figure 18, tasks contain a *main* method that implements their functionality. This method is executed within the scope of the SystemC thread where the task is bound. Tasks can invoke TTL primitives on their ports (e.g. `tryReAcquireData(CinP...)` and `write(CoutP...)`), which are executed via the TTL shell available in the PE. Likewise, tasks can explicitly control their execution via calling methods in the TS (e.g. `Suspend(…)`). Furthermore, the time spent for the task during computational operations can be explicitly modeled via the `ConsumeTime(…)` function explained before.

- **The second step in the mapping procedure is to map the TTL channels on the storage elements available in the architecture model.** As specified in Section 2.4.2.2, channels are composed of the CHB (where data is stored) and the CHAT (used for data access and synchronization). CASSE allows mapping both the CHB and all CHATs on different SEs. This allows one to model multiple communication and synchronization schemes. For example, CHB and CHAT might be mapped on the same memory (typical for shared memory systems) or they might be mapped on different local memories (typical for distributed systems). Another important feature is that once all channel parts are mapped, the tool automatically calculates and fills all CHATs with the required information (e.g. cross-references). Likewise, the tool configures a special table in the TTL shell with the location (i.e. reference) of the CHAT associated to each port of
every task mapped on the PE. In this way, TTL ports can access their associated channels with no explicit information regarding the system architecture or mapping. This automatic process eases the designer’s work, since he/she does not have to deal with low-level details regarding the memory map or address calculation.

- The third step of the mapping procedure is to map the communication paths from the logical TTL ports of the tasks to the device level communication of the PE interfaces. CASSE allows specifying separately through which ICCP interface to access the CHB, the local CHAT and the remote CHATs for each port of every task mapped on the PE. The local CHAT has to be accessed in order to obtain the status of the channel (i.e. its fields are read and the number of tokens that are available/free calculated). The CHB is accessed to transport the data from the task to the channel or vice versa. Finally, the remote CHATs (i.e. the CHATs in the other side of the channel) have to be accessed in order to update the value of their fields (channel status). Such flexibility permits to model, for example, a PE with a dedicated memory storing all its local CHAT and a dedicated synchronization interface for accessing the remote CHATs, or two PEs communicating via a dedicated SE where the CHB, the local CHAT and the remote CHAT are stored (i.e. acting as a FIFO memory).
Finally, specific data structures belonging to a task can be mapped on a storage element. Tasks can access directly those data structures by means of the VAL module that provides access to the Initiators available in the PE. In order to avoid the need of changing the code of the tasks, ICCP primitives do not have to be directly invoked from the tasks to initiate the communication. Instead, CASSE implements smart pointers that can be templatized with the type of the data structure (including user defined types). These smart pointers implement operator-overloading techniques to access the Initiator modules of the PE, whenever data is read from or written to the smart pointer. Of course, mapping any single local variable of a task on a SE via the smart pointers can be a tedious process and, therefore, only specific structures for which their mapping can have a significant impact on the system performance should be mapped in that way.

2.6.1 Timing annotation

Timing annotation is a modelling technique where computation delays are inserted into the untimed application model to represent the cost of the operations after mapping. This timing information can be obtained from the RTL specification when mapped on HW and via analyzing the source code when mapped on SW. Typically this process is known as characterization.

In source code annotation for SW the basic idea is to instrument each C statement of the application model to determine the number of assembly instructions generated by the compiler for that statement. The resulting execution time in clock cycles is then obtained by multiplying this number of assembly instructions with an estimate of the average number of clock cycles per assembly instruction (CPI). The result can then be used again to annotate each C statement of the application in the tasks of the application model. This is depicted in Figure 19. The advantage of this approach is that an ISS is not required to execute the SW binary, which speed up simulations. Main disadvantage is the lack of accuracy for complex processor architectures, because estimations are based on average numbers of clock cycles per assembly instruction (discarding caching effects, pipeline effects, etcetera).
An example of a tool that performs such source-level automatic timing annotation is CTAP [46]. This tool provides the number of clock cycles that a C statement requires once is compiled. The tool provides this information by finding out the number of assembler instructions needed to execute such C statement. CTAP uses the compiler information to annotate back the source code with the duration information. This annotation is performed at each C statement indicating the duration information by means of `duration(X)` statements. The X number is the number of clock cycles needed to perform that C statement. A simple example is shown in Figure 20.

```c
//source code
1 int foo(int a, int b) {
2     int x, c;
3     for (x = 0; x < 8; x++) {
4         if (a == 1)
5             c += b;
6         else
7             c -= b;
8     }
9     return c;
10 }
```

```c
//annotated code
1 int foo(int a, int b) {
2     duration(5);
3     int x, c;
4     duration(2);
5     for (x = 0; x < 8; x++) {
6         duration(2);
7         duration(2);
8         if (a == 1) {
9             duration(3);
10             c += b;
11         } else {
12             duration(1);
13             c -= b;
14         }
15     } duration(2);
16     return c;
17 }
```
2.7 ADVANCED PERFORMANCE ANALYSIS

During performance simulations CASSE can obtain and record information about the system execution by means of built-in monitors available in the predefined elements. CASSE predefined elements are able to monitor and record two different kinds of information: performance metrics (or statistics) and transactions.

- Performance metrics give an overview of the status of the system at a specific moment of time. There are two basic metrics that can be obtained: time metrics (e.g. total execution time of a task on a PE) and quantity metrics (e.g. number of TTL primitives on a port of a task). Typically, such metrics are generated at the end of the simulation time in order to see the status of the system. Besides dumping at the end of the simulation time, the tool can be configured to produce and dump the statistics periodically (e.g. every 10 milliseconds). This periodicity can help in analyzing what is happening in the system during the simulation and in detecting dynamic behavior that would otherwise be unobserved.

- Transaction recording provides a representation about the occurrence of certain events and their duration during simulation. Events are associated with, for instance, start/end of a task execution on a PE or start/end of a write operation on a SE, and so on. Such representation can be used to analyze what is happening in the system during the execution time and might guide optimizations or help in the detection of bugs. Transaction recording in CASSE relies on the SystemC Verification library (SCV).

Performance monitoring can have a significant impact on the simulation speed of the architecture models. Thereby, instead of monitoring and recording all possible information regarding the system execution, the tool provides a fine grain controllability of what information to trace and where to trace it. Concretely, the tool enables to select individual elements and even individual interfaces within an element for monitoring. Likewise, a user can select to obtain performance metrics, transactions recording, or both for each individual predefined element or interface. This allows to trade-off the amount of information to gather against the simulation speed of the executable model.
Performance metrics gathering and transaction recording can be carried out at several levels within each predefined element. These levels are: task level, TTL communication level, and device level. At the task level the tool collects information regarding the execution of the tasks (e.g. time spent on computation or communication, execution rate, etc). At the TTL communication level the tool collects information regarding the TTL primitives (e.g. number of primitives of each type, latencies for their execution, synchronization overhead, etc). And finally, at the device level the tool collects global information related with the predefined elements (e.g. communication load per interface, communication latencies, etc). For transaction recording the relation among the three levels is shown in Figure 21. Since analyzing metrics and transactions based only on textual descriptions is a quite inefficient and a tedious task, CASSE also generates different formats that can be visualized using standard wave/transaction viewers.

**Figure 21. Transactions view example for a PE**
2.8 MPEG-4 DECODER CASE STUDY

In order to provide a more detailed view of the tool capabilities, the CASSE design flow is applied on the modeling and analysis of an MPEG-4 decoder [55]. This case study is part of the ARTEMI project (ARchiTEctures for Multimedia and Internet) [56][57], which aims to develop a system for receiving low-quality digital video transmitted over the Digital Audio Broadcasting (DAB) network, using as target technology mixed programmable platforms. A key part of the ARTEMI system is a MPEG-4 decoder, which has to be implemented on the Altera Excalibur FPGA platform. This platform is composed of a million equivalent gates programmable logic device (PLD) and an embedded processor subsystem (Excalibur Stripe). Instead of focusing on the final implementation of the MPEG-4 decoder on the Excalibur platform, this section focuses on how CASSE is applied at the very beginning of the design process to obtain meaningful information that can guide the implementation phase.

![MPEG-4 application model](image)

2.8.1 Application and architecture modeling

The first step in the CASSE design flow is to decompose the sequential MPEG-4 decoder reference code into a group of concurrent tasks communicating with each other using the TTL primitives. The obtained process network, see Figure 22, is composed of seven tasks (Frontend, VLD, IQUANT, IDCT, MVDEC, CMOV and Backend) and ten channels (four multicast and six unicast channels). This
application model is then simulated using CASSE at the functional level and the results are verified against the reference code. At this level, the tool can also be used to derive the maximum channels size required for the application, which minimizes the number of times tasks are blocked. In order to achieve this, the tool dynamically increases the channel size whenever there is not enough space available to write in the channel. In Table 2, maximum channel size for a GOP (Group of Pictures) with a sequence pattern IPBBPBB – being (I) Intra, (P) Inter, and (B) Bidirectional prediction frames - and QCIF size is shown.

<table>
<thead>
<tr>
<th>Channel</th>
<th>Token size (bytes)</th>
<th>Number of tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>2</td>
<td>1901</td>
</tr>
<tr>
<td>VOL_data</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>VOP_data</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>MB_data</td>
<td>8</td>
<td>696</td>
</tr>
<tr>
<td>motionMB_data</td>
<td>48</td>
<td>594</td>
</tr>
<tr>
<td>Blk_VLD</td>
<td>264</td>
<td>1129</td>
</tr>
<tr>
<td>Blk_IQ</td>
<td>260</td>
<td>1129</td>
</tr>
<tr>
<td>Blk_IDCT</td>
<td>256</td>
<td>1129</td>
</tr>
<tr>
<td>MV</td>
<td>68</td>
<td>425</td>
</tr>
<tr>
<td>frame_inf</td>
<td>152072</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total memory required:</strong></td>
<td><strong>1.860.058 bytes</strong></td>
<td></td>
</tr>
</tbody>
</table>

Although the partitioning of the reference application into separated tasks requires some effort, it has also clear benefits. First, the obtained application model remains architecture-independent and tasks might still be selected for execution either in HW or SW modules. And second, there is clear separation between communication and computation inside each task, where data transport and synchronization points are made explicit by means of TTL calls. Thanks to this separation, time occurrence of those synchronization and communication events can still be identified once the application is mapped onto the architectural model. This helps in analyzing whether the application is fulfilling all its timing constraints.
Second step in the design flow is to model the HW platform on which the application is executed. In this case, our goal is to create a model that emulates the Excalibur Stripe architecture using the predefined elements and interfaces available in the tool libraries and configuring them accordingly. The Excalibur Stripe architecture model is shown in Figure 23. RAM memories are modeled using generic SE, and configuring them with the right sizes, number of Target interfaces, latencies, etc. AHB busses, bridges, and memory controllers are modeled using generic NE. NE modeling AHB busses (i.e. AHB1 and AHB2) are configured with no buffered input ports and round-robin arbitration. Likewise, right latencies according to the AHB specification and response address-range for all output ports are programmed reflecting the real memory map of the Excalibur Stripe. NE modeling bridges (i.e. AHB, PLD2AHB, and AHB2PLD) are configured with buffered input ports. The ARM9 CPU is modeled as a generic PE with a single ICCP interface. Since the final implementation will contain an uC/OS operating system running in the ARM9, the task scheduler of the PE is configured accordingly to match as close as possible its behavior (e.g. scheduling policy, context switching delay, etc). Besides modeling the embedded stripe of the Excalibur device, a video input (VIN) and a video output (VOUT) coprocessors, which will be implemented in the PLD part of the device, are also modeled using generic PE. Finally, clock information is attached to the elements based on its clock domain. There are two basic clock domains, the Stripe...
domain with a 150 MHz, and the PLD domain with an estimated 50 MHz clock frequency. Clock domains are adapted using buffered NE, that is, the bridges of the architecture. Using CASSE such complex architectural model is quickly created and configured by means of an architectural description file that only takes 195 lines.

2.8.2 Mapping and analysis

2.8.2.1 Initial mapping: SW solution

The next step is to map the tasks and channels composing the MPEG-4 application on the initial architecture model. In this first approach, all channels (i.e. CHB and CHAT) are mapped on the SRAM memory located in the Stripe model. Likewise, this initial mapping locates the *Frontend* (input coded video) and *Backend* (output decoded video) tasks on the VIN and VOUT PE, respectively. The remaining tasks are mapped on the ARM PE. This procedure is rapidly described by means of the mapping description file, which only requires 60 lines. Performance simulations are then carried out. Computation delays were manually annotated in the SW tasks running in the ARM PE. In order to be as accurate as possible these computation delays were obtained after compiling with the CTAP tool (see Section 2.6.1). Besides providing information about the system performance this simulation can be used to assess functional correctness, that is, to check whether the application is still providing the same results when mapped onto the architecture model. For this example, CASSE took around 90 seconds to simulate one second of the system execution. For a clock reference of 150 MHz the simulation performance (or simulation speed) reaches approximately 2 Mcycles/s that is four orders of magnitude higher than typical CA simulations.

For this initial architecture and mapping the modeled system is able to decode 12 frames within this second of execution. During this simulation, traces are set to measure the data load in the SRAM memory where the channels are mapped. Figure 24 shows the data load in the SRAM memory sampled every 100 milliseconds of execution time and its relationship with the decoded frame. Total data load measured on the SRAM is approximately 50 Mbytes. However, this raw data of 50 Mbytes is not enough to provide information about what is happening in
the system and to guide further optimizations. For that reason CASSE allows to tag all transactions happening in the system architecture with an individual identifier. This identifier is associated to each port of each task running in the system. For instance, the data load produced for the port connected to the channel bits of the task Frontend can be individually observed in the SRAM memory. Figure 25 shows the data load produced into the SRAM memory in a task/port basis. Analyzing this information, it is observed that the port bits of the Frontend task and the port frame_inf of the Backend task produce two third of the total data load in this memory.

![Figure 24: SRAM data load vs. time.](image)

![Figure 25. SRAM data load per task/port](image)
Likewise, Table 3 shows the number of ITCP calls on those ports. This table derives the significant amount of test operations compared with the transport or update operations. Therefore, it is deducted that most of the load produced on those ports is due to reading their CHAT information. This means that those tasks are too often blocked waiting for data or room in their channels, and such high load is produced whereas polling the channel status. Such synchronization overhead is too high and might be too costly in terms of latency and power consumption due to the large path of busses and bridges that the data has to cross from the VIN and VOUT coprocessors to the SRAM memory. Hence, a first optimization is to reduce the impact of such synchronization overhead on the architecture.

Table 3. ITCP calls on Frontend/bits and Backend/frame_inf ports

<table>
<thead>
<tr>
<th>Task/Port</th>
<th>Test</th>
<th>transport</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frontend/bits</td>
<td>2.427.830</td>
<td>5.854</td>
<td>5.854</td>
</tr>
<tr>
<td>Backend/frame_inf</td>
<td>2.408.838</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4. ARM computational load per task

<table>
<thead>
<tr>
<th>Task</th>
<th>Cycles</th>
<th>% Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLD</td>
<td>14.676.715</td>
<td>7,8</td>
</tr>
<tr>
<td>IQUANT</td>
<td>28.263.398</td>
<td>15</td>
</tr>
<tr>
<td>IDCT</td>
<td>67.551.091</td>
<td>36</td>
</tr>
<tr>
<td>MVDEC</td>
<td>245.888</td>
<td>0,13</td>
</tr>
<tr>
<td>CMOV</td>
<td>75.907.139</td>
<td>40</td>
</tr>
<tr>
<td>Context Switching Overhead</td>
<td>1.104.250</td>
<td>0,6</td>
</tr>
</tbody>
</table>

2.8.2.2 First iteration: architectural optimizations

With the aim of reducing the communication between the coprocessors and the SRAM memory, local memories are added to the VIN and VOUT coprocessors and their associated CHAT are mapped into them. Now tasks do not have to use the complex multi-level bus infrastructure of the Stripe to read their CHAT information but they do it locally. Unlike conventional tools, in CASSE such modification simply
requires adding and modifying a few lines into both the architectural and mapping description file. After running new simulations the data load into the SRAM memory has been reduced by 77%, that is, from 50 Mbytes to 11.5 Mbytes. However, this decrease on communication load does not produce an increase on the decoding frame rate of the system that remains being 12 fps.

2.8.2.3 Second iteration: HW-SW solution

Next iteration is intended to increase the decoding frame rate of the system. Hence, more computational resources, where to execute some of the task running on the ARM, are added to the architectural model. In order to decide which task should be mapped in a separated PE, the tool derives the percentage of time the ARM is used for each task. That information is shown in the Table 4.

![Figure 26. New architecture model and mapping](image)

According to those results, both the CMOV (motion compensation) and IDCT (inverse discrete cosine transform) tasks are the more computational expensive tasks running on the ARM, respectively. However, since the IDCT is more suitable for a HW implementation, a new PE executing the IDCT task is added to the
architecture model. Channels belonging to the IDCT task are mapped onto the DPRAM memory that has a second port available for direct access from the PLD area. The IDCT PE and the DPRAM SE are connected together by a direct ICCP link using this available second port. Source code of the IDCT task is then annotated with new delays taking into account a HW implementation. We estimate that an IDCT coprocessor might process an 8x8 block in 128 cycles (the previous SW IDCT implementation needed around 1000 cycles in the ARM9). This new architecture and its corresponding mapping are shown in Figure 26.

![Figure 27. Channel size vs. frame rate](image)

With this new architecture and mapping instance, performance simulation derives a decoding frame rate of 14 fps. In order to squeeze the possibilities of this new platform model, a brief exploration of the channels size and their relation with memory usage and frame rate is performed. Using CASSE this exploration only requires feeding the tool with different task-graph description files that change the size of the channels. This analysis is shown in Figure 27, where it is observed like increasing channels size to 100 tokens increases the performance in one extra frame, whereas the total memory used for the channels remains within the maximum memory available. It is important to mention that the results shown in this case study are just brief examples of the tool capabilities since many others architectures and/or mapping might be explored, and much other information can be obtained which can guide further optimizations.
The MiniNoC [60] consists of a very simple MPSoC platform with four processor nodes interconnected with a NoC (network on chip) component [59]. The components of the MiniNoC platform are described in a RTL specification, thus the platform is synthesizable and 100% accurate. However, there are also disadvantages: it needs a high modeling effort and simulations are extremely slow. This fact makes the current model unsuitable for performance analysis and exploration activities - where many simulations have to be carried out following an iterative approach. In order to improve the simulation speed and to ease the modeling of multiprocessor systems, the CASSE environment is used. This exercise shows the capabilities of CASSE to integrate components at the RTL level and to perform mix-level simulations. CASSE predefined elements are combined with the NoC component of the MiniNoC platform in order to perform an experimental evaluation in terms of speed up of the simulations and accuracy of the results. The goal is to demonstrate the advantages of using CASSE for NoC-based MPSoC exploration.

Figure 28: MiniNoC Platform
2.9.1 Mini-NoC platform

The *MiniNoC* platform is composed of four homogeneous tiles containing each a *mMIPS* processor. Each time communicate with each other via the *NoC* network which contains four routers. The *NoC* component consists of a torus network with two nodes wide and two nodes high that follows an E-cube routing. The complete *MiniNoC* platform is shown in Figure 28.

In order to execute an application on the *MiniNoC* platform the first step is to compile such application for the *mMIPS* processors with the LCC C compiler. After that, the binary are loaded in the *mMIPS* processors models and the complete platform is executed with the SystemC simulator. The *MiniNoC* components described in SystemC are compiled with the standard GCC C++ compiler. This is depicted in Figure 29.

