Contribution à l'amélioration des plateformes virtuelles SystemC/TLM : configuration, communication et parallélisme

Guillaume Delbergue

► **To cite this version:**

Guillaume Delbergue. Contribution à l'amélioration des plateformes virtuelles SystemC/TLM : configuration, communication et parallélisme. Electronique. Université de Bordeaux, 2017. Français. NNT : 2017BORD0916 . tel-01778172

HAL Id: tel-01778172
https://theses.hal.science/tel-01778172

Submitted on 25 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

université de BORDEAUX

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DES SCIENCES DE L'INGÉNIEUR
SPÉCIALITÉ : ÉLECTRONIQUE

par **Guillaume DELBERGUE**

Advances in SystemC/TLM Virtual Platforms : Configuration, Communication and Parallelism

Directeur de thèse : Christophe JÉGO
Co-encadrant de thèse : Bertrand LE GAL

soutenue le 18 décembre 2017

Jury :

Jean-Philippe DIGUET	Directeur de Recherche CNRS	Lab-STICC	*Président du jury*
François PÊCHEUX	Professeur des Universités	Université Pierre-et-Marie-Curie (Paris 6) - LIP6	*Rapporteur*
François VERDIER	Professeur des Universités	Université Nice Sophia Antipolis - LEAT	*Rapporteur*
Christophe JÉGO	Professeur des Universités	Bordeaux INP, IMS	*Directeur*
Bertrand LE GAL	Maître de Conférences	Bordeaux INP, IMS	*Co-encadrant*
Mark BURTON	Fondateur	GreenSocs	*Examinateur*

PhD realized at the laboratory INTÉGRATION DU MATÉRIAU AU SYSTÈME (IMS)
of Bordeaux, inside the CSN team, CONCEPTION group.

Université de Bordeaux, IMS Laboratory
UMR 5218 CNRS - Bordeaux INP
351 Cours de la Libération
Bâtiment A31
33405 Talence Cedex
FRANCE

PhD carried out within a CONVENTION INDUSTRIELLE DE FORMATION PAR LA RECHERCHE
(CIFRE) with GREENSOCS.

GreenSocs
Le Bourg
24380 Chalagnac
FRANCE

They did not know it was impossible so they did it.

Mark Twain

Acknowledgements

***

First and foremost, I would like to express my sincere gratitude to my University supervisors Prof. Christophe Jégo and Associate Prof. Bertrand Le Gal for the continuous support of my Ph.D study and related research, for their patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better supervisors and mentors for my Ph.D study.

Besides my University supervisors, I would like to thank my industrial funding and supervisor Dr Burton Mark for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

My sincere thanks also goes to teachers