![Figure 29: Compiling and executing an application in the *MiniNoC* platform](image)

2.9.2 JPEG decoder application

The application chosen to perform the experiments is a simple JPEG decoder. A multiprocessor implementation of the JPEG decoder is created for the *MiniNoC* platform [61] (see Figure 30). The decoder takes the compressed image data as its input. It then subsequently applies a variable length decoding (VLD), zigzag scan
[ZZ], dequantization [DQ], inverse discrete cosine transform [IDCT], a color conversion and reordering to it. It then obtains the reconstructed image.

Figure 30: Multiprocessor implementation of the JPEG decoder

2.9.3 CASSE Mix-Level model

This chapter presents the CASSE Mixed-Level model and the steps followed in its implementation: application modeling, architectural modeling, mapping and time annotations.

2.9.3.1 Application modelling

In order to map an application to the CASSE predefined elements, the tasks of the application need to be compliant with the task interface of CASSE. According to that, the JPEG decoder tasks have to inherit from the class `task_if` (see Section 2.4.2.1). Moreover the MiniNoC platform uses message passing to communicate among processors, therefore it is necessary to adapt the specific message passing communication primitives used by the JPEG decoder tasks to comply with the ITCP communication protocol of CASSE.
2.9.3.2 Architecture modelling

The four processor tiles of the mix-level model are modelled using CASSE predefined elements. That is, each \textit{mMIPS} processor is modelled with a PE, the local memory with a SE and the local bus (\textit{MEMDEV} module) with a NE. The \textit{mNoC} RTL component is integrated as an External Component named NOC. This component contains the four routers and network interfaces of the original \textit{MiniNoC}. The next step was to connect the NOC module with the P tiles. Since the \textit{MiniNoC} is described at signal level (RTL) and, hence, did not comply with the ICCP protocol, an adaptor module is created to connect the tiles with the NoC element. The complete Mixed-Level model is shown in Figure 31.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure31.png}
\caption{CASSE Mixed-Level model}
\end{figure}

All modules in the model are instantiated, connected and configured in CASSE via the "architectural description file". The complete platform is described in around 100 lines of code.
2.9.3.3 Mapping

As explained before, the JPEG decoder is divided into three tasks. Each task runs on a separate tile (node). This is shown in Figure 30. Node1 is at (X, Y) = (0, 0), Node2 is at (1, 0), Node3 is at (0, 1) and the Node4 (1, 1) remains unused. This mapping is described by means of the “mapping description file” used for CASSE, requiring only 3 lines of code (each line maps a task on its respective processor).

Moreover the tasks are annotated with timing delays by applying the CTAP tool, as described in Section 2.6.1. An example of the annotated IDCT function is presented in Figure 32, in which each C statement adds the timing information by using the `duration()` function. The `duration()` function calls in turn the `ConsumeTime` function of CASSE. Since no pipeline effects can be observed with the CTAP tool. A corrective CPI factor (k) of 1.3 is used in the annotations.

![Figure 32: IDCT annotation example](image)

2.9.4 MiniNoC RTL versus CASSE Mixed-Level comparison

The MiniNoC platform and the CASSE Mixed-Level model are presented in the previous chapters. At this point the comparison of both platforms can be carried out. Two aspects are taken into account to compare the simulation speed and accuracy of the CASSE Mixed-Level model against the MiniNoC RTL model. These two aspects are:
• Execution of the complete JPEG decoder for a set of images. Metric to compare are total number of clock cycles (accuracy), and the number of clock cycles that are simulated per second of simulation time (speed).

• Start time and duration for each macro block processed in a specific image. This metric enable to judge the accuracy in intermediate points of the simulation and not only at the end.

2.9.4.1 Execution of the complete JPEG decoder application

A total of seven images of different sizes are compared. A Pentium IV 3.2GHz processor running GNU/Linux with 4096 MB of RAM is the host machine used.

Figure 33 depicts the comparison in terms of total number of cycles to execute the JPEG decoder application. This figure shows an average accuracy for all the images around 95%. Figure 34 presents the simulation time in seconds needed to execute the decoder for all seven images. As shown in the logarithmic graphic, the Mixed-Level model simulation time is in average almost one and a half orders of magnitude faster than the RTL simulation. This is surprisingly positive since the Mixed-Level model still contains a RTL component that act as the bottleneck for the simulation. Finally, Figure 35 depicts the simulation speed comparison between the two models. The results reveal an average improvement factor of 34 for the Mixed-Level model when compared to the RTL execution. Summing up, the results show in average a CASSE mixed-level model 34 times faster than the RTL model whereas maintaining the 95% of accuracy.
Figure 33: Accuracy comparison

Figure 34: Simulation time comparison

Figure 35: Simulation speed comparison
2.9.4.2 Latency and duration of individual macro blocks

As explained in Section 2.9.3.1 the JPEG process has been divided in three nodes. The input image applied to the decoder is separated in a number of macro-blocks that describe a certain region of the image. The splitting process is performed at node 1, which sent the macro-blocks blocks through the $mNoC$ to node2. The blocks are received and processed at node 2 and then forward to node 3, where they are processed and reordered into a final image. The objective of the experiment is to compare the intermediate moments when the macro-blocks are processed and the number of clock cycles that takes to process them (latency) along the three parallel nodes. The experiment is done for an image with a total of 12 macro-blocks.

Figure 36 represents the processing time for each macro-block in the image. Results show an average accuracy in the duration of the individual macro-block above 90% for the Mixed-Level model. Figure 36 depicts the start and end processing times for each macro-block in the image. This figure shows some deviations between the execution of the RTL model and the CASSE mixed-level model, although overall the time lines of both executions are quite similar for most of the macro-blocks. The main reason for this deviation is in the node1 before the macro-blocks are sent to node2. This deviation is produced by the highly data dependent VLD algorithm for which the default CPI factor is not enough to capture the dynamics involved in the processor data-paths.
Figure 36: Macro-blocks processing duration (latency comparison)

Figure 37: Macro-block start and end processing time
2.10 CONCLUSIONS

Contributions of the work presented in this chapter are twofold.

First, enabling the separation between the application functionality and the architecture, as well as their posterior mapping, allow evolving the typical HW/SW co-design approach with more sophisticated methodologies that fit better with the requirements of the “Early Architecture Exploration” VP use-case. This is further supported by the CASSE framework implemented in this work, which simplifies the iterative process of setting up the HW platform model, map the application model, configure the overall system, simulate and analysis the obtained results.

Second, the host-code emulation techniques and the abstract task scheduler model (with support for complex scheduling and preemption) presented in this work, allow modeling and analyzing multi-core and multi-task real-time system at the higher abstraction levels using SystemC/TLM. This avoids the usage of Instruction Set Simulators and binary SW, hence reducing the effort to perform that analysis and fully exploiting the simulation speed improvements provided by TLM.
Different VP use-cases have different requirements in terms of availability, simulation speed, and timing accuracy. A VP is mainly a composition of HW IP models and, therefore, those requirements are heavily influenced by the characteristics of the TLM models that are part of it. Ideally, models must be accurate enough, fast enough and simple enough to create in order to fit all VP use-cases. However reality shows that different requirements are achieved only by using different type of models. This is because TLM modeling is a multidimensional problem where the different dimensions (speed, timing accuracy and modeling effort) are orthogonal with each other. Typically, high speed models contain very little time information and accurate models are very slow. It is of course possible to create an optimized model that can be fast and still contain significant time information. However, such optimizations are most of the time model dependent and to a great extent
they can be considered an art, and mastering an art always requires big effort. This chapter proposes advanced modeling techniques that enable the creation of fast and reusable HW IP models with limited effort.
3.1 INTRODUCTION

Nowadays models that require a good trade-off between simulation speed and accuracy are still implemented by hand. This chapter proposes a modeling strategy to create speed optimal behavior models that can be progressively refined with more timing accuracy with limited effort. Techniques such as time annotation and dynamic clock generators are basic to keep the speed of models high. Concepts such as hierarchical modeling and dynamic layout are introduced as the foundations to enable model refinement for the behavior part. Structural composition is the key concept to reduce manual modeling effort. These concepts are explained further in Section 3.3.

The idea of structural composition is especially suitable to model the behavior part of the IP model (i.e. IP behavior core, see Section 1.5.1). The basic idea is to create models by composing together predefined building blocks that contain specific functions or features. These building blocks have to fit together in a “plug & play” fashion in order to be worthwhile. The rationale behind this approach is the typical “80-20 rule”, which applied to models express that 80% of the model can be reused and 20% will be new. Therefore, a library of optimized generic modeling features that can be used to quickly compose most of the model structure and functionality can reduce significant the modeling effort. As shown in Figure 38 this library can be an intermediate layer between the standard language libraries and the models. An example of generic modeling features is given in Section 3.4.

![Figure 38. Relationship between modeling libraries, models and VP](image-url)
3.2 RELATED WORK

Ideally the more effective way to reduce the effort to create a SystemC TLM model is by automatically generating it from another model representation. Two basic options are possible. In the first option, there is a RTL version of the IP block, typically written in a HDL language like VHDL or Verilog. A SystemC model can be extracted from the HDL code by using generation tools like [69] or [70]. In the second option, the IP block does not have a RTL counterpart, but there exist a functional reference model written in a software language like C or C++. In this case a SystemC model can be generated by using high level synthesis tools like [80], [81] or [82]. Unfortunately in both cases the resulting model is low-level resembling a RTL style with signal-level interfaces and clock sensitive processes, which makes it too slow to be used on most of the VP use-cases described in Section 1.4.5.

In [92] and [90] two approaches based on structural composition to create HW IP models are presented. In [92] the UNISIM simulation environment is introduced. UNISIM is a simulation environment build on top of SystemC and a library of APIs and features focused on modeling HW control logic. Although this work supports Transaction-Level communication its main benefit is in reducing the effort to model HW components at a detailed Cycle-Level. On the other hand, the main drawback of this solution is that deviates from evolving industry standards without adding significant benefits to the field. In [90] a library of performance features especially targeted to create processor models (e.g. pipelined execution units, caches, issue queues, etc) is shown. This library is part of the SLATE tool, which aims to provide early analysis of performance, power, physical and thermal characteristics of multi-core systems.

In [91] Cornet et al. introduces a technique to refine from un-timed to timed TLM models whereas keeping the functionality of the model untouched. This approach proposes to split the model in two parts: one part containing the functionality and a separated part containing the detailed timing information for both communication and computation. Although the approach seems to be useful to reduce modeling effort, it has inherent limitations since relies on a non-standard model-to-model synchronization mechanism. These limit the free exchange and integration of
models from different sources. This work is an extension to the Transaction Accurate Communication (TAC) modeling library from STMicroelectronics [66].

In [63] an approach to enable reusability for multiple abstraction levels by separating communication, behavior, and timing in IP models is presented. Each of these aspects is nicely supported by a set of well-defined interfaces and modeling objects part of the SCML library (SystemC Modeling Library) [62]. This library provides a set of building blocks and methodology guidelines to effectively create reusable peripheral models in short time and with good simulation speed. Re-using a model for different abstraction levels (i.e. VP use-cases) is only limited to replace the communication part of the model, leaving the behavior part unchanged. In the opinion of the author this is not always sufficient, especially for complex models where the granularity of the computation part can have a significant impact. The work presented in this Chapter extends the capabilities of the SCML approach by enabling a more structured behavior modeling and providing the means to refine the behavior part similarly as the communication part.

### 3.3 ADVANCED MODELING CONCEPTS

#### 3.3.1 Separation of concerns in the IP behavior core

Several orthogonal layers can be identified when creating an HW IP model. These layers have specific purpose within the model:

- **Behavior.** Basically, the idea is to model the internal behavior of the IP model in a more structured manner. Functions are encapsulated in individual blocks that communicate among each other via predefined ports and interfaces. This approach permits easily to model the data and control flow within the IP model and explicitly identify the synchronization points. Having a good understanding of the synchronization points within the model is vital to reduce the number of context switches and maximize speed. Specialized
features that allow modeling Finite State Machines at the TLM level and specialized ports that allow a seamless connection with the other layers are also part of the behavior layer.

- **Timing.** There are two ways to model timing information in a TLM model: time annotation or clock sensitivity. Time annotation leads typically to faster models and clock sensitive models are typically more accurate. In this work a solution based on advanced clock features that support time annotation and where clock sensitivity can be enabled/disabled dynamically within the model is proposed. This approach has two benefits: first, allow models to be refined with more accuracy without modifying its interfaces, and second allow models to switch their level of accuracy during simulation.

- **Storage and synchronization.** The goal of this layer is twofold: first, allow the communication and synchronization between the interfaces of the IP core model and the behavior layer and, second, allow the communication and synchronization among computing blocks within the behavior layer. The basic blocks composing this layer are implemented as SystemC channels that contain specific virtual interfaces (derived from `sc_interface`) and which can be accessed from a specialized port (i.e. a derived class from a `sc_port` using the specific interface).

- **Generic core interfaces.** The idea is to provide a set of generic interfaces that can be used in all IP models. These generic core interfaces are focused on two aspects: clock and memory-mapped bus protocols. The generic interfaces for memory-mapped busses do not model any specific standard protocol, but just enable the separation between the behavior part and the communication part. The protocol specific information has to be implemented in the separated IP interface adaptors.
3.3.2 Hierarchical modeling

A key concept of the proposed modeling strategy is the concept of hierarchical modeling. This concept fits perfectly with the layered approach presented before and proposes a more structured way to create the internals of the IP model. Basically behavior blocks are independent modules (deriving from the SystemC standard \textit{sc\_module}) and synchronization & storage blocks are channels that have to be connected to the behavior blocks via ports (deriving from the SystemC standard \textit{sc\_port}). The instantiated building blocks and their connections create what we name the internal model layout. This is shown in Figure 40.
Benefits of this approach are:

- **More structured code.** Models are composed of separated blocks that encapsulate pieces of functionality. This solution is scalable and makes the creation of complicated IP cleaner and easier to debug and maintain.

- **Reusability.** With this approach most of the model is created from predefined blocks and the modeler only have to define the IP specific functionality.

- **Simpler for HW designers.** Hierarchy modeling makes the model creation more intuitive from a HW designer point of view since there can be a one-to-one relationship between the blocks in the IP datasheet and the blocks in the model. Moreover predefined blocks hide SystemC and TLM specific construct with more intuitive features (e.g. Finite State Machines).
• *Refinement and configurability is improved.* This leads to the concept of dynamic layout explained in Section 3.3.3.

Hierarchical modeling could have also inherent drawbacks that are listed below:

• *Performance overhead due to extra indirections.* However, real complex examples showed a measured overhead lesser than 1% of the total simulation time.

• *Code overhead.* Due to the extra level of hierarchy, models contain more code describing the block instantiation and connections (netlisting). This also means that every behavior block has to declare own constructor, ports, members, etc. Fortunately most of the extra code is structural and can be easily generated by template-based code generation tools.

### 3.3.3 Dynamic layout

The introduction of hierarchy in the internals of the IP model brings out the possibility to, in a simple way, change/configure the layout of the model during its instantiation (construction and/or elaboration time). A change on the model layout can mean one or several of the following things:

• **Add/Remove functionality.** This means that new functionality can be added/removed by instantiating (or not) blocks during the construction of the IP model. This allows creating models that can behave as different IPs (of the same family) in a very reusable way, just by changing the internal layout.

• **Refine functionality.** Internal blocks of an IP can be replaced by different versions of the same functionality depending on the level of accuracy required. For example, let's assume an IP model of a video computing engine required for a use-case where simulation speed is more important than cycle accuracy. This model would require its behavior blocks computing
video processing operations in a very coarse way (e.g. granularity at frame level) with little time information. In case we want to reuse this model for use case where accuracy is important, the granularity of theirs operations as well as its timing information has to be more detailed (see Figure 41). Replacing those relevant behavior blocks with versions that bring extra accuracy enables to reuse the same model for different use-cases.

- **Increase the number of blocks of a same type.** This is especially useful to increase the parallelism of certain operations within a model. Typically those models that have a configurable number of interfaces can benefit of the simplicity of this approach.

- **Change the connections among blocks.** Obviously adding or removing (and potentially replacing blocks) implies to have new or different connections among the instantiated blocks to create the new model layout.

![Figure 41. Functionality refinement](image)

### 3.3.4 Timing modeling

Techniques to model timing information, whereas keeping the simulation speed high, are discussed in this section. The general idea is to move away from cycle-callable models evaluated every clock cycle, which are extremely expensive in terms of simulation speed. Instead fast cycle-count accurate models that combine time annotation techniques with controllable clock event generators that can be enabled / disabled dynamically are the way forward.
In general the following concepts are advocated when modeling timing in a SystemC/TLM model:

- **Use time annotation techniques instead of clock sensitive processes.** Time annotation allows reducing the number of times a model needs to synchronize with the SystemC kernel. With time annotation, delays parts of the model functionality are stored in `sc_time` member variables. However they are not immediately consumed by calling `sc_core::wait(sc_time)`, but can be accumulated for later consumption (e.g. when a mandatory synchronization point is reached).

- **Use controllable clock event generators in case clock sensitive processes cannot be avoided.** If clock sensitive processes cannot be avoided, at least use clock generator where the event generation can be dynamically switched on and off when the model does not need to be active.

- **Use SC_METHOD clock sensitive processes instead of SC_THREAD.** If clock sensitive processes have to be used SC_METHOD is more efficient than SC_THREAD since they do not produce a context switch. However, because of their state-less nature modeling with SC_METHOD increases significantly the effort to create the IP models.

When using **time annotation** the following aspects should also be taken into account:

- **Separate communication timing (at the interfaces) from the computation timing (among the internal blocks).** Communication time should be consumed in the specific interface adaptors (i.e. outside the IP behavior core boundary). Therefore the generic interfaces should have means to carry the required time information from the IP internals to the interface adaptors. Computation time is consumed in operations within the behavior blocks or among blocks through the synchronization & storage features.

- **Separate synchronization points from timing cost of operations.** Models that contain threads need to yield the control to the simulation kernel in order to
allow other threads to make progress. This holds even for untimed models. This yield call can be implemented as a wait for a delta cycle (e.g. `sc_core::wait(SC_ZERO_TIME)`) or can actually consume certain time that represents the latency of the operations (e.g. `sc_core::wait(sc_time)`). In order to keep the model simulation speed as high as possible synchronization points should be reduced to the minimum (without jeopardizing the model functionality) and consuming time should only occur in those synchronization points.

- **Use clock relative time annotations instead of absolute timing.** Absolute wait statements (`sc_core::wait(sc_time)`) constraint the model usage, since they are not sensitive to clock frequency changes. In order to create models that can change its timing when clock frequency changes (even dynamically), clock period information has to be retrieved from an external clock. For this purpose a special clock port is required since typical `sc_in<bool>` port will not work for timing annotation. The total time is calculated by, for instance, `sc_core::wait(number_of_cycles*clock_port.get_clock_period())`.

- **Delayed event notification.** Another technique that can be used very efficiently with time annotation is delayed event notification. Basically the idea is to fire events that in turn trigger functionality after certain time (`sc_event.notify(sc_time)`). This saves context switches and contributes to the overall speed of the simulations.

**Communication timing** has to be modeled in the (bus specific) IP interface adaptors according to the communication protocol and abstraction level selected. The IP behavior core must send generic write/read transactions with no-cost via the generic interfaces to the adaptors, where they have to take care of the timing from the moment a transaction is received until the end of the transaction. Since interaction with the behavior part can also influence the overall timing of the communication, it is advised to use non blocking interfaces that post the transaction from the IP core to the interface adaptor, together with synchronization mechanism that indicate the different phases of the communication protocol and/or the availability of the data. For example, SCML provides means for this modeling style.
with their post interface and the scml_array synchronization mechanism [62]. Similarly, TLM2.0 provides the nb_transport interface with a phase attribute for this purpose [19].

**Computation timing** represents the cost of executing certain behaviors/operations and should be part of the IP model itself. Two kind of generic delays can be identified in every HW IP model:

- **Roundtrip delay (begin input stimuli to end input stimuli).** This delay is measured from the moment a generic input interface of the IP (e.g. bus target) receives a stimuli (e.g. transaction) to the moment the IP core respond to it (e.g. return the interface call). This is shown in Figure 42. When using time annotation the round trip delay can be as simple as a static parameter annotated in the behavior or it can be a very complex dynamic calculation based on the IP internal state and configuration. In order to support round trip delays with time annotation the generic input interface should be able to send back the calculated delay when the transaction call returns (i.e. it will be handle in the interface adaptor). In case this is not possible the IP core should call explicitly `sc_core::wait(round_trip_delay)` before the transaction call returns.

- **Forward delay (begin input stimuli to begin output stimuli).** This delay is measured from the moment a generic input interface of the IP (e.g. bus target) receives a stimuli (e.g. transaction) to the moment an output stimuli is generated as a reaction in a generic output interface (e.g. initiate a transaction or generate an interrupt). Moreover the forward delay can be split in two delays (see Figure 42):
  - *Activation delay:* This delay is basically the time that takes to trigger certain behavior in the IP and represents more the internal synchronization or communication delays among behavior blocks. Delayed event notification is a perfect technique to model this type of delays when using time annotation.
  - *Processing delay:* This delay is basically the time that takes to compute a certain behavior and represent more the cost of the
operations within a block. Explicit `sc_core::wait` has to be used when consuming these delays.

![Diagram of computation delays](image)

**Figure 42. Computation delays**

### 3.4 GENERIC MODELING FEATURES

#### 3.4.1 Behavior

The features provided by the behavior layer are basically a set of specialized ports and functionality placeholders. That is, default functions are not provided in the library, but user defined functionality has to be encapsulated in these placeholders (a.k.a. computing blocks). Therefore, the IP model has to define its behavior as a group of computing blocks that interact with each other and with the other layers. This interaction is performed through specialized ports. More concretely the behavior layer is composed of three parts:

- *Specialized ports*
- *Basic computing blocks*
- *Finite State Machine blocks*
3.4.1.1 Specialized ports

As mentioned before, computing blocks communicate with each other via synchronization & storage channels. Specialized ports derive from the standard sc_port and provide access to a specific channel via its interface method calls. These specialized ports are grouped on three kinds:

- **Standard SystemC ports.** The ports in this category are basically: sc_fifo_in and sc_fifo_out. A complete explanation of these ports can be found in the SystemC LRM document [18].
- **Extended SCML ports.** This includes ports to extended SCML features [62]. See Section 3.4.2.
- **New specific ports.** This includes ports to new specialized channels. See Section 3.4.2.

3.4.1.2 Basic computing blocks

Computing blocks are available in three different flavors:

- **function_thread.** This component is a derived class from sc_module, containing a dynamic sc_spawn acting as a SC_THREAD. A function_thread contains also a virtual main() method that have to be filled with the user specific functionality. This main function is linked to the sc_spawn during elaboration time. A function_thread allows both dynamic and static sensitivity to be used. Static sensitivity has to be used via the set_sensitivity method. An example of a user defined function_thread is shown in Figure 43.

- **function_method.** This component is a derived class from sc_module, containing a dynamic sc_spawn acting as a SC_METHOD (using the spawn_method() function). A function_method contains also a virtual main() method that have to be filled with the user specific functionality. This main function is linked to the sc_spawn during elaboration time. A function_method allows static sensitivity to be used via the set_sensitivity method and dynamic sensitivity via next_trigger functions. An example of a user defined function_method is shown in Figure 44.
• **function_callback.** This component is also a derived class from `sc_module`. At difference of the other computing blocks, a `function_callback` does not create a `sc_spawns` during elaboration time nor has a `virtual main()` method to be completed by the user. A `function_callback` is a container for one or several user defined functions, which have to be registered as callbacks and executed from another component. Obviously the user defined functions should have the callback signature to allow the binding. An example of user defined `function_callback` is shown in Figure 45.

```cpp
class finishTransfer : public function_thread {
public:
    memory_port<unsigned int> m_regDmaEnable;
    memory_port<unsigned int> m_regStatus;
    event_notify_port m_assign_p_IRQ_event;
    pool_port<trans_req> m_trans_pool;
    value_port<bool> m_busy;
    sc_fifo_in<trans_req*> m_trans_req_fifo;

    // Constructor
    finishTransfer(sc_module_name name) : function_thread(name) {
    }

    void main() {
        while(1) {
            // Wait for write transaction
            m_trans_req_fifo.read(m_write_trans);
            // Write transaction is the last, so wait for completion
            wait(m_write_trans->end_event()); // dynamic sensitivity!!!

            m_regDmaEnable = 0;
            m_busy = false;
            m_regStatus = 1;
            m_assign_p_IRQ_event.notify();
            // release all used transactions to the pool
            m_trans_pool.release(m_write_trans);
            ...
        }
    }
}
```

**Figure 43.** Code example for a user defined `function_thread`
A `function_callback` is used in combination with the SCML callback signatures (more information about callbacks can be found in the SCML documentation [62]).
3.4.1.3 Finite State Machine blocks

The FSM blocks are derived from the basic computing blocks and simplify the modeling of finite state machines at the TLM level. At this level, states in the state machine are not bound to a clock cycle (i.e. a state does not need to execute within a clock cycle), but to synchronization events. Basically the state machine is fired every time an event, to which the state machine is sensitive, occurs. A FSM block is basically a container where users can add states. States are implemented as independent member functions of the FSM class. A state contains the specific functionality to perform and returns the state to execute in the next firing. The order between states and control branches has to be defined for the user. A FSM block only implements all necessary mechanism to register states, as well as to move from one state to another.

FSM blocks are available in two flavors:

- **fsm_method.** This component derives from the `function_method` and therefore allows its same event sensitivity to be used. State functions have to return the next state to execute in the next firing. The next state is fired either when an event from the static sensitivity list is notified or when the event/condition notified by `next_trigger` happens.

- **fsm_thread.** This component derives from the `function_thread` and therefore allows both dynamic and static sensitivity to be used within a state. States have to return the next state to execute. By default the next state is executed right after the previous state without any explicit synchronization between them. In a `fsm_thread` synchronization has to be made explicit. States can wait on one (or multiple) events coming from one the specialized ports `wait(sc_event)`, wait a certain time `wait(sc_time)` and/or do a `wait()` without arguments that resumes the execution when any of the event of the sensitivity list is fired.
Figure 46. Code example for a user defined `fsm_thread`
3.4.2 Storage and synchronization

The approach for this layer is to implement the basic blocks as SystemC channels that contain specific virtual interfaces (derived from `sc_interface`) and which can be accessed from a specialized port (i.e. a derived class from a `sc_port` using the specific interface). There are three different kinds of blocks than can be used in this layer:

- Standard SystemC channels
- Extended SCML features
- New specific channels

3.4.2.1 Standard SystemC channels

This includes channels such as `sc_fifo`. More information about these channels can be found in the SystemC LRM document [18].

3.4.2.2 Extended SCML features

These channels are derived class from the original SCML features. Basically each original SCML components is wrapped together with a specific virtual interface that allows the binding and access from the specialized ports. The SCML features wrapped in this way are: `scml_memory`, `scml_bitfield`, `scml_array`, `scml_pool`, and `scml_router`. These resulting new channels have exactly the same functionality and user methods as the original SCML ones. More information about the original SCML features can be found in [62]. As an example Figure 47, shows how a wrapper for the `scml_bitfield` is implemented.
3.4.2.3 New specific channels

These channels have been created to extend the storage and synchronization layer with newer capabilities. The specific channels in this category are:

- **event.** This channel basically wraps a standard `sc_event` and provides two virtual interfaces to notify or wait for the event from two different specialized ports (`event_notify_port` and `event_wait_port`).

- **value.** This channel wraps a shared variable (standard type or user defined) and provides a virtual interface to access its content from a specialized port (`value_port`).

```cpp
class bitfield : public scml_bitfield, virtual public bitfield_if {
public:

    // Constructors
    bitfield( const char * n,
              scml_bitfield_container_if & c,
              size_type offset,
              size_type size ) : scml_bitfield(n,c,offset,size) {
    }

    // Operators
    this_reference_type operator=( value_type v) {
        scml_bitfield::operator= (v); return *this;
    }

    // Interface methods
    value_type get() const {
        return scml_bitfield::get();
    }

    void put( value_type v) {
        scml_bitfield::put(v);
    }

    value_type read() const {
        return scml_bitfield::read();
    }

    void write( value_type v) {
        scml_bitfield::write(v);
    }
};
```

Figure 47. Wrapper for the `scml_bitfield`
- **fair_mutex.** This channel wraps a standard *sc_mutex* and adds a fair arbitration mechanism (FIFO) on top. A virtual interface is provided to lock/unlock the mutex from a specialized port (*fair_mutex_port*).

- **multififo.** This channel groups a configurable number of *sc_fifo* and provides two virtual interfaces similar to the standard *sc_fifo_in_if* and *sc_fifo_out_if* that have been extended with an index parameter to perform out-of-order accesses. A *multififo* can be accessed from the specialized ports (*multififo_in_port* and *multififo_out_port*). Thanks to the out-of-order nature of this channel, an input port can put values in a *sc_fifo* with index *i* while an output port can get values from a *sc_fifo* with index *j*.

- **arbiter.** This feature is a hierarchical channel that derives from *function_thread* and implements the arbiter interface. This feature contains also a configurable number of *arbiter_export* that connects to the arbiter interface. Each export provides access to the interface with a different identifier. *arbiter_port* can connect to the *arbiter* through the exports. A *arbiter* can be used to negotiate granted access to a shared resource according to a user-defined arbitration policy. It provides a non-blocking interface `void nb_request() / sc_event& acknowledge_event()` and a blocking interface `void b_handshake()`. Both can be used indistinctly. After a *arbiter* receives a request from any of the ports connected to it, it calls the user-defined `void delay()` method. This method defines how much time the arbiter shall wait before calling the arbitration function. The arbitration function `int arbitrate(std::vector<bool>& v)` must be implemented by the user, but it is called automatically by the *arbiter*. Figure 48 shows an example of a user-defined arbiter implementation.
The generic core interface layer aims to provide a group of predefined ports to model the interfaces of the IP core. These ports aim to be reusable for different modeling styles. The generic core interfaces are only focused on three aspects of the IP behavior core:

- **Clock**
- **Generic bus interfaces (i.e., protocol agnostic)**
- **Interrupt signals**

### 3.4.3.1 Clock interface

The clock interface is provided through the `clock_port` feature. A `clock_port` derives from the standard `sc_port` and supports the same hierarchical connectivity as `sc_port`. This is shown in Figure 49. At the top level a `clock_port` has to be connected to a `clock` channel or to one of the provided clock generators modules (`clock_generator` or `combined_clock_generator`). A `clock` channel contains the information about the clock period and unit, which is passed to it during construction. The same holds for the clock generator modules. At the bottom level a `clock_port` is used directly in the behavior blocks to obtain the clock information (for time annotation techniques) and/or to receive clock events to which the behavior can be sensitive to. More information about the usage of these channels and the general timing modeling approach can be found in Section 3.4.4.
3.4.3.2 Generic bus interfaces

As mentioned before the aim here is to provide a protocol independent interface that can be used for emulation memory-mapped bus communication. Protocol specific information is added on the next layer through specific adaptor components.

As shown in Figure 50, two generic interfaces are proposed:

- **scml_post_port**, for the Initiator/Master interfaces. This interface is used to post transaction requests from the IP core into the interface adaptors. Transactions are initiated for the behavioral blocks composing the IP core. These transactions provide also access to the data to transfer between the IP core and the adaptor by means of a `scml_array` reference. The adaptor and the IP behavior synchronize on the data availability through the methods and functionality provided by the `scml_array`. This channel fully relies on the
`scml_post_port` implementation provided by the SCML library. It is important to notice that the request channel can post multiple transactions in the adaptor before handling the data and/or the response information. That is, it has a non-blocking behavior. This allows modeling pipelining, multiple outstanding requests and out-of-order communication.

- **PVTarget_port**, for the Target/Slave interfaces. This interface is used to execute the requested transactions on the IP core. **PVTarget_port** has a blocking behavior, which means that the transactions have to be completed (including the reading or writing of data) when the call return. The **PVTarget_port** is available as part of the SCML library. This port can be connected directly to a `memory` (for register file modeling) within the `storage & synchronization` layer of the IP core.

![Generic Bus Interfaces](image)

**Figure 50. Generic bus interfaces**

### 3.4.3.3 Interrupt interface

This interface is meant to replace the usage of `sc_signal` in TLM models for modeling interrupt lines. Although the usage of `sc_signal` does not introduce a
significant overhead on simulation speed, their use can lead to wrong interactions with untimed TLM models. This is mainly because sc_signals rely on the evaluate/update delta mechanism of the SystemC kernel, which it is not required in TLM models.

Figure 51. Interrupt interface

The interrupt interface is composed of two ports:

- **Interrupt_port.** This port is used in the IP model as the output port interface that generates the interrupt. The interrupt can be initiated by any of the behavior blocks. Then the interrupt is propagated to all interrupt_export connected to it in a hierarchical way. A boolean value can be used (optionally) in the interface to indicate either the interrupt line carries a level high (true) or a level low (false).

- **Interrupt_export.** This port is used in the IP model to receive a interrupt notification from another module in the system. In order to link the reception of an interrupt to the behavior layer, the interrupt_export has to be connected to a event channel. interrupt_export and event can be connected directly through a bind method. A behavior block containing its own thread of execution has to be connected to the other side of the event channel and

```cpp
void notify(bool level = true)

interrupt_export(const char* name = 0,
int_active_types _type = ACTIVE_LEVEL_HIGH);
```
wait for the event to be fired. The event in the channel will be triggered when
the interrupt arrives, in case the level (high/low) of the interrupt notification
matches the active type configured in the interrupt_export. This connection
and the possible activation types are shown in Figure 51.

![Figure 52. Clock features example](image)

### 3.4.4 Clock features

The clock features architected in this work supports clock event generation as well
as time annotation modeling styles, which allow a natural refinement on the timing
accuracy without having to modify the IP interfaces. The features are:

- **clock_port.** A clock_port derives from the standard sc_port and support the
  same hierarchical connectivity as sc_port. This is shown in Figure 49. At the
top level a clock_port has to be connected to a clock channel or to one of the
provided clock generators modules. At the bottom level a clock_port is used directly in the behavior blocks to obtain the clock information (for time annotation techniques) and/or to receive clock events to which the behavior can be sensitive to. Unlike the standard sc_clock, in this case events are not coming from the clock but from the clock_port directly. This allows enabling/disabling the clock event generation dynamically during simulation for each behavior block individually, which can be very beneficial to improve simulation speed. By default clock event generation is disabled and clock_port can only be used to retrieve the clock information (period, unit) from the clock. A clock_port implements also functions to set a divider factor that divides the incoming clock period. Again this can be done separately for each behavior block that has a clock_port.

- clock_gate / gate_port. A clock_gate is used to enable or disable all the clock_port connected to it. This only applies to clock event generation and the clock period can still be retrieved from a top level clock. A clock_gate can be controlled from a behavior block through a gate_port.

- clock_divider / divider_port. A clock_divider is used to divide the clock period of a top level clock. The resulting clock period will be used for all clock_port connected to it. This applies to both the clock event generation and the clock period information retrieval. A clock_divider can be controlled from a behavior block through a divider_port.

- clock_counter / counter_port. A clock_counter is used to count the number of clock cycles that have elapsed during certain time. A clock_counter will take into account whether the clock path is enabled or disabled for its calculation. A clock_counter can be controlled from a behavior block through a counter_port.

- clock. A clock is a channel containing the information about the clock period and clock unit. A clock does not generate clock events, but it is only a container for the clock domain information.
• **clock_generator.** A *clock_generator* is a module deriving from *sc_module* that provides a *sc_export* implementing the clock interface. A *clock_generator* has to be constructed with the same information about the clock period and clock unit as *clock*.

• **combined_clock_generator.** A *combined_clock_generator* is also a module deriving from *sc_module* that provides a single *sc_export* where to connect clock ports. The main characteristic of this block is that allow connecting both *clock_port* and standard *sc_in<bool>* ports to it. A *combined_clock_generator* contains both a *clock* and a *sc_clock* sharing the same clock domain information. This feature allows mixing new TLM models with old-style models whereas sharing the same clock connection, and therefore simplifying the top level netlisting.

Figure 52 and Figure 53 show a complex example of how clock features can be used together.
class clocked_ip : public sc_module {

public :

PVTarget_port<unsigned int, unsigned int> p_regs;
clock_port clock_domain_1;
clock_port clock_domain_2;
...

// CLOCK OBJECTS
clock_counter counter;
clock_gate main_gate;
clock_divider main_divider;
clock_divider sub_divider_0;
clock_divider sub_divider_1;
clock_gate sub_gate_0;
clock_gate sub_gate_1;
...

clocked_ip(const sc_module_name name) : sc_module(name)
, p_regs("p_regs")
, clock_domain_1("clock_domain_1")
, clock_domain_2("clock_domain_2")
...

// Clock objects
, counter ("counter")
, main_gate("main_gate")
, main_divider("main_divider",1)
, sub_divider_0("subdiv_0",10)
, sub_divider_1("subdiv_1",20)
, sub_gate_0("subgate_0")
, sub_gate_1("subgate_1")
{
...

// clock binding
counter(clock_domain_2);
main_gate(clock_domain_1);
main_divider(main_gate);
sub_divider_0(main_divider);
sub_divider_1(main_divider);
sub_gate_0(sub_divider_0);
sub_gate_1(sub_divider_1);

Figure 53. Timing example code
3.5 CONCLUSIONS

In this chapter advanced modeling methods, such as *hierarchical modeling* and *dynamic layout*, are proposed to enable refining specific parts of the internal behavior of the model with more details, without changing the remaining parts. Obviously the parts that should be refined are only those which can contribute to adding timing accuracy in a model.

Another important contribution of this chapter is the concept of *structural composition* on TLM modeling, which aims to reduce the effort to create complex models. The foundation of this idea is to quickly create models by composing together predefined blocks (or features) in a "plug and play" fashion. In this work a set of generic modeling features are implemented to prove the concept.

Finally, this chapter also contributes modeling techniques and best practices to create speed optimal models. The general idea is to move away from cycle-callable models evaluated every clock cycle, which are extremely expensive in terms of simulation speed. Instead fast cycle-count accurate models that combine time annotation techniques with controllable clock event generators that can be enabled / disabled dynamically are proposed.
Modeling methods for protocol specific cycle accurate communication using TLM2.0

The typical issue with TLM techniques is the accuracy vs. simulation speed trade-off for the communication blocks. Cycle accuracy for these models (e.g. busses, bridges, memory controllers, etc) is critical in order to address the architecture exploration and performance verification use-cases, since performance are heavily influenced by them. However, most models that have a high degree of time accuracy (approaching RTL) are inherently slow due to the big amount of events generated during their execution. On the contrary, models that can run at high simulation speeds are often modeled at abstraction levels that make them
unsuitable for those use-cases. This chapter introduces a new methodology that enables the creation of fast and cycle accurate protocol specific bus-based communication models, based on the new TLM 2.0 standard from the Open SystemC Initiative.
4.1 INTRODUCTION

Most of the cycle-accurate models that are created today are still being modeled at
the register transfer level with clock sensitive processes and signal-level interfaces,
which dramatically reduces the simulation speed of the model [86]. The main cause
of this slow simulation speed is the fact that signal-level interfaces and clock-
sensitive processes cause a lot of context-switching to occur in the SystemC kernel
every clock-cycle, even when nothing is being transmitted. A benefit of TLM models
is that, typically, they are not sensitive to clock events and, therefore, can reach
much higher simulation speeds than RTL models. Although this often means that
the model will no longer be cycle-accurate.

Another concern regarding TLM is the lack of a standard for creating the models.
This has led to many different implementations that are generally not compatible
with each other [22][27][30]. Recently, OSCI has tackled this problem by proposing
TLM 2.0 as the interoperability standard [19]. The OSCI TLM 2.0 standard
addresses several of the shortcomings of the TLM 1.0 standard with respect to
model interoperability and simulation speed. The standard focuses on SoCs that are
based on memory-mapped buses and defines three coding styles: un-timed (UT),
loosely-timed (LT) and approximate-timed (AT). However, no coding-style for cycle-
accurate (CA) modeling is defined in the TLM 2.0 standard.

The ideal goal for model designers is to combine the accuracy of RTL models with
the speed-up that TLM models provide using the TLM 2.0 interoperable standard.
The methodology presented in this chapter contributes to that goal by:

- Demonstrating that the TLM 2.0 standard provides the necessary
  mechanisms to create protocol-specific Bus Cycle Accurate (BCA) models.
- Providing a structured way to create protocol-specific BCA TLM 2.0
  interfaces and transactors.
- Demonstrating that the obtained models are at least one order of magnitude
  faster than typical SystemC signal-level based RTL implementations, while
  retaining the cycle-accuracy.
4.2 RELATED WORK

Considerable work has been done to improve the speed of cycle-accurate RTL models. Clock suppression techniques have been used to gain more simulation speed by disabling certain portions of the system when they are not needed [86]. While clock suppression can result in a big improvement of the simulation speed over traditional RTL models, they are still much slower than TLM models. TLM have been successfully used in design tasks ranging from embedded SW development to functional verification. The requirements of the TLM model depend on the use-case for which it is needed. Cai and Gajski [22] present an overview of six models that vary both in the abstraction level of communication and computation from completely un-timed to fully cycle-accurate. The models that are created with our methodology have a close resemblance to the bus-functional model proposed by that work, but using standards like SystemC and TLM 2.0.

For the exploration of on-chip communication performance, models which have a high degree of timing accuracy are needed. In [27] Pasricha et al. present a transaction-based abstraction level called Cycle Count Accurate at Transaction Boundaries that is shown to have a simulation speed-up of 55% over traditional cycle-accurate models, although lack of support for pipelining and out-of-order transactions in these models limits the accuracy for real protocols such as AXI or OCP. 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 [30]. 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.

Some IP providers have developed their own cycle accurate TLM APIs. Examples are IBM’s CoreConnect TLM models [73], the OCP-IP SystemC channels [74] and the ARM RealView APIs [75]. The problem with these implementations is that they do not follow a single standard and are therefore not interoperable with each other. The work shown in this chapter contributes to extend the scope of the TLM2.0 standard by using its extension capabilities to create bus-cycle-accurate protocol-specific models.
4.3 METHODOLOGY FOR CA MODELING WITH TLM2.0

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 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 is one of the key aspects that enable the reusability of models. In this work we have chosen the generic interfaces provided by the SystemC Modeling Library (SCML) from CoWare [62], as explained in Section 3.4.3.2.

Figure 54 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 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.
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 55 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. 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 56 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 (preferred) 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.
The final step is the creation of the transactors that enable the connection of the TLM 2.0 BCA protocol interface to IP models with the generic 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. The generic modeling features and concepts explain in Chapter 3 are used to model the behavior (FSMs) of the transactors.

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 [70], the Open Core Protocol from OCP-IP [72] and the Device Transaction Level (DTL) from NXP Semiconductors [77]. 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.
4.3.1 Protocol analysis

4.3.1.1 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. 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 be easily merged with the other groups. Hence no separate group is needed for those signals.

Table 5 shows the four attribute groups that are identified for the DTL protocol.

<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>
4.3.1.2 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 57 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 58 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) 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.

![Timing points for the Command and Read groups of the DTL protocol](image)

**Figure 58: Timing points for the Command and Read groups of the DTL protocol**

### 4.3.1.3 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.

### 4.3.2 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.

#### 4.3.2.1 Protocol Specific TLM 2.0 Sockets

The TLM 2.0 non-blocking transport interface implements the `nb_transport_(fw/bw)` 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 is 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.

Table 6: DTL Transaction Phases

<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 tag_ack signal</td>
</tr>
</tbody>
</table>

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.

There were 8 timing points identified for the DTL protocol in the previous step. These can be directly mapped to 8 transaction phases.

Table 6 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 59 shows a code example of the payload extension for the write channel.

```cpp
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 59: Code for the DTL write channel payload extension

4.3.2.2 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
an 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 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 60 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, which means that synchronization between the FSMs of the different channels is needed.

Figure 60: DTL channels and phase flows
4.3.3 Transactor creation

The final step is the creation of the initiator and target transactors that are needed to connect the core IP models with the generic interfaces (SCML based) 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.

![Schematic of the initiator and target transactor templates](image)

**Figure 61: Schematic of the initiator and target transactor templates**

4.3.3.1 Transactor templates

Figure 61 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 `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.

### 4.3.3.2 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. 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 62 and Figure 63 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 Section 3.4.1.3). Synchronization between the state machines, threads and generic interfaces is realized with SystemC events and FIFOs. The `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.
Modeling methods for protocol specific cycle accurate communication using TLM2.0

Figure 62: TLM 2.0 DTL Initiator Transactor

Figure 63: TLM 2.0 DTL Target Transactor
The generic interface in the initiator transactor implements the SCML post interface (see Section 3.4.3.2) 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.

4.4 ACCURACY AND SPEED RESULTS

In order to test the TLM 2.0 cycle-accurate DTL interface and transactors a number of test scenarios are 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.

4.4.1 Cycle accuracy verification approach

To verify if the TLM 2.0 DTL transactors are actually operating at the cycle accurate level and following the DTL protocol, a verification testbench using a Specman [76] verification component (eVC) is set up. Traffic monitors are used to extract the individual DTL signals from the TLM 2.0 transactions. Since the Specman protocol checker operates at the signal level, cycle-accurate functional behavior is verified.

4.4.2 DTL point-to-point test

Figure 64 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 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 writes operation in the memory.

Figure 64: Testbench to test the DTL interface and transactors

![Diagram](image)

Figure 65 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.
Figure 65: Waveform of simulation with a write and a read transaction

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 test 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. Configuration 2 simulates processing delays by introducing a 2 cycle delay for every command and data phase. Speed results for these configurations are shown in Figure 66.
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 $t_{\text{sim}}$ 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).

4.5 CONCLUSIONS

A methodology to create protocol specific BCA interfaces and transactors using the TLM 2.0 standard is presented in this Chapter. This work demonstrates that: 1) the extension mechanisms of the TLM 2.0 standard can be used to create a protocol specific BCA interface and 2) the simulation speed of the resulting models is between one and two orders of magnitude better than signal-level RTL models, while retaining the same level of accuracy.

At the moment, the methodology presented in this work consists of quite a few steps that all have to be performed manually. In the future, parts of this methodology will likely be automated. For instance, the creation of the transactors, which is the most time-consuming task, could potentially be automated in a similar fashion as shown in [78].
This chapter presents the main conclusions of this PhD thesis and the future research areas that can extend and improve this work.
5.1 CONCLUSIONS

The ultimate goal of any design methodology is to enable designers to create more complex systems, in less time, and with more quality. As introduced in this work, conventional design and verification methods applied broadly in industry cannot cope any longer with the complexity of the SoC designs and their ever-shorter time-to-market. As complement to conventional solutions, ESL design methods and tools are being proposed to improve the productivity of the designers and to bridge the design and verification gaps. The success of ESL methods relies on the rapidly adoption from Semiconductor industry and EDA vendors of standards such as SystemC and TLM 2.0.

The main area where ESL solutions are being successfully applied on current design flows is Virtual Prototyping. A VP can be applied effectively to several activities (use-cases) during the design cycle, for instance “Early Architecture Exploration”, “Embedded SW development” or “HW/SW Performance verification”, each use-case having different requirements in terms of availability, accuracy, or simulation speed. Since a VP is primarily a composition of SystemC/TLM HW IP models, those requirements are heavily influenced by the characteristics of the models that are part of it. Ideally, models must be accurate enough, fast enough and easy to create in order to fit all VP use-cases. However reality shows that different requirements are achieved only by using different type of models (the right model for the right use-case). This is because TLM modeling is a multidimensional problem where the different dimensions (speed, timing accuracy and modeling effort) are orthogonal with each other.

Having to create and maintain a separated model for each use-case is drastically reducing the benefits of VP technology, due to elevated cost of creating and maintaining the models consistent with each other. Therefore, model reuse and refinement is a must for the success of ESL technology.
The primary contributions of this PhD thesis are innovative modeling methods, as well as supporting tools and libraries, which enable model reuse and refinement along different ESL abstraction levels. The modeling methods presented in this work rely completely on the IEEE 1666 SystemC and the new OSCI TLM 2.0 standards. More concretely the primary contributions can be separated on two refinement steps: *Functional to Architecture model refinement* and *Architecture model gradual refinement*.

- In the *functional to architecture model refinement* step new methods to model the application functionality separated from the architecture details are proposed. To create the functional models a SystemC implementation of the Task Transaction Level programming is described in this work. This acts as an enabler to carry out the next steps. The major contribution of this part are the innovative techniques that enable mapping the TTL based model (split on computation, communication and synchronization parts) on predefined (but configurable) high-level architecture elements representing the resources of the HW platform. The outcome of this mapping stage is the refined architecture model.

- Secondary contributions of the *functional to architecture model refinement* step are twofold:
  - First, enabling the separation between the application functionality and the architecture, as well as their posterior mapping, allow evolving the typical HW/SW co-design approach with more sophisticated methodologies that fit better with the requirements of the “Early Architecture Exploration” VP use-case. This is further supported by the CASSE framework implemented in this work, which simplifies the iterative process of setting up the HW platform model, map the application model, configure the overall system, simulate and analysis the obtained results.
  - Second, the host-code emulation techniques and the abstract task scheduler model (with support for complex scheduling and preemption) presented in this work, allow modeling and analyzing multi-core and multi-task real-time system at the higher abstraction
levels using SystemC/TLM. This avoids the usage of Instruction Set
Simulators and binary SW, hence reducing the effort to perform that
analysis and fully exploiting the simulation speed improvements
provided by TLM.

- In the *gradual architecture model refinement* step the separation of the HW
IP TLM model in two distinct parts, *behavior core* and *interface adaptors*,
(although not original from this work) is the key concept that supports the
methodology applied in this PhD thesis. More concrete contributions of this
work are:
  - Advanced modeling methods, such as *hierarchical modeling* and
  *dynamic layout*. These methods are proposed to enable refining
  specific parts of the internal behavior of the model (*behavior core*)
  with more details, without changing the remaining parts. Obviously
  the parts that should be refined are only those which can contribute
to adding timing accuracy in a model.
  - Another important contribution is the concept of *structural
  composition* applied on TLM modeling, with the aim of reducing the
effort to create complex models. The foundation of this idea is to
quickly create models by composing together predefined blocks (or
features) in a “plug and play” fashion. In this work a set of generic
modeling features are introduced to prove the concept.
  - Modeling techniques and best practices to create speed optimal
models are also summarized and presented in this work. Although
most of these techniques are not original from the author, there is not
a unique publication where they can be found together. A unique
contribution of this work is, however, the implementation of a clock
object that combines time annotation techniques with controllable
event generation (i.e. can be enabled / disabled dynamically during
simulation).
  - Finally, innovative modeling methods to create bus protocol specific
Cycle Accurate *interfaces adaptors* using the new TLM 2.0 standard
are a unique contribution in this work. Whereas the TLM 2.0 standard
describes and supports modeling at the Loosely-Timed and
Approximately-Timed levels using the GP protocol, a modeling style to capture Cycle-Accurate communication is not described or supported by the standard. The work presented on this PhD thesis demonstrates that: 1) the extension mechanisms of the TLM 2.0 standard can be used to create a protocol specific Bus Cycle-Accurate interface and 2) the simulation speed of the resulting models is two orders of magnitude better than signal-level RTL models, while retaining the same level of accuracy. As far as the author knows, this is the first work that tries to implement a Cycle-Accurate interfaces and adaptors using the TLM 2.0 standard. Certainly it will not be the last one and industry most likely will follow a similar approach for other proprietary bus protocols, such as AXI or OCP.

5.2 FUTURE WORK

There are two main open aspects that deserve further investigation as a continuation of this work:

- *Dynamic application reconfiguration support.* This aspect will be a continuation of the work done in the area of application / architecture separation and mapping. The rationale behind is given by the fact that today’s devices execute different combinations of applications (video, audio, radio, security, connectivity, etc) simultaneously, providing to the end user a vibrant dynamic multimedia experience. New methods and tools are necessary to model and analyze the dynamics of such systems, in order to study the impact on the overall performance early during the specification phase of a design. Questions like “can my device start downloading information from Internet using my 3G connection at the same time the user is watching a movie? How many applications can run simultaneously before the performances degrade?” are very difficult to estimate at early stages, which could lead to costly over dimensioned designs or unwanted feature
reduction that impact the viability of the design. When looking to application reconfiguration three types of dynamic are distinguished:

- **Execution**: This means, for instance, stopping an application that is running or starting an application that is not running yet during the simulations.
- **Topology**: Tasks / channels composing the application model may be added or removed, and/or the connections may be changed.
- **Mapping**: The tasks may be bound to different processors and channels may be bound to different memory areas in the HW architecture.

Modeling all these aspects encounter fundamental limitations on the SystemC language. Therefore new innovative techniques are required to overcome these limitations without breaking the compatibility with legacy and current standards.

- **Automate the generation of TLM architecture models.** Automatic generation of the structural code representing the internal behavior of the model could be generated from a user-friendly graphical tool based on a SysML [99] description. This will bring to the typical manual TLM modeling process, more advanced Model Driven Development [100] concepts and their inherent benefits. Moreover, automatic generation of the interface adaptors, which is the most time consuming task from the communication refinement part, could be potentially automated in a similar fashion as shown in [78].
Resumen en español
Resumen en español
6.1 Introducción

A medida que la capacidad de integración de las tecnologías basadas en silicio para fabricación de circuitos integrados continúa creciendo, los sistemas embebidos en un único chip continúan añadiendo más componentes y funcionalidad lo que incrementa significativamente su complejidad. Además la gran competitividad en el campo de la electrónica de consumo hace que los nuevos productos que quieran entrar en el mercado tienen que ser diseñados y producidos en cada vez menos tiempo. Esta gran competencia produce una enorme presión en los equipos de desarrollo de los grandes fabricantes de semiconductores ya que tienen que hacer frente a sistemas más complejos en menos tiempo. El objetivo final de cualquier metodología o herramienta de diseño electrónico es el permitir crear a los diseñadores sistemas en un único chip más complejos, en menos tiempo y con menos defectos. Los métodos convencionales de diseño y verificación que típicamente se aplican en la industria no pueden hacer frente por más tiempo la complejidad de los diseños y a los cada vez más cortos tiempo de entrada al mercado.

Los métodos de diseño a nivel de sistema (ESL) esta siendo propuestos en la industria de los semiconductores como un complementa a los soluciones convencionales, con el objetivo de mejorar la productividad de los diseñadores y aliviar las tareas de diseño y verificación de sistemas en un único chip. Una visión general de la tecnología ESL se puede leer en [84] y [85]. ESL se esta estableciendo rápidamente en la industria de los semiconductores gracias a los estándares SystemC y TLM2.0 (modelado a nivel de transacciones) que permiten elevar el nivel de abstracción de los diseños por encima del típico nivel a transferencia de registros (RTL). Más concretamente, ESL esta siendo aplicado en los flujos de diseño de forma exitosa por medio de prototipos virtuales (VP) descritos en SystemC y TLM. Un prototipo virtual es básicamente un modelo en software que describe la funcionalidad y temporización de la arquitectura hardware del sistema.
6.1.1 Diseño a nivel de sistema

Para poder hacer frente a la complejidad de los SoC actuales y para superar las limitaciones de los métodos de diseño convencionales, métodos de diseño y verificación a nivel de sistema se han propuesto por la comunidad de investigación durante años. A grandes rasgos, métodos SLD se han enfocado en tres aspectos principales: mover diseñadores a trabajar con niveles de abstracción por encima del nivel de transferencia entre registros (RTL), separar los varios aspectos de un diseño para permitir una exploración más efectiva de soluciones alternativas, y permitir el refinamiento progresivo de los sistemas desde descripciones abstractas hasta la implementación. Más en detalle:

- **Abstracción.** Así como el nivel de abstracción de los diseñadores se movió desde diseño con células estándares usando técnicas full-custom a lenguajes de descripción hardware usando RTL en los años 90, nuevos métodos de diseño proponen el elevar el nivel de abstracción del diseñador desde RTL hasta nivel de sistema. A nivel de sistema los elementos básicos tienen una mayor granularidad (e.g. bloques IP o subsistemas completos) y están descritos usando lenguajes de modelado de alto nivel.

- **Separación.** Separando los varios aspectos de un diseño más allá de hardware y software es un aspecto clave en métodos SLD para poder superar las limitaciones de los métodos de diseño convencionales. Los conceptos de ortogonalización de aspectos de diseño [10], el diseño basado en interfaces [11] y la estrategia Y-chart [12] han puesto la base para proporcionar dicha separación en diseño SoC.

- **Refinamiento.** El refinamiento progresivo de un diseño desde una descripción abstracta hasta un modelo de implementación es una estrategia clara para reducir la complejidad de los diseños [13]. Aunque dicho refinamiento es puede hacerse de forma manual, el beneficio real está cuando el proceso se puede automatizar por medio de herramientas. Ejemplos son las herramientas de síntesis de alto nivel que transforman
automáticamente una descripción C a una implementación RTL [83] o las herramientas de generación de código que crean código C o C++ desde descripciones UML.

6.1.2 Estándares de la Open SystemC Initiative

El uso de estándares es la diferencia clave entre el diseño a nivel de sistema académico y el concepto ESL adoptado por la industria. Se ha previsto que las herramientas ESL conducirán el mercado de las EDA en los próximos años, y que las herramientas de próxima generación ESL estarán soportadas por la continua adaptación de SystemC y TLM tanto en la industria y academia. La Open SystemC Initiative (OSCI) controlada por un grupo de las trece mayores compañías en EDA y la industria electrónica, promociona el desarrollo y la estandarización de ambos SystemC y TLM.

El estándar IEEE 1666 SystemC

SystemC es un lenguaje de modelado que esta pensado para permitir el diseño a nivel de sistema e intercambio de IP a múltiple niveles de abstracción, para sistemas que contienen componentes hardware y software. SystemC es un lenguaje de programación basado en C++ y extiende sus capacidades para permitir descripciones hardware. Estas extensiones se consiguen por medio de una librería de clases que proporcionan nuevos mecanismos para modelar elementos hardware, paralelismo, tiempo y comportamiento reactivo. SystemC también proporciona un núcleo de simulación que permiten simular una especificación ejecutable del diseño.

El estándar OSCI TLM 2.0

En TLM (modelado a nivel de transacciones) los detalles de comunicación entre los componentes están separados de los detalles de su implementación. Comunicación se modela por medio de canales y las peticiones de transacciones ocurren por
medio de llamadas a funciones de interfaz en dichos canales. El objetivo principal de TLM es incrementar la velocidad de simulación, mientras se mantiene suficiente precisión funcional y temporal para permitir el explorar y validar diferentes alternativas a los más altos niveles de abstracción. Dicho incremento en la velocidad se consigue al abstraer el número de eventos y la cantidad de información que se tiene que procesar durante la simulación al mínimo requerido. Además de este incremento en velocidad, TLM reduce la cantidad de detalles que el diseñador tiene que manejar, simplificando por lo tanto el modelado. TLM 2.0 proponen un estandarizado conjunto de llamadas a métodos de interfaz (IMC) para crear modelados a nivel de transacción, un paquete de datos genérico estándar (GP) para modelar comunicación a nivel de buses y mecanismo para extender el paquete de datos estándar con información específica [20].

6.1.3 Niveles de abstracción con SystemC

Simulaciones a nivel RTL limitan el tamaño y la complejidad de los sistemas debido a la enorme cantidad de detalles que el diseñador tiene que manejar y las tremendamente bajas prestaciones (en el orden de miles de ciclos por segundo). Varios niveles de abstracción son necesarios para representar los SoC al nivel de detalle requerido para las diferentes actividades que existen en el ciclo de diseño (e.g. si el nivel de detalle es suficientemente detallado para exploración de la arquitectura HW, entonces la velocidad de simulación es demasiada baja para análisis algorítmico). Los tres niveles de abstracción que pueden ser modelados con SystemC son:

- **Nivel funcional (FL).** Este nivel de abstracción solo representa bloques funcionales que componente los algoritmos o aplicaciones. Comunicación entre bloques funcionales se realiza a nivel de mensajes. El nivel de abstracción funcional también se conoce como nivel algorítmico o de procesos comunicantes. A día de hoy no existe un conjunto estándar de interfaces para crear modelos SystemC a este nivel.
**Nivel de arquitectura (AL).** El nivel de arquitectura tiene como objetivo el describir tanto la funcionalidad como la estructura de los bloques IP en una plataforma HW. Los modelos tienen que ser completamente correctos en su funcionalidad, pero los detalles microarquitecturales no tienen por qué tenerse en cuenta a este nivel. Comunicación entre los bloques IP se realiza a nivel de transacciones (TL), donde las interfaces representan las interfaces reales de los bloques IP.

**Nivel de implementación (IL).** Este nivel de abstracción tiene como objetivo el capturar los detalles de implementación de las interfaces y los bloques internos de los bloques IP. Comunicación entre bloques IP se realiza a nivel de señales/pines.

### 6.1.4 Estilos de modelado a nivel de transacción con OSCI TLM 2.0

Modelos TLM pertenece al nivel de abstracción AL. Modelos TLM se puede categorizar como modelos donde la comunicación entre los bloques se abstrae del estilo RTL a nivel de pines, a usar llamadas a función. Sin embargo el término TLM no implica ningún nivel particular de granularidad con respecto a la abstracción de la información temporal, estructural o del comportamiento, y por lo tanto no hay una manera única de crear modelos TLM. Para proporcionar una solución a este problema OSCI ha propuesto un grupo de estilos de modelado para crear modelos TLM. Estos estilos se diferencias entre ellos principalmente por la granularidad en la temporización de la comunicación. Los diferentes estilos de modelado y las definiciones propuestas por OSCI se describen a continuación:

- **Sin tiempo (UT).** Este es un estilo de modelado donde no hay una mención explícita de tiempo o ciclos, pero donde se incluye el paralelismo y la secuencia de operaciones.

- **Temporización suelta (LT).** Este es un estilo de modelado donde es posible el establecer una correspondencia entre los estado del modelo TLM
y los estado observable de un modelo RTL de referencia, cuando los momentos temporales se muestrean en los límites de una transacción.

- **Temporización aproximada (AT).** Este es un estilo de modelado para el cual existe una correspondencia entre los estados del modelo y los estados de un modelo RTL de referencia, donde se conserva los estados de transición pero no su temporización exacta. El grado de precisión temporal es indefinido.

- **Precisión a nivel de ciclo (CA).** Este es un estilo de modelado donde es posible predecir los estados del modelo en cualquier ciclo en los límites externos del modelo y por lo tanto se puede encontrar una correspondencia directa con los estados de un modelo RTL de referencia.

### 6.1.5 Prototipos Virtuales

Un prototipo virtual (VP) es un modelo software escrito típicamente en SystemC a nivel de transacciones que emula la funcionalidad de la plataforma HW y hasta cierto punto su temporización (dependiendo del estilo de modelado). VP son modelados a nivel de transacciones para así poder conseguir la velocidad de simulación requerida para ejecutar modelos de sistemas complejos. Un prototipo virtual tiene varios beneficios cuando se comparan con otras soluciones de prototipado, como por ejemplo emuladores, FPGA o simuladores RTL. Un prototipo virtual puede ser disponible antes en el ciclo de diseño. Esto es gracias a que los modelos SystemC TLM están creados a niveles de abstracción mucho mayor que RTL, lo cual implica menos detalles de implementación y por lo tanto puede ser creados con menos esfuerzo. Además puesto que estos modelos contienen menos detalles, cambios pueden hacerse más rápido y nuevas instancias de un diseño se crean en menos tiempo que con otras soluciones de prototipado. Esto hace de un VP una solución más flexible. Finalmente siendo una solución SW un prototipo virtual ofrece una mejor visibilidad de lo que ocurre en el sistema cuando se
compara con emuladores o FPGAs. Además puesto que el nivel de detalle es mayor con un VP es más fácil el señalar los errores que con simulaciones RTL.

Con un VP el HW y el SW se pueden integrar y verificar juntos mucho antes que con otras soluciones. Requisitos funcionales y no-funcionales pueden ser validados, y errores y/o cuellos de botella se pueden detectar y arreglar siguiendo mucho más cortas iteraciones. Esta estrategia ahorra una considerable cantidad de tiempo, reduce el riesgo de rediseño y por lo tanto de fallar la ventana de mercado y los plazos de un diseño. Además un VP sirve como una representación ejecutable de la plataforma HW, la cual pude complementar la ambigua especificación en papel. Esto reduce el riesgo de introducir errores durante la fase de implementación debido a interpretaciones erróneas de la especificación. Un VP sirve como puente entre la fase de especificación y el resto del flujo de diseño, puesto que puede ser utilizada como una referencia para el desarrollo del SW o el HW. Nuevos bloques IP se pueden desarrollar a nivel RTL usando el VP como un banco de pruebas a nivel de sistema. Esto reduce considerablemente el número de estímulos y escenarios que tienen que ser creados para testear el bloque IP y por lo tanto reduce el tiempo y el esfuerzo requerido para la verificación RTL. Además el VP puede contener suficientes detalles de la plataforma HW para permitir el desarrollo del SW que depende del HW en paralelo con el desarrollo de RTL.
6.2 Contribuciones de la tesis doctoral

6.2.1 Planteamiento del problema

Los prototipos virtuales se pueden aplicar a diversas tareas durante el ciclo de diseño. Los típicos usos de un prototipo virtual están en el desarrollo de software embebido antes de que un prototipo físico del hardware esté disponible o la exploración de diversas alternativas de implementación durante la fase de especificación del chip, entre otros. El problema aparece cuando tenemos en cuenta que estos diferentes usos típicos de un prototipo virtual tienen diferentes requisitos en términos de velocidad de simulación, precisión en los resultados, etc., que hace muy difícil elegir un único estilo para modelar un prototipo virtual de forma que cumpla todos los requisitos para todos los casos. Además el crear diferentes modelos para diferentes casos, teniendo que mantener los diferentes modelos consistentes, añade una gran complejidad y por lo tanto un elevado coste al desarrollo de dichos prototipos virtuales que reduce de forma drástica los beneficios de las metodologías a nivel de sistema. Es por lo tanto obvio que métodos y técnicas que permitan reutilizar y refinar modelos para diferentes casos utilizando prototipos virtuales es clave para el éxito de las tecnologías ESL.

Como se describió en las secciones anteriores un modelo se puede representar en tres niveles de abstracción distintos: nivel funcional, nivel arquitectural y nivel de implementación. Además dentro del nivel arquitectural existen varios estilos de modelado posible con TLM (LT, AT, CA), cada uno encajando mejor con un uso específico del VP. Para crear un camino sin fisuras que permite el refinar los modelos a lo largo de los diferentes niveles de abstracción y estilos de modelado, los pasos requerido son los siguientes:
1. Refinar de un modelo a nivel funcional a un nivel arquitectural.

2. Refinar un modelo arquitectural a través de los distintos estilos de modelado. Esto consiste de los siguientes pasos:
   - Refinar un modelo LT a AT.
   - Refinar un modelo LT/AT a CA

3. Refinar hasta un modelo de implementación desde cualquiera de los niveles superiores.

**Refinamiento de nivel funcional a nivel arquitectural**

Tipicamente la funcionalidad de las aplicaciones se refina en modelos arquitecturales por medio de SW embebido que se ejecuta en simuladores de conjunto de instrucciones (ISS) o bloques IP HW específicos que tienen que ser integrados en la arquitectura del sistema. Creando SW embebido y modelos específicos HW para un nuevo diseño requiere un esfuerzo considerable que solo tiene sentido si la partición HW/SW se conoce de antemano. Por la tanto, debido al extremo acoplamiento de la funcionalidad y arquitectura les falta flexibilidad con respecto a la capacidades de exploración. Tener la capacidad de analizar y explorar las diferentes opciones es muy importante al principio del proceso de diseño, especialmente para aquellos diseños donde la mayor parte de las decisiones de mapeado y arquitecturales todavía se tienen que tomar. Por dicha razón, métodos de modelado que permiten separar la funcionalidad de la aplicación de la descripción de la arquitectura, y su mapeado posterior, son vitales para mejorar el proceso de tomas de decisiones y la exploración temprana de diferentes escenarios.

**Refinamiento de modelos arquitecturales (LT, AT, CA)**

Debido a sus diferentes requisitos, los modelos creados para un específico tipo de VP no pueden ser reutilizados para otro tipo. Por ejemplo, los modelos TLM creado
para un caso de “verificación de prestaciones” son demasiado lentos para ser usados en un caso para “desarrollo y optimización de SW”. Asimismo los modelos creados para un caso de “verificación funcional” no están disponibles suficientemente pronto (y no tienen la flexibilidad suficiente) para ser usados en un caso de “exploración temprana de la arquitectura”. El concepto clave para refinar un modelo arquitectural a través de los distintos estilos de modelado y, por lo tanto, el permitir su reutilización para diferentes usos, es el separar el comportamiento del IP (computación) de los detalles de sus interfaces (comunicación). Como resultado de esta separación el modelo resultante se compone de dos partes separadas conectadas por medio de una interfaz genérica. Estas dos partes son:

- **El núcleo de comportamiento del IP**, que representa la parte computacional y contiene las principales funciones/operaciones que el bloque IP tiene que realizar.

- **Los adaptadores de interfaz del IP**, que representan la parte de la comunicación y contienen los detalles del protocolo que el IP tiene que ejecutar.

La interfaz genérica que conecta juntos el núcleo de comportamiento del IP con los adaptadores de interfaz del IP puede ser considerada como un artefacto de modelado. Es un elemento cuya presencia solo se justifica para permitir esta separación. Dicha interfaz genérica tiene que ser abstracta, es decir no deber añadir ningunos detalles de implementación, y tiene que ser neutral, es decir no debe influir ni la funcionalidad ni la temporización de la dos partes separadas.

Las partes de computación y comunicación tienen diferentes costes asociados a ellas. Refinar cada una de estas dos partes separadamente con más información temporal es la manera de obtener más precisión en los modelos con un esfuerzo limitado.
El refinamiento en la comunicación se puede conseguir por medio de remplazar los adaptadores de interfaz del IP con versiones que contienen más información temporal. Creando adaptadores con el estilo de modelado LT para un cierto protocolo sería el primer paso en el proceso de refinamiento. El modelado LT es muy simple y suficiente información puede ser encontrada en el manual TLM 2.0 [20]. El siguiente paso sería refinar a un modelo AT. Otra vez este paso es directo siguiendo las reglas de TM 2.0. Incluso sería posible el tener un único adaptador LT/AT ya que la interfaz en términos del paquete de datos del protocolo y las fases temporales es la misma. El siguiente paso de refinamiento sería hacia un estilo CA, deseablemente usando el estándar TLM 2.0. Sin embargo este estilo no se define en el estándar lo cual rompe la estrategia de refinamiento propuesto en estas páginas.

El refinamiento en la computación se puede conseguir por medio de reemplazar los bloques internos descritos a un nivel algorítmico por bloques funcionales modelados a una granularidad menor, es decir, bloques que representen un mayor número de detalles arquitecturales. A diferencia del refinamiento en la comunicación, en este caso no existe una definición estándar o estilos especiales de modelado que puedan ayudar a implementar los pasos de refinamiento en el núcleo de comportamiento del IP.

### 6.2.2 Contribuciones

Esta tesis doctoral tiene las siguientes contribuciones:

- Métodos de modelado que permiten la separación entre modelos funcionales y arquitecturales, permitiendo su posterior mapeado usando SystemC y TLM. Estos métodos permiten evolucionar la típica estrategia de co-diseño HW/SW con mas sofisticadas metodologías que encajan
mejor con los requisitos del uso típico para la exploración temprano de arquitecturas usando prototipos virtuales.

- Métodos de modelado que permiten la separación entre computación y comunicación en un modelo arquitectural usando SystemC y TLM, axial como su posterior refinamiento a través de los distintos estilos de modelado. Otras contribuciones de estos métodos son:
  - Proporcionar métodos estructurados innovadores para modelar el comportamiento/computación y el refinamiento con TLM. Además dichos métodos reducen el esfuerzo requerido para crear modelos TLM óptimos en velocidad.
  - Proporcionar métodos innovadores para modelar interfaces de comunicación a nivel de ciclo y adaptadores para específicos protocolos usando TLM 2.0.
6.2.3 Estructura de la tesis doctoral

Este resumen en español de la tesis doctoral está estructurada de la siguiente forma:

La sección 1.1 es esta introducción.

La sección 1.2 describe los métodos de modelado que permiten la separación y mapeado de los modelos funcionales y arquitecturales. Esta sección también cubre los requisitos y técnicas necesarias para dar soporte a la exploración temprano de arquitecturas usando VP y presenta una herramienta llamada CASSE que implementa dichos requisitos.

La sección 1.3 introduce los conceptos avanzados de modelado para crear modelos óptimos TLM que pueden ser refinados para diferentes usos con VP. Esta sección se enfoca en las técnicas que permiten modelar y refin r la parte computacional (el núcleo del comportamiento del IP) con poco esfuerzo.

La sección 1.4 presenta los métodos de modelado para crear interfaces y adaptadores de comunicación a nivel de ciclo para protocolos de bus específicos usando el estándar TLM 2.0.

La sección 1.5 Discute las principales conclusiones de este trabajo y las futuras líneas de investigación.
6.3 Métodos de modelado para la exploración temprana de la arquitectura

6.3.1 Introducción

Los métodos de modelado propuestos en esta sección tienen como objetivo el ayudar a los arquitectos de sistema a explorar y analizar modelos de las aplicaciones ejecutándose en modelos de la plataforma HW que están disponibles antes en el ciclo de diseño. Lo requisitos principales a considerar son los siguientes:

- **Modelado y mapeado de aplicaciones.** Durante la fase de especificación es bastante improbable que la aplicación esté disponible en su estado final, es decir SW embebido y/o implementación HW. Por lo tanto, otras técnicas deben de ser aplicadas para permitir modelar los requisitos de la aplicación (y deseablemente su funcionalidad) de forma más abstracta. Además, para permitir una exploración extensiva de escenarios, la separación de los aspectos de diseño tiene que ser utilizada. Esto significa que el comportamiento, las interfaces y el coste temporal de las aplicaciones tienen que modelarse de forma ortogonal y mezclarse una vez la aplicación se mapea en la arquitectura deseada.

- **Una librería de altamente configurable modelos arquitecturales genéricos.** Durante la fase de especificación diferentes arquitecturas HW con diferentes propiedades necesitan ser exploradas. Para tener prototipos virtuales disponibles antes durante la fase de especificación, modelos IP tienen que ser disponibles rápidamente y por lo tanto el esfuerzo para crearlos y ensamblarlos tiene que ser mínimo. Como una forma de solucionar esta limitación una librería de altamente configurable de modelos genéricos que emulan la funcionalidad común y la temporización de varios tipos de IP se tiene que proporcionar. Tales modelos IP genéricos pueden ser usados cuando el modelo para el IP específico no está disponible todavía. Además estos modelos arquitecturales genéricos tienen que soportar/permiten el mapeado directo de los modelos de las aplicaciones.
Métodos de modelado para la exploración temprana de la arquitectura

- **Análisis avanzado de las prestaciones.** Durante las simulaciones, toneladas de diferente datos de prestaciones tienen que ser interpretados y analizados antes de decidir si el sistema cumple o no con sus requisitos. Reducir esta gran cantidad de información para rápidamente identificar cuellos de botella en las prestaciones y posibles optimizaciones no es una tarea obvia en sistemas multiprocesador donde multitud de tareas se están ejecutando juntas. Por lo tanto, nuevas técnicas de análisis de prestaciones deben permitir el identificar puntos calientes y cuellos de botella en un diseño complejo y fácilmente correlar la información obtenida con el modelo de la aplicación. Esto permitiría aislar y optimizar los diseños de forma más sencilla y rápida.

- **Iteraciones rápidas y cortas.** Para permitir la exploración un VP necesita ser creado, configurado, simulado y los resultados obtenidos analizados en poco tiempo. Entre más pequeñas son las iteraciones, mayor es el número de diferentes escenarios que pueden ser explorados. Mas específicamente, los requisitos son:
  - Pequeño esfuerzo para cambiar/configurar el VP. Se requieren herramientas agradables al usuario y capacidades de automatización para permitir las modificaciones rápidas en la configuración del modelo VP.
  - Simulaciones rápidas. Al menos un factor de mejora x1000 comparando con simulaciones RTL se requieren para realizar simulaciones normales durante la fase de exploración. Esto nos lleva a velocidades de simulación en el rango de MHz.
  - Precisión relativa. Aunque precisión absoluta cuando se compara con la referencia RTL es siempre bienvenida, la mayor parte de la veces el nivel de precisión se puede reducir con el objetivo de reducir el esfuerzo de modelado y mejorar la velocidad de simulación. específicamente para la actividad de exploración el concepto de precisión relativa o fidelidad se introduce, de forma que el VP se pueda utilizar para evaluar, si la arquitectura A es mejor o peor que la arquitectura B para ciertos aspectos.
En el resto de esta sección se discute el estado del arte en herramientas y métodos que tienen como objetivo solventar los problemas mencionados anteriormente y se introduce la herramienta CASSE desarrollada como parte de esta tesis doctoral. CASSE es un entorno de simulación a nivel de sistema basada en SystemC que tiene como objetivo facilitar y acelerar el modelado y análisis de complejos SoC durante la fase de especificación.

6.3.2 Trabajo relacionado

Desde el punto de vista metodológico las herramientas actuales ESL, como por ejemplo [52], [53] o [54], siguen una estrategia de diseño basada en componentes, separan la computación de la comunicación en los modelos implementados, y son capaces de refinarlos desde descripciones abstractas hasta descripciones mas precisas. Sin embargo, estas herramientas no pueden ser aplicadas antes de que se decida la partición de la funcionalidad de la aplicación en módulos HW y SW. En este respecto, la funcionalidad de la aplicación se añade a los modelos arquitecturales por medio de SW embebido que se ejecuta en un simulador de juego de instrucciones (ISS) o un modelo ejecutable a nivel de ciclo (CCM) del modelo del procesador, o módulos HW específicos que tienen que ser integrados en la arquitectura del sistema. Creando SW embebido y modelos HW específicos para un nuevo sistema todavía requiere un esfuerzo considerable que solo tiene sentido si la correcta partición entre HW y SW se conoce de antemano. Por lo tanto y debido al apretado acople entre la funcionalidad y la arquitectura, se pierde flexibilidad con respecto a las posibilidades de exploración del espacio de diseño (DSE). DSE es muy importante al principio de ciclo de desarrollo, especialmente para aquellos diseños donde la mayor parte de las decisiones arquitecturales y de mapeado todavía se tienen que tomar. Por dicha razón, herramientas y métodos que eleven el nivel de abstracción por encima de HW/SW y permiten separar la aplicación de las descripciones arquitecturales, axial como su mapeado y análisis, son vitales para mejorar el proceso de toma de decisiones y la exploración temprana del espacio de diseño.
Además, a las herramientas actuales ESL les faltan características importantes para modelar tareas y planificación SW, las cuales son requeridas para modelar sistemas operativos de tiempo real (RTOS). Esto es principalmente debido a las limitaciones del lenguaje SystemC. El modelado y análisis de los parámetros de un RTOS es extremadamente importante ya que su incorrecta configuración y uso puede influir negativamente las prestaciones del sistema global. Actualmente, estas herramientas hacen frente con la configuración de RTOS y el desarrollo de SW a bajo nivel por medio de usar modelos ISS de lo procesadores embebidos, lo cual reduce las ganancias en términos de esfuerzo de modelado y velocidad de simulación obtenida al usar modelos de alto nivel. Por lo tanto, se requieren nuevas técnicas de modelado y métodos que permiten el modelado de tareas SW y RTOS a altos niveles de abstracción.

El esquema Y-chart [33] es un ejemplo típico de una metodología que aplica la separación de aspectos de diseño. El esquema Y-chart simplifica el proceso DSE por medio de modelar independientemente la funcionalidad y la arquitectura, y posteriormente combinándolos juntos en una fase de mapeado separada. CASSE sigue un a metodología Y-chart donde la aplicación se separa de la arquitectura, pero también donde la comunicación se separa de la computación por medio de una interfaz a nivel de tareas [34]. Entornos similares son Spade [13], Sesame [32] y VCC [48]. Spade y Sesame empieza con simulaciones funcionales de la aplicación que se describen en la forma de una red de procesos de Kahn [49]. Sin embargo, estos entornos aplican simulaciones arquitecturales conducidas por trazas donde los modelos arquitecturales (anotados con figuras de prestaciones y tiempo) se alimentan con trazas obtenidas durante simulaciones funcionales. VCC, aunque no disponible comercialmente, fue uno de los primeros ejemplos comerciales de un entorno de diseño a nivel de sistema. VCC se basa en el entorno Polis [36].

Entornos más novedosos son Metropolis y Daedalus. En [37] se presenta el entorno Metropolis. Metropolis proporciona un meta-modelo de computación que ofrece mecanismos sintácticos y semánticas para soportar la captura y análisis de la funcionalidad, axial como la descripción de la arquitectura y el mapeado de la
funcionalidad en los elementos arquitecturales. Además de la separación función/arquitectura, Metropolis también proporciona la separación entre las capacidades de un modelo arquitectural y el coste que conlleva su implementación. El entorno Daedalus [38] extiende el trabajo de Sesame [32] por medio de añadir la paralelización automática de la aplicación desde descripciones secuenciales en C/C++ [50] y síntesis automática e integración RTL en tecnología FPGA. Aunque bastante valiosos desde el punto de vista innovador y metodológico, estos entornos no están basados en SystemC. Esto es una importante limitación con respecto a su integración e interoperabilidad con flujos de diseño ESL adoptados en la industria.

Entornos basados en SystemC que aplican técnicas similares que CASSE se encuentran en [51], [39] y [40].

CoFluent [51] es una herramienta comercial que aplica separación entre función y arquitectura y mapeado. De forma similar como en CASSE, esta herramienta ayuda el la exploración temprana del espacio de diseño. Asimismo esta herramienta proporciona modelado abstracto de RTOS incluyendo propiedades importantes tales como arbitraje, preemtividad, etc. Esto se discute en detalle en [42]. CoFluent está basado en la metodología MCSE y su modelo funcional describe al sistema como un conjunto de tareas y las relaciones entre ellas. A diferencia de este trabajo que usa un modelo de programación paralelo independiente de la arquitectura, CoFluent modela la comunicación y sincronización entre las tareas usando elementos cerca a la implementación final.

Kogel et al. [39] presenta la metodología de mapeado arquitectural virtual que permite la evaluación cuantitativa de un mapeado entre aplicación y arquitectura por medio de un modelo de prestaciones ejecutable. Recursos de procesamiento compartidos se modelan vía la unidad de procesamiento virtual que permite el mapeado espacial y la ejecución de un grupo de tareas en un único elemento. De forma similar a nuestra estrategia, este entorno acelera la exploración de un largo espacio de diseño por medio de archivos de descripción donde anotaciones de tiempo individuales axial como el mapeado se especifica. Este entorno usa el modelo de programación tCEFSM (máquina de estado finita extendida con comunicación general temporal).
Finalmente, Madsen et al. [40] propone un entorno de modelado a nivel de sistema para modelas SoCs que consisten de multiprocesadores heterogeneos y estructuras de red en chip. De forma similar a CASSE, este entorno cubre modelado abstracto de RTOS, pero a diferencia de nuestro esquema que también incluye información funcional, las tareas de las aplicaciones están representadas solo por anotaciones de tiempo para procesamiento y comunicación.

### 6.3.3 Metodología

CASSE sigue una metodología tipo Y-chart donde la funcionalidad de la aplicación y la arquitectura se modelan independientemente y se combinan en una fase de mapeado separado. Información cuantitativa acerca de la ejecución del sistema se obtiene por medio de la simulación. La herramienta puede ser usada para realizar simulaciones funcionales de solo la aplicación o simulaciones de prestaciones de la aplicación mapeada y ejecutándose en el modelo de la arquitectura. Después de las simulaciones la información obtenida puede ser visualizada y analizada para así guiar optimizaciones posteriores en la arquitectura, aplicación y/o mapeado.

CASSE controla todo las fases en su flujo de ejecución por medio de archivos de descripción textuales. Estos archivos de descripción se leen y analizan por la herramienta durante el tiempo de elaboración para así crear y de forma adecuada configurar el modelo del sistema deseado. El resultado es un modelo ejecutable que es simulado el núcleo de simulación de SystemC. Con esto la herramienta simplifica la exploración de varios escenarios por medio de estos archivos de descripción que pueden ser modificados de forma sencilla para crear una nueva instancia del sistema. Puesto que cambiar los archivos de descripción no requieren recompilar los modelos existentes, un barrido extensivo de los parámetros del sistema se puede automatizar de forma más sencilla. La estructura interna de la herramienta se muestra en la Figure 10.
6.3.3.1 Modelado de la aplicación

CASSE usa un modelo de programación paralelo basado en la interfac de tareas a nivel de transacción llamado TTL [34]. De acuerdo con TTL una aplicación se describe como un gráfico de tareas donde las tareas paralelas se comunican entre ellas por medio de canales unidireccionales. Una tarea es una entidad que realiza computaciones. Las tareas están conectadas a los canales vía puertos y ellas comunican y sincronizan entre ellas por medio de invocar primitivas TTL en sus puertos. Por lo tanto, TTL estrictamente separa computación y comunicación a nivel de aplicación. TTL puede ser usada tanto para desarrollar modelos de aplicación paralelo y como una interfaz de plataforma para integrar módulos HW y SW. Un aspecto positivo de TTL es que solo define las primitivas de la interfaz y su funcionalidad, pero deja su implementación abierta al desarrollador.

En la herramienta, las tareas que contienen la funcionalidad de la aplicación están escritas en C/C++. Esto significa que la funcionalidad por tarea es fija en tiempo de compilación. Sin embargo, la estructura del gráfico de las tareas (i.e. conexiones entre puertos y canales) y su configuración (e.g. tamaño de los canales) se describen en un fichero de texto separado. La herramienta usa ese archivo de descripción para instanciar y conectar juntos las tareas y los canales creando de esta forma un modelo ejecutable de la aplicación independiente de la arquitectura. Dicho modelo de aplicación puede ser simulado usando CASSE para validar la corrección funcional.

6.3.3.2 Modelado arquitectural

CASSE consigue un modelado arquitectural sencillo y rápido por medio de usar un grupo de altamente configurable elementos predefinidos. Estos elementos proporcionan funcionalidad genérica que se puede parametrizar para así cubrir ya sea de forma individual o combinada un amplio rango de componentes arquitecturales. Estos elementos predefinidos son: elementos de procesamiento (PE), elementos de almacenamiento (SE) y elementos de red (NE). Un modelo PE
emula unidades computacionales multitarea genéricos. Un PE incluye un modelo abstracto de un planificador de tareas que soporta esquemas de arbitraje y características avanzadas como interrupciones y preempitividad. Un modelo SE emula elementos de memoria de acceso aleatorio, tales como archivo de registros o memorias RAM. Finalmente, un NE modela una interconexión compartida genérica tipo bus, incluyendo un árbitro programable, un decodificador de direcciones y búferes de entrada opcionales. Todos los elementos predefinidos pueden ser conectados juntos de forma directa por medio de una interface TLM llamada ICCP (protocolo de comunicación entre componentes). ICCP define un protocolo abstracto de comunicación a nivel de dispositivo, incluyendo una interfaz punto a punto y un grupo de primitivas de comunicación entre dos entidades llamadas maestro y esclavo. Tanto la interfaz ICCP y la librería de elementos predefinidos se han desarrollado usando el estándar IEEE 1666 SystemC y el estándar TLM 2.0. Además de estos elementos predefinidos los modelos arquitecturales se pueden extender con nueva funcionalidad por medio de componentes externos (EC). Dichos EC se pueden describir a cualquier nivel de abstracción usando SystemC y se puede conectar de forma directa a los elementos predefinidos siempre y cuando sus interfaces sean acorde al estándar TLM 2.0 GP.

Un archivo de descripción separado se usa para especificar la composición de la arquitectura del sistema (i.e. número de elementos de cada tipo, numero de interfaces por elemento y su interconexión) y su configuración (e.g. mapa de memoria, tamaño de las memorias, latencias en la comunicación por interfaz, política de arbitraje de las tareas, etc.)

6.3.3.3 Mapeado y ejecución

Una de las ventajas principales de la herramienta como entorno unificado es el soporte directo del mapeado de las aplicaciones en el modelo de la arquitectura. Es decir, CASSE soporta el mapeado de las aplicaciones TTL (tareas y canales) en los elementos predefinidos. El código original de las tareas se puede ejecutar directamente en los PE disponibles en el modelo arquitectural. Esta técnica es
normalmente llamada emulación de código (HCE). HCE evita el uso de modelos precisos HW y/o modelos ISS, y por lo tanto reducen el esfuerzo de modelado y permiten simulaciones más rápidas. No obstante, las latencias de tiempo reflejando el coste de computación se tienen que anotar en las tareas. Este se puede realizar de forma manual o por medio de métodos automáticos, como los descritos en [46].

Como en los pasos previos otro archivo de descripción se usa en la herramienta para controlar el procedimiento de mapeado. El resultado de la fase de mapeado es un modelo ejecutable que contiene una instancia de la combinación aplicación/arquitectura. Dicho modelo ejecutable se ejecuta usando el núcleo de simulación de SystemC para así validar tanto la funcionalidad como las prestaciones del sistema.

6.3.3.4 Análisis de las prestaciones

Durante las simulaciones la herramienta obtiene y almacena información acerca de la ejecución del sistema. Esta información permite al usuario analizar e identificar cuellos de botella arquitecturales y posibles optimizaciones del sistema a diferentes niveles. Basándose en este análisis posteriores iteraciones se pueden llevar a cabo donde los modelos de la aplicación y la arquitectura se pueden afinar o un nuevo mapeado seleccionado.

Los elementos predefinidos de CASSE son capaces de tracear y grabar dos tipos de información: estadísticas de prestaciones y transacciones. Dicha recolección de trazas es posible ya que todos los elementos predefinidos incorporan monitores internos que pueden ser individualmente activados para automáticamente recoger estadísticas y grabar transacciones durante la ejecución. Los monitores también se pueden controlar a través de un archivo de descripción textual.
6.3.3.5 Refinamiento

Finalmente, una vez los requisitos esperados se cumplen con un conjunto dado de aplicación/arquitectura/mapeado el sistema está listo para implementación. Los modelos HW se pueden refinar progresivamente desde descripciones más abstractas a descripciones más precisas, por medio de reemplazar los elementos predefinidos con elementos externos (EC) conteniendo los modelos más precisos. Asimismo, módulos SW puede ser directamente tomados en un compilador y posteriormente integrados en el sistema por medio de un EC que contiene un ISS.

6.3.4 Estructura de la herramienta

Como se muestra en la Figure 10, CASSE se estructura en tres capas:

- La capa de presentación. Esta capa sirve como interfaz de usuario que controla la herramienta. Esta capa se compone de las librerías de usuario y los archivos de descripción. Existen dos tipos de librerías de usuario: la librería de tareas conteniendo tareas TTL que conforman la aplicación y la librería de componentes externos que contienen modelos en SystemC para ser añadidos en el modelo arquitectural. Como se ha explicado anteriormente, hay cuatro archivos de descripción: el archivo de aplicación, el archivo arquitectural, el archivo de mapeado y el archivo de análisis.

- La capa de elaboración. Esta capa implementa la funcionalidad principal de la herramienta. Además de un parser que lee e interpreta los archivos de descripción, esta capa contiene dos librerías específicas: la librería de aplicación donde la interfaz TTL se implementa y la librería de elementos predefinidos. La herramienta es capaz de llevar a cabo dos clases de simulaciones: simulación funcional y simulación de prestaciones. Durante la simulación funcional la herramienta solo requiere el archivo de aplicación. Basándose en este archivo la herramienta automáticamente instancia y
conecta juntas las tareas y canales (desde las librerías de usuario y de la herramienta, respectivamente), creando un modelo ejecutable de la aplicación. Durante las simulaciones de prestaciones la herramienta lee e interpreta los archivos de aplicación, arquitectura y mapeado. Los elementos predefinidos y los componentes externos son automáticamente instanciados y conectados siguiendo un esquema modular de acuerdo con el archivo arquitectural. Las tareas y los canales se asignan en específicos elementos PE y SE de acuerdo al archivo de mapeado. Todos los elementos se configuran de acuerdo con el archivo de aplicación y de los parámetros especificados en los otros archivos de descripción. El resultado de este proceso es un modelo ejecutable del sistema.

- La capa del simulador. Los modelos ejecutables se ejecutan en el simulador de SystemC, el cual constituye la tercera capa. Durante las simulaciones las trazas y estadísticas de la simulación se obtienen y graban en los ficheros de salida para inspección y análisis posterior.

### 6.3.5 Conclusiones

Las contribuciones de esta sección son dos:

- Primeramente, el permitir la separación entre la funcionalidad de la aplicación y la arquitectura, así como su posterior mapeado, permite evolucionar el típico esquema de co-diseño HW/SW con más sofisticadas metodologías que encajan mejor con los requisitos para usar un VP en la exploración temprana de arquitecturas. Esto se demuestra con el entorno CASSE implementado en esta tesis doctoral, el cual simplifica el proceso iterativo de crear el modelo de la plataforma HW, mapear el modelo de la aplicación, configurar el sistema completo, simular y analizar los resultados obtenidos.
En segundo lugar, las técnicas de emulación HCE y el modelo abstracto del planificador de tareas, permiten modelar y analizar sistemas multiprocesador y multitarea en tiempo real a niveles de abstracción usando SystemC/TLM.
6.4 Métodos de modelado para la creación de componentes HW reutilizables con TLM

6.4.1 Introducción

Hoy en día los modelos que requieren un buen equilibrio entre velocidad de simulación y precisión se implementan de forma manual. Esta sección propone una estrategia de modelado para crear modelos arquitecturales óptimos en términos de velocidad que puedan ser refinados de forma progresiva con más precisión temporal con un esfuerzo limitado. Las técnicas de anotación de tiempo y los generadores de reloj dinámico son básicas para mantener la velocidad de los modelos alta. Conceptos como modelado jerárquico y disposición dinámica se introducen como la base para permitir el refinamiento de los modelos en su núcleo comportamental. Por otro lado, el concepto de composición estructural es la clave para reducir el esfuerzo de modelado manual.

La idea de composición estructural es especialmente adecuado para modelar la parte comportamental del IP. La idea básica es el crear modelos por medio de componer juntos bloques predefinidos que contienen funciones específicas. Estos bloques tienen que encajar juntos de forma directa y sin fisuras para poder ser efectivos. La justificación detrás de este esquema es el típica regla 80-20, la cual cuando se aplica a modelado expresa que el 80% del modelo se puede reutilizar y solo el 20% se tiene que crear nuevo. Por lo tanto, una librería de bloques genéricos optimizados que se puede usar para rápidamente componer la mayor parte de la estructura y funcionalidad puede de forma significativa reducir el esfuerzo de modelado.

6.4.2 Trabajo relacionado

Idealmente la forma más efectiva de reducir el esfuerzo para crear un modelo SystemC TLM es por medio de automáticamente generarlo desde otra representación del mismo modelo. Dos opciones básicas están disponibles. En la
primera opción, existe una versión RTL del bloque IP, típicamente escrita en un lenguaje HDL como VHDL o Verilog. Un modelo SystemC se puede extraer del código HDL por medio de usar herramientas de generación como [69] o [70]. En la segunda opción, el bloque IP no tiene una contrapartida en RTL pero existe un modelo funcional de referencia escrito en un lenguaje como C o C++. En este caso un modelo SystemC se puede generar por medio de usar herramientas de síntesis de alto nivel como [81] o [82]. Desafortunadamente en ambos casos el modelo resultante es un modelo de bajo nivel que recuerda el estilo RTL con interfaces a nivel de pines y procesos sensitivos a eventos de reloj, lo cual los hace demasiado lentos para ser usados en la mayoría de los usos de un VP.

En [90] y [92] se presentan dos esquemas básicos basados en composición estructural para crear modelos HW IP. En [92] el entorno de simulación UNISIM se describe. UNISIM es un entorno de simulación construido encima de SystemC y una librería de API y características enfocada en el modelado de lógica de control HW. Aunque este trabajo soporta comunicación a nivel de transacciones sus beneficios principales esta en reducir el esfuerzo para modelar componentes HW a un nivel de señales con precisión de ciclo. Por otro lado, el principal fallo de esta solución es que se desvía de los estándares de la industria sin añadir un beneficio claro. En [90] se describe una librería de características de prestaciones especialmente enfocada a crear modelos de procesador (e.g. unidades de ejecución de pipeline, caches, colas de despacho, etc.) Esta librería es parte de la herramienta SLATE, que tiene como objetivo proporcionar análisis temprano de las prestaciones, potencia, características físicas y térmicas de sistemas multiprocesador.

En Cornet et al. [91] introduce una técnica para refinar modelos TLM sin tiempo a modelos con tiempo mientras se mantiene la funcionalidad de los modelos si tocar. Este esquema propone el separar el modelo en dos partes: una parte conteniendo la funcionalidad y otra parte separada conteniendo la información temporal detallada para ambos la comunicación y la computación. Aunque este esquema parece útil para reducir el esfuerzo de modelado, tiene algunas limitaciones inherentes ya que se apoya en un mecanismo de sincronización entre modelos no
estándar. Esto limita el intercambio libre e integración de modelos entre diferentes fuentes. Este trabajo es una extensión a la librería de modelado TAC de STMicroelectronics [66].

En [63] se presenta un esquema que permite reutilizar modelos entre varios niveles de abstracción por medio de separar comunicación, comportamiento y temporización. Cada uno de estos aspectos se soporta por medio de un grupo de interfaces y objetos de modelado parte de la librería SCML [62]. Esta librería proporciona un conjunto de bloques predefinidos en guías de modelado para crear de forma efectiva modelos reutilizable de periféricos en poco tiempo y con una buena velocidad de simulación. En este esquema el reutilizar un modelo para diferentes niveles de abstracción se limita solo a remplazar la parte de comunicación del modelo, dejando la parte comportamental sin cambiar. En la opinión de este trabajo esto no es siempre suficiente, especialmente en modelos complejos donde la granularidad de la parte computacional puede tener un impacto significativo. El trabajo presentado en esta sección extiende las capacidades del esquema SCML por medio de permitir un modelado del comportamiento más estructurado y proporcionando los medios para refinar dicha parte de forma similar como la parte de la comunicación.

6.4.3 Conceptos avanzados de modelado

6.4.3.1 Capas en el núcleo de comportamiento del modelo IP

Varias capas ortogonales se pueden identificar cuando se crea un modelo IP HW. Estas capas tienen un propósito específico dentro del modelo, que se explica a continuación:

- **Comportamiento.** Básicamente la idea es modelar la funcionalidad interna del modelo en una manera estructurada. Las funciones se encapsulan en bloques individuales que se comunican entre ellos por medio de puertos e interfaces predefinidas. Este esquema permite modelar fácilmente el flujo de
control y de datos dentro del modelo y explícitamente identificar los puntos de sincronización. El tener un buen entendimiento de los puntos de sincronización dentro del modelo es vital para reducir el número de cambios de contexto durante la simulación y maximizar la velocidad. Una parte importante de esta capa son bloques especiales que permiten que permitan el modelar maquinas de estado finitas a nivel de transacciones y puertos especializados que permitan una conexión directa con las otras capas.

- **Temporización.** Existen dos formas de modelar información temporal en un modelo TLM: anotaciones de tiempo o sensibilidad al reloj. Anotaciones de tiempo típicamente conlleva a simulaciones más rápidas y modelos sensibles al reloj son típicamente más precisos. En este trabajo se propone una solución basada en objetos de reloj con funcionalidad avanzada los cuales soportan tanto anotación de tiempo como generación de eventos de reloj, pero donde dicha generación se puede activar / desactivar de forma dinámica desde dentro del modelo. Este esquema tiene dos beneficios: primero, permite que los modelos puedan ser refinados con más precisión sin modificar sus interfaces, y segundo permite a los modelos cambiar su nivel de precisión durante simulación.

- **Almacenaje y sincronización.** El objetivo de esta capa es doble: primero, permitir la comunicación y sincronización entre las interfaces del núcleo del modelo IP y la capa de comportamiento y, segundo, permitir la comunicación y sincronización entre los distintos bloques de computación dentro de la capa de comportamiento. Los bloques básicos que componen esta capa están implementados como canales SystemC que contienen interfaces virtuales específicas y las cuales se pueden acceder desde un puerto especializado.

- **Interfaces genéricas.** La idea de esta capa es proporcionar un conjunto de interfaces abstractas y neutras que se puedan usar en todos los modelos IP. Estas interfaces genéricas se enfocan en tres aspectos: reloj, interrupciones y comunicación a nivel de bus. Las interfaces genéricas a nivel de bus no
modelan ningún protocolo específico, pero solo permiten la separación entre la capa de comportamiento y la parte específica de comunicación. Este parte de protocolo específico tiene que implementarse en los adaptadores de interfaz del IP.

6.4.3.2 Modelado jerárquico

Un concepto clave de la estrategia de modelado propuesta en este trabajo es el modelado jerárquico. Este concepto encaja perfectamente con el esquema de capas presentado anteriormente y propone una manera más estructurada de crear la parte interna de los modelos. Básicamente los bloques funcionales son módulos independientes, y los bloques de almacenaje y sincronización son canales que tienen que ser conectados a los bloques comportamentales por medio de puertos. Los bloques instanciados así como sus conexiones conforman la disposición interna del modelo. Los beneficios principales de este esquema son:

- **El código es más estructurado.** Los modelos se componen de bloques separados dos que encapsulan distintas piezas de funcionalidad. Esta solución es escalable y hace más fácil de mantener y depurar modelos complejos.

- **Reutilización.** Con este esquema la mayor parte del modelo se crea utilizando bloques predefinidos y el modelador solo tiene que definir la funcionalidad específica del IP.

- Más simple para diseñadores HW. El modelado jerárquico hace la creación del modelo más intuitiva desde un punto de vista del diseñador HW ya que puede existir una correspondencia uno a uno entre los bloques de la hoja de características del IP y los bloques en el modelo. Además los bloques predefinidos ocultan detalles específicos de SystemC y TLM, haciendo su uso más intuitivo por medio de objetos de más alto nivel (e.g. máquinas de estado finitas).
El refinamiento y la configuración se mejoran. Esto conduce al concepto de disposición dinámica que se explica en la sección siguiente.

El modelado jerárquico también tiene algunos inconvenientes inherentes que se describen a continuación:

- **Impacto en las prestaciones debido a extra direccionamientos.** No obstante, ejemplo complejos muestran que dicho impacto es menor que un 1% del total del tiempo de simulación.

- **Incremento en el número de líneas de código.** Debido al nivel extra de jerarquía, los modelos contienen más líneas de código para describir el instanciamiento de los bloques y sus conexiones. Esto también significa que cada bloque en la capa de comportamiento declara su propio constructor, puertos, variables, etc. Afortunadamente la mayor parte del código extra es estructural y puede ser generado fácilmente por medio de herramientas generadores de código basadas en plantillas.

### 6.4.3.3 Disposición dinámica

La introducción de jerarquía en el interior de los modelos proporciona la posibilidad de cambiar / configurar la disposición del modelo de forma sencilla durante su creación (típicamente durante tiempo de elaboración). Un cambio en la disposición del modelo puede significar una de las siguientes acciones:

- **Añadir o eliminar funcionalidad.** Esto significa que la nueva funcionalidad puede ser añadida o eliminada por medio de instanciar o no ciertos bloques durante la elaboración del modelo. Esto permite crear modelos que pueden comportarse como ligeramente diferentes IP de forma reutilizable por medio de solo cambiar su disposición interna.
• *Refinar la funcionalidad.* Los bloques internos de un IP se pueden reemplazar por versiones diferentes de la misma funcionalidad dependiendo del nivel de precisión requerida. Por ejemplo, asumiendo que un modelo IP de un motor de computación de video se requiere para un cierta simulación donde la velocidad de simulación es más importante que la precisión temporal. Este modelo requeriría que sus bloques comportamentales estén modelos con una granularidad gruesa (e.g. granularidad a nivel de imagen) con poca información temporal en los puntos intermedios. En el caso de que se quiera reutilizar este modelo para un caso donde la precisión temporal sea más importante, entonces la granularidad de las operaciones y la información temporal tiene que ser más detallada. Reemplazando solo aquellos bloques del comportamiento que sean relevantes con versiones que añadan precisión extra, permitiría reutilizar el mismo modelo para diferentes usos con diferentes requisitos.

• *Incrementar el número de bloques del mismo tipo.* Esto es especialmente útil cuando se quiere incrementar el paralelismo de ciertas operaciones dentro de un modelo. Típicamente aquellos modelos que tienen un número configurable de interfaces se pueden beneficiar de la simplicidad de este esquema.

• *Cambiar las conexiones entre los bloques.* Obviamente añadir o remover bloques implica que nuevas o diferentes conexiones entre bloques se tienen que instanciar para crear la nueva disposición.

### 6.4.3.4 Modelado temporal

En esta sección se describen las técnicas para modelar información temporal, mientras se mantiene la velocidad de simulación alta. La idea general es separarse de los modelos a nivel de ciclo que se evalúan en cada ciclo de reloj incluso cuando no están realizando ninguna actividad, los cuales son extremadamente costosos en términos de velocidad de simulación. En lugar de dichos modelos en este trabajo se
proponen modelos precisos en términos de ciclos totales de reloj pero que combinen técnicas de anotación temporal con relojes generadores de eventos controlable que se puedan desactivar cuando el modelo no está activo.

En general los conceptos que se recomiendan cuando se modela información temporal en un modelo SystemC/TLM son los siguientes:

• **Usar técnicas de anotación temporal en lugar de procesos sensibles a los eventos de reloj.** La anotación temporal permite reducir el número de veces que un modelo necesita sincronizar con el núcleo de simulación de SystemC. Con anotación temporal, los retrasos parte de la funcionalidad del modelo se almacena en variable sc_time de SystemC. Dicho tiempo no es consumido inmediatamente, sino que puede acumularse para ser consumido más tarde, cuando se alcance un punto de sincronización obligatorio.

• **Usar generadores de eventos de reloj controlables en el caso de que procesos sensibles al reloj no se puedan evitar.** Si dichos procesos sensibles al reloj no se pueden evitar, al menos la generación de eventos se debería de desactivar cuando el modelo no está activo. Cuando el modelo esté activo otra vez el reloj se podría activar otra vez de forma dinámica.

• **Usar procesos sensibles al reloj que sean de tipo SC_METHOD en lugar de SC_THREAD.** Si un proceso sensible al reloj se tiene que utilizar es más eficiente el utilizar un SC_METHOD en lugar de un SC_THREAD, ya que ellos no producen un cambio de contexto por cada activación. Sin embargo, debido a su naturaleza de no guardar el estado el modelar con SC_METHOD conlleva más esfuerzo.

Cuando se utiliza anotación de tiempo los siguientes aspectos se tienen que tener en cuenta:
• **Separar la información temporal en las interfaces (comunicación) de la información temporal en el interior del modelo (computación).** El tiempo asociado a la comunicación se debe consumir en los adaptadores de interfaz, es decir fuera del núcleo de comportamiento del IP. Por lo tanto la interfaz genérica debe llevar consigo también la información de tiempo del interior del IP hasta los adaptadores de interfaz. El tiempo asociado con la computación se consume en las operaciones dentro de los bloques comportamentales o entre los bloques por medio de los objetos e sincronización y almacenaje.

• **Separar los puntos de sincronización del coste de las operaciones.** Los modelos que contiene procesos SC_THREAD (hilos) tienen que explícitamente ceder el control al núcleo de simulación de SystemC para así permitir que otros hilos en la simulación se puedan ejecutar. Este es así incluso para modelos sin tiempo anotado. La acción de ceder el control se puede implementar por medio de esperar por un tiempo delta cero en la simulación o por medio de consumir la latencia real de las operaciones. Para poder mantener la velocidad de simulación tan alta como sea posible los puntos de sincronización se deben reducir al mínimo, sin por supuesto reducir la funcionalidad del modelo, y consumir el tiempo anotado solo en esos puntos.

• **Usar anotaciones de tiempo relativas en lugar de temporización absoluta.** El uso de operaciones que consumen el tiempo anotado sc_time de forma absoluta limita el uso de los modelos, ya que no podrían entonces ser sensitivos a cambios en la frecuencia de reloj. Para poder crear modelos que puedan cambiar su temporización cuando la frecuencia de reloj cambia (incluso de forma dinámica), la información de ciclo de reloj se tiene que obtener de un reloj externo al modelo. Para este propósito se requiere un puerto especial de reloj ya que el típico puerto sc_in<bool> no funciona con anotación de tiempo. El tiempo total se calcularía por medio de multiplicar el número de ciclos consumidos en el modelo por la información de reloj obtenida a través del puerto especial.
• Notificación retrasada de eventos. Otra técnica que se puede usar de forma eficiente junto con anotación temporal es la notificación retrasada de eventos. Básicamente la idea es la de notificar los eventos que a su vez inician cierta funcionalidad después de un cierto tiempo. Esto ahorra cambios de contexto y contribuye a la aceleración global de las simulaciones.

La temporización asociada con la comunicación tiene que ser modelado en los adaptadores específicos de interfaz de acuerdo con el protocolo de comunicaciones y el nivel de abstracción seleccionado. El núcleo del comportamiento del IP debe enviar transacciones de escritura o lectura genéricas sin coste extra alguno por medio de las interfaces genéricas hacia los adaptadores donde ellos tienen que hacerse cargo de la temporización desde el momento en que la transacción es recibida hasta el momento en que la transacción termina. Puesto que la interacción con la parte comportamental puede influenciar la temporización global de la comunicación, se recomiendo el usar interfaces no bloqueantes que envíen la transacción hasta el adaptador, junto con un mecanismo de sincronización que indica las distintas fases del protocolo de comunicación y/o la disponibilidad del dato. Por ejemplo, SCML proporciona interfaces genéricas que soportan este estilo por medio de su post interfaz y el mecanismo de sincronización implícito en el objeto *scml_array*. De forma similar, TLM 2.0 proporciona la interfaz *nb_transport* con un atributo indicando la fase del protocolo por el mismo motivo.

La temporización asociada con la computación representa el coste de ejecutar el comportamiento / operaciones y debe ser parte de núcleo del modelo. Se pueden identificar dos clases de retrasos temporales en el núcleo del modelo IP:

• Retraso de ida y vuelta (desde el comienzo de un estimulo en las entradas hasta el final del estimulo en las entradas). Este retraso se mide desde el momento en que un interfaz genérica de entrada al IP (e.g. una interfaz de bus o una interrupción) recibe un estimulo (e.g. una transacción) hasta el momento en el que el interior del IP responde al estimulo (e.g. retornando la
llamada a la interfaz). Cuando se usa anotación temporal el retraso de ida y vuelta puede ser un simple parámetro estático anotado en el comportamiento o puede ser calculado de forma dinámica con fórmulas complejas que tengan en cuenta el estado del IP y su configuración. Para poder soportar este tipo de retrasos temporales con anotación de tiempo la interfaz genérica de entrada debe ser capaz de enviar de vuelta el retraso anotado cuando la transacción retorna, el retraso se consume en el adaptador de interfaz.

- **El retraso acción / reacción (desde el comienzo de un estímulo de entrada hasta el comienzo de un estímulo de salida).** Este retraso se mide desde el momento en que una interfaz genérica de entrada al IP (e.g. una interfaz de bus esclava) recibe un estímulo (e.g. una transacción) hasta el momento en que un estímulo de salida se genera como una reacción en una interfaz genérica de salida (e.g. iniciar una transacción o generar una interrupción). Además este retraso se puede dividir en dos partes:

  - **El retraso de activación:** Este retraso indica básicamente el tiempo que lleva el disparar cierto comportamiento en el IP y representa mayormente los retrasos de sincronización y comunicación en el interior de los bloques comportamentales. La técnica de notificación retrasada de eventos es perfecta para modelar este tipo de retraso cuando se usa anotación temporal.
  
  - **El retraso de procesamiento:** Este retraso indica básicamente el tiempo que lleva el computar cierta operación y representa mayormente el coste de las operaciones dentro de un bloque. El tiempo en este tipo de retraso se tiene que consumir explícitamente cuando un punto de sincronización se alcanza.
6.4.4 Conclusiones

En esta sección los métodos avanzados de modelado, modelado jerárquico y disposición dinámica, se proponen para permitir el refinamiento de la parte interna comportamental del modelo con más detalles. Obviamente solo las partes que pueden añadir precisión temporal deben de ser refinadas, manteniendo el resto de las partes sin cambiar.

Otra importante contribución de esta sección es el concepto de composición estructural en modelos TLM, la cual tiene como objetivo el reducir el esfuerzo para crear modelos complejos. La base de esta idea es rápidamente crear modelos a través de conector juntos bloques predefinidos de manera directa.

Finalmente, otra contribución son las tecnicas de modelado y practicas recomendadas para crear modelos óptimos en términos de velocidad de simulación. La idea general es el reemplazar los típicos modelos evaluados cada ciclo de reloj, que son bastantes costosos en términos de velocidad de simulación, con modelos precisos en número total de ciclos que combinan tecnicas de anotación temporal con relojes generadores de eventos que pueden ser activados y desactivados dinámicamente durante la simulación.
6.5 Métodos de modelado para comunicación precisa a nivel de ciclo usando TLM2.0

6.5.1 Introducción

La mayor parte de los modelados precisos a nivel de ciclo están modelados hoy en día a nivel de transferencia de registros con procesos sensibles al reloj e interfaces a nivel de señales / pines, lo cual reduce dramáticamente la velocidad de simulación de los modelos. La razón principal de estas lentas simulaciones es el hecho que la interfaces a nivel de señales y procesos sensibles al reloj producen muchos cambios de contexto en el simulador SystemC, incluso cuando no hay actividad en el modelo. Un beneficio de los modelos TLM es que típicamente no son sensibles a los eventos de reloj y por lo tanto pueden alcanzar simulaciones de velocidad mucho mayores que los modelos RTL. Aunque esto significa que los modelos no son típicamente precisos a nivel de ciclos.

Un aspecto que limitaba el uso de modelos TLM era la falta de un estándar para crearlos. Esto ha llevado a muchas diferentes implementaciones no compatibles entre ellas. Reciente, OSCI ha solucionado este problema por medio de proponer el estándar TLM 2.0 como el estándar de interoperabilidad. El estándar TLM 2.0 cubre varias de las limitaciones del estándar TLM 1.0 con respecto a velocidad de simulación e interoperabilidad. El estándar se enfoca en SoC basados en buses mapeados en memoria y define tres estilos de modelado: sin tiempo (UT), temporización floja (LT) y temporización aproximada (AT). Sin embargo, el estándar no define ningún estilo para modelar temporización a nivel de ciclos (CA).

El objetivo ideal es combinar la precisión de los modelos RTL con la velocidad de los modelos TLM usando el estándar interoperable TLM 2.0. La metodología presentada en este capítulo contribuye a dicho objetivo por medio:

- Demonstrar que el estándar TML 2.0 proporciona los mecanismos suficientes para crear modelos precisos a nivel de ciclo para protocolos de bus específicos.
• Proporcionar una manera estructurada para crear interfaces y adaptadores de bus precisos a nivel de ciclo (BCA) con TLM 2.0.

• Demonstrar que los modelos obtenidos son al menos un orden de magnitud más rápidos que los típicos modelos a nivel de implementación (RTL), mientras se mantiene el nivel de precisión.

6.5.2 Trabajo relacionado

Otros trabajos se han realizado para mejorar la velocidad de los modelos RTL con precisión de ciclo. Tecnicas de supresión de reloj se han usado para ganar velocidad de simulación por medio de deshabilitar ciertas partes del sistema donde el reloj no se necesita [86]. Mientras la supresión de reloj puede resultar una gran mejora en simulación de reloj sobre modelos tradicionales RTL, todavía son mucho más lentos que los modelos TLM. TLM se ha aplicado de forma exitosa en tareas de diseño que van desde desarrollo de SW embebido a verificación funcional. Los requisitos de un modelo TLM dependen de la actividad donde se utilicen. Cai y Gajski [22] presentan una visión general con seis modelos que varían tanto el nivel de abstracción de la comunicación y la computación desde completamente sin tiempo hasta completamente precisos a nivel de ciclo. Los modelos creados es nuestra metodología se parece al modelo funcional a nivel de bus propuesto por dicho trabajo, pero usando los estándares SystemC y TLM 2.0.

Para la exploración de prestaciones se necesitan modelos con un alto nivel de precisión. En [27] Pasricha et al presenta un nivel de abstracción basado en transacciones llamado CCATB (nivel de precisión en número total de ciclos en los límites de una transacción) que tiene una mejora del 55% en velocidad de simulación comparado con modelos a nivel de ciclo tradicionales, aunque perdiendo precisión en protocolos complejos como AXI o OCP. Un estudio del balance entre velocidad y precisión para varios modelos TLM del bus AMBA se presenta en [30]. La conclusión de este trabajo es que mientras modelos TLM pueden mejorar la
velocidad de simulación en un factor de 100 sobre un modelo preciso a nivel de ciclos, la precisión cae hasta un 45% cuando la cantidad de tráfico en el bus es grande.

Algunas empresas que proporcionan modelos IP han desarrollado sus propias API TLM a nivel de ciclo. Ejemplos son los modelos TLM de IBM CoreConnect [73], los canales OCP-IP [74] y las API ARM RealView [75]. El problema con estas implementaciones es que no siguen un único estándar y por lo tanto no son interoperables entre ellas. El trabajo mostrado en esta sección contribuye a extender el alcance del estándar TLM 2.0 por medio de utilizar su mecanismo de extensión para crear modelos precios a nivel de ciclo para protocolos de bus específicos.

6.5.3 Metodología

El enfoque de esta metodología está en crear una interfaz dedicada que permita comunicación BCA para un protocolo dado usando los mecanismo proporcionados por el estándar TLM 2.0. Una importante parte de este proceso es identificar y crear extensiones a la estructura de datos y fases de tiempo genéricas definidas en el estándar. Además, esta metodología propone una forma estructurada de crear los adaptadores necesarios para conectas dicha interfaz BCA a la interfaz genérica de más alto nivel usada en el núcleo comportamental del modelo IP. En este trabajo se ha elegido las interfaces genéricas proporcionadas por la libraría SCML de CoWare. Estas interfaces son la interfaz PVTarget_port (bloqueante) y la interfaz scml_post_port (no bloqueante) que se utilizan para modelar las interfaces maestro y esclavo de un modelo IP.

La metodología propuesta se compone de tres pasos distintos: el análisis del protocolo, el mapeado en TLM 2.0 y la creación del adaptador.

El primer paso de la metodología es el análisis del protocolo que se va a modelar. Hay tres aspectos relevantes del protocolo que se necesitan obtener: los atributos de protocolo, los puntos temporales y las transiciones entre estados. Los atributos
del protocolo (señas) que están relacionadas se agrupan en grupos. Los puntos de tiempo indican donde ciertas señales /atributos son validos y se usan como puntos de sincronización entre el maestro y el esclavo. Finalmente, las transacciones entre estados capturan el comportamiento del protocolo.

El segundo paso de la metodología es el mapeado de los atributos, los puntos temporales y las transiciones entre estados en estructuras pertenecientes a TLM 2.0. Los atributos se mapean directamente en la estructura de datos genérica (GP) o bien se crean extensiones dedicadas a la estructura de datos por medio del mecanismo de extensión proporcionado por el estándar. Los puntos temporales se traducen en extensiones a las fases genéricas del protocolo GP. En esta sección se define un canal como el conjunto de atributos asociados con un punto temporal. Un protocolo puede tener uno o más canales. La salida de este paso es un par de puertos TLM 2.0 para el maestro y el esclavo conteniendo la interfaces dedicadas del protocolo. Otra acción que se realiza en este paso es el mapeo de los canales a las transiciones entre estados definidas en el primer paso. Esto resulta en un conjunto de maquinas de estado para la adaptadores maestro y esclavo.

El paso final es la creación de los adaptadores que permitirán conectar los modelos IP con la interfaz genérica de alto nivel a la interfaz BCA con TLM 2.0 para el protocolo dado. Cuando se implementa los adaptadores la salida de los pasos anteriores se usa para completar una plantilla del adaptador, creando así el adaptador final para el maestro y el esclavo.

Estos pasos son guías generales que se pueden aplicar a la mayoría de los protocolos de bus mapeados en memoria. Ejemplos de los protocolos en el ámbito de este trabajo son, por ejemplo, AMBA AHB y AXI de ARM [71], OCP de OCP-IP [72] y los protocolos DTL y MTL de NXP Semiconductors [77].

Las secciones siguientes explican cada uno de los paso más en detalle. Además los pasos se explican con un ejemplo de un protocolo real como es el protocolo DTL. Este protocolo es una conexión punto a punto que puede ser usada para la
comunicación directa IP a IP o para la conexión a un bus. Es comparable en funcionalidad y complejidad a los protocolos AXI u OCP.

6.5.3.1 Análisis del protocolo

Identificando los atributos del protocolo

El objetivo de esta tarea es el examinar las señales del protocolo y agrupar juntas las señales relacionadas, formando un conjunto de atributos del protocolo.

Para el protocolo DTL la identificación de atributos es simple ya que el protocolo ya define un conjunto de seis grupos de señales. El grupo de sistema contiene las señales de reloj y reset. El grupo de comando consiste de la dirección y otras señales de control de la transacción. El grupo de escritura tiene las señales de datos de escritura. El grupo de lectura tiene las señales de datos de la lectura. El grupo de administración de los buffer tiene señales de control que van del maestro al esclavo. Finalmente, existe un grupo de error/abortar que consiste de señales que se usan para indicar errores. Un total de cuatro grupos se implementan finalmente en la interfaz TLM2.0. El grupo de sistema se elimina ya que el reset y el reloj no son parte de la interfaz TLM y también el grupo de error se mezcla con los otros grupos de forma que no hace un grupo dedicado.

Identificando los puntos temporales

Un punto temporal se define en el momento que un grupo de señales se pone válido y puede ser muestreado por el receptor. En principio, un punto temporal puede ser identificado para cada transición de una señal en el protocolo. Sin embargo, solo unos pocos se requieren para modelar el protocolo con el deseado grado de precisión. Solo aquellas transiciones de señales que indican un punto de tiempo donde la información se está transmitiendo o donde el estado del protocolo se cambia son puntos de tiempo valiosos. Identificar todos los posibles puntos de tiempo solo incrementaría la complejidad del modelo sin añadir extra precisión.
Identificar puntos de tiempo para el protocolo DTL es sencillo ya que el protocolo usa un mecanismo de handshake para indicar cuando un grupo de señales es válido y cuando el receptor ha muestreado las señales. Esto significa que los puntos temporales para el protocolo DTL se pueden alinear a los flancos de subida de las señales de handshake. Un total de ocho puntos de tiempo (dos por grupo de atributos) se identifican para el protocolo DTL.

**Identificando las transiciones entre estados**

Creando una máquina de estado que capture el comportamiento del protocolo de comunicación puede ser bastante complicado, especialmente para protocolos complejos con muchos grupos y puntos de tiempo. Es por lo tanto inteligente el reducir el número de grupos y puntos de tiempo que se describen anteriormente al mínimo requerido, respetando la funcionalidad del protocolo. Con esto el diseño de las maquinas de estado se simplifica, sin reducir la precisión temporal. La base para definir las maquinas de estado son los puntos de tiempo del protocolo y como se relacionan entre ellas. Identificar los flujos de puntos temporales que se soportan en el protocolo ayuda a crear las maquinas de estado. En este estado las maquinas pueden ser descritas a una nivel abstracto. Más detalles se añaden posteriormente durante el mapeado a TLM 2.0.

**6.5.3.2 Mapeado al estándar TLM 2.0**

La primera tarea de este paso es el mapear juntos los grupos de atributos y los puntos de tiempo y crear la extensiones TLM 2.0 requeridas. Dichas estructuras de datos dedicadas (canales) se usan todas juntas para crear las interface y los puertos TLM 2.0. La segunda tarea es la creación de la maquinas de estado en el lado del maestro y en el esclavo.

La interfaz de transporte no bloqueante de TLM 2.0 tiene tres argumentos: un objeto con la estructura de datos, un objeto con las fases de tiempo y un objeto con
tiempo. Los objetos con la estructura de datos y con la fase de tiempo se pueden adaptar para permitir el modelado de un protocolo específico. Si los puntos de tiempo del protocolo se han identificado correctamente estos corresponderán con las fases de tiempo añadidas.

Existen ocho puntos de tiempos identificados en el modelo del protocolo DTL. Estos puntos se mapean directamente a ocho fases en la transacción. Además los canales se mapean a las máquinas de estados en este momento. Primero, la máquina de estado se divide en la FSM del maestro y en la FSM del esclavo asignando a cada una las fases en la transacción que pertenecen a cada lado. Segundo, cada una de las FSM se dividen en múltiple máquinas de estado paralelas más pequeñas, cada una de ellas dedicadas a un canal diferente. Cuando se hace esto se tiene que tener cuidado de asegurar de que todas las dependencias entre canales se sincronizan adecuadamente entre las distintas máquinas de estado.

6.5.3.3 Creando los adaptadores

El paso final es el crear los adaptadores para los lados del maestro y el esclavo que se necesitan para adaptar la interfaz genérica SCML a la interfaz específica con TLM 2.0. Ya que la estructura básica es siempre la misma, se ha creado una plantilla para los adaptadores que simplifica su modelado.

Las Figure 62 y Figure 63 muestran dichas plantillas. Cada adaptador tiene una parte activa que inicia transacciones de salida y una parte reactiva que procesa las transacciones entrantes. Los puertos TLM 2.0 contienen tanto un interface de ida como una de vuelta. El camino de ida se usa para enviar transacciones del maestro al esclavo, mientras que el camino de vuelta se use para enviar transacciones del esclavo al maestro. Por lo tanto, la parte activa del maestro maneja el camino de ida y la parte activa del esclavo maneja el camino de vuelta. La parte reactiva de los adaptadores manejan las correspondientes transacciones de entrada. Entre los caminos de ida y vuelta hace falta una capa de sincronización en los adaptadores. Finalmente, ambos adaptadores tienen una parte que maneja la interfaz genérica
Métodos de modelado para comunicación precisa a nivel de ciclo usando TLM 2.0

de alto nivel. También esa parte se tiene que sincronizar con el resto de la funcionalidad del adaptador.

6.5.4 Verificación de los adaptadores

Para verificar si los adaptadores TLM 2.0 para DTL están operando a nivel de ciclo con la precisión requerida y de acuerdo con el protocolo, un banco de pruebas de verificación utilizando un componente de verificación (eVC) implementado con Specman se ha utilizado. El tráfico se monitoriza y se extraen señales de la interfaz para que así el chequeador del protocolo de Specman que opera a nivel de señales compruebe que el comportamiento a nivel de ciclo es correcto.

6.5.5 Resultados

Las prestaciones de la implementación de los adaptadores se han testeado con un simple esquema donde un generador de tráfico maestro se conecta a una memoria esclava ambos utilizando los adaptadores DTL TLM 2.0.

La velocidad de simulación de este banco de pruebas se mide en ciclos de reloj por segundo y se calcula de la manera siguiente:

$$CPS = \frac{t_{sim}}{(t_{cpu} * d_{cycle})}$$

donde $t_{sim}$ es el tiempo que se quiere simular, $t_{cpu}$ el tiempo físico que lleva simular y $d_{cycle}$ es el periodo de reloj de referencia. La Figure 66 muestra los resultados obtenidos. El generador de tráfico envía dos millones de transacciones de lectura y escritura alternativas. El test se realiza para tamaños de transacciones de 1 a 64 palabras. El test simula dos configuraciones: la configuración 1 no tiene retrasos y en cada ciclo de reloj un dato se transmite. La configuración 2 introduce retrasos en el procesamiento del comando y el dato.
Como se observa en dicha figura, el número de ciclos por segundo se incrementa para transacciones con tamaño mayor. Esto es debido a que tanto \( t_{\text{sim}} \) como \( t_{\text{CPU}} \) se incrementan a diferentes tasas. Además se observa que en la configuración 2 aunque se incrementa el número de ciclos simulados, el tiempo que llevar realizar la simulación permanece el mismo. Esto significa que el ratio de ciclos por segundo se mejora cuando los modelos tienen retardos de computación mezclados con los retardos de comunicación. La razón para esto es que los adaptadores no son sensibles al reloj sino que usan anotación de tiempo y solo son activos cuando una transacción está ocurriendo.

### 6.5.6 Conclusiones

En esta sección se presenta una metodología que permite crear interfaces precisas a nivel de ciclo para protocolos de bus y sus correspondientes adaptadores. Este trabajo demuestra que: 1) el mecanismo de extensión del estándar TLM 2.0 se puede utilizar para modelar comunicación precisa nivel de ciclo y 2) que la velocidad de simulación de los modelos resultantes son casi dos órdenes de magnitud mas rápidos que los típicos modelos RTL a nivel de señales, mientras se mantiene el mismo nivel de precisión.
6.6 Conclusiones y líneas futuras de investigación

6.6.1 Conclusiones

Los métodos de diseño a nivel de sistema (ESL) están siendo propuestos en la industria de los semiconductores como un complemento a las soluciones convencionales, con el objetivo de mejorar la productividad de los diseñadores y aliviar las tareas de diseño y verificación de sistemas en un único chip. ESL se está estableciendo rápidamente en la industria de los semiconductores gracias a los estándares SystemC y TLM2.0 (modelado a nivel de transacciones) que permiten elevar el nivel de abstracción de los diseños por encima del típico nivel a transferencia de registros (RTL).

Más concretamente, ESL está siendo aplicado en los flujos de diseño de forma exitosa por medio de prototipos virtuales (VP) descritos en SystemC y TLM. Un prototipo virtual es básicamente un modelo en software que describe la funcionalidad y temporización de la arquitectura hardware del sistema. Los prototipos virtuales se pueden aplicar a diversas tareas durante el ciclo de diseño. Los típicos usos de un prototipo virtual están en el desarrollo de software embebido antes de que un prototipo físico del hardware esté disponible o la exploración de diversas alternativas de implementación durante la fase de especificación del chip, entre otros.

El problema aparece cuando tenemos en cuenta que estos diferentes usos típicos de un prototipo virtual tienen diferentes requisitos en términos de velocidad de simulación, precisión en los resultados, etc., que hace muy difícil el elegir un único estilo para modelar un prototipo virtual de forma que cumpla todos los requisitos para todos los casos. Además el crear diferentes modelos para diferentes casos, teniendo que mantener los diferentes modelos consistentes, añade una gran complejidad y por lo tanto un elevado coste al desarrollo de dichos prototipos virtuales que reduce de forma drástica los beneficios de las metodologías a nivel de sistema. Es por lo tanto obvio que métodos y técnicas que permitan reutilizar y...
refinar modelos para diferentes casos utilizando prototipos virtuales es clave para el éxito de las tecnologías ESL.

Esta tesis doctoral tiene como mayores contribuciones métodos, herramientas y librerías de modelado a nivel de sistema que permiten reutilizar y refinar modelos para diferentes niveles de abstracción, basándose enteramente en el estándar IEEE SystemC y el nuevo estándar de modelado TLM2.0.

Más concretamente las contribuciones se centran en dos pasos del proceso de refinamiento de los modelos: el paso de refinamiento de un modelo funcional a un modelo arquitectural y el paso de refinamiento gradual dentro de los modelos arquitecturales para añadir precisión temporal.

- En el caso de refinamiento de modelos funcionales a modelos arquitecturales esta tesis doctoral propone métodos de modelado que se basan en: 1) definir una clara separación entre las funciones que componen una aplicación y los detalles de la arquitectura y 2) técnicas de mapeado que permiten mezclar funciones y elementos arquitecturales de forma conjunta. Estos conceptos y técnicas se han implementado en la herramienta CASSE, la cual simplifica el proceso iterativo de crear el modelo de la arquitectura, mapear el modelo de la aplicación, configurar el sistema completo, simular y analizar los resultados obtenidos.

- En el caso de refinamiento gradual dentro de los modelos arquitecturales para añadir precisión temporal el concepto clave propuesto en este trabajo es la separación del modelo arquitectural en dos partes diferencias: comportamiento e interfaces, con el objetivo de simplificar el refinamiento de cada una de las partes. Esta tesis doctoral contribuye métodos y conceptos avanzados para el modelado TLM, como por ejemplo el modelado jerárquico, composición estructural dinámica, objetos tipo reloj con funcionalidad avanzada, etc., con el objetivo de crear modelos arquitecturales refinables óptimos en velocidad con un esfuerzo limitado. Finalmente, nuevos métodos se proponen también para la creación de adaptadores de interface con precisión a nivel de ciclo de reloj para
protocolos de bus específicos usando el nuevo estándar TLM 2.0. Esta trabajo de muestra que: 1) el mecanismo de extensión del estándar TLM 2.0 puede ser usado para crear modelos de interfaces a nivel de ciclo y 2) la velocidad de simulación de los modelos resultantes es alrededor de dos órdenes de magnitud mayor que los típicos modelos RTL a nivel de pines, mientras que el nivel de precisión temporal se mantiene.

6.6.2 Líneas futuras de investigación

Hay dos aspectos principales que merecen una mayor investigación como continuación de este trabajo:

- **Soporte para la reconfiguración dinámica de aplicaciones.** Este aspecto se puede considerar una continuación del trabajo realizado en el área de separación y mapeado entre la aplicación y la arquitectura. Las bases detrás de esta idea se dan por el hecho de que los dispositivos de hoy en día ejecutan una combinación de diferentes aplicaciones simultáneamente, como por ejemplo video, audio, radio, seguridad, internet, etc., proporcionando al usuario una experiencia multimedia vibrante y dinámica. Nuevos métodos y herramientas son necesarios para modelar y analizar los aspectos dinámicos de dichos sistemas, para así estudiar si impacto en las prestaciones globales del sistema lo antes posible en la fase de especificación del diseño. Todos esos aspectos dinámicos son muy difíciles de estimar en estado tempranos del diseño, lo que puede llevar a implementaciones sobredimensionadas muy costosas o la eliminación de características en el diseño final que reducirán la viabilidad de los dispositivos en el mercado. Tres tipos de aspecto dinámicos en la reconfiguración de aplicaciones se pueden distinguir:
  - Cambios en la ejecución: lo que significa, por ejemplo, parar una aplicación que se está ejecutando o comenzar una aplicación que se estaba ejecutando durante las simulaciones.
Cambios en la topología: lo que significa que las tareas o canales que componen un modelo de la aplicación se puede añadir o eliminar, y sus conexiones cambiarse.

Cambios en el mapeado: lo que significa que las tareas se podrían conectar a distintos procesadores o que los canales se podrían asignar a otras memorias en la arquitectura durante las simulaciones.

El modelar todos estos aspectos se encuentran con la limitación fundamental del lenguaje SystemC. Es por lo tanto que nuevas técnicas se requieren para superar estas limitaciones sin romper la compatibilidad con los estándares actuales.

- **Automatizar la generación de los modelos arquitecturales TLM.** El código estructural representando las partes internas de los modelos se podrían generar de forma automática desde una herramienta gráfica basada en el lenguaje SysML derivado de UML. Esto permitiría utilizar conceptos avanzados MDD [100] al más manual proceso de modelado TLM. Además, los adaptadores de interfaz BCA con TLM 2.0 se podrían generar automáticamente siguiendo técnicas similares a las presentadas en [78].

Embedded market forecasters, 2003

Proceedings of the 40th IEEE/ACM Design Automation Conference, ACM Press,
pp. 419-424, 2003

Automation and Test in Europe, 2004


of EUROMICRO DSD 2004


Proceedings of ACM Design Automation Conference, San Diego, California, USA, June 2004


[57] ARTEMI project (TIC2003-09687-C02-02) under the Spanish Ministry of Education and Sciences


[73] IBM. CoreConnect SystemC TLMs. www.ibm.com

[75] ARM RealView ESL APIs. [www.arm.com](http://www.arm.com)

[76] Specman verification environment, [www.cadence.com](http://www.cadence.com)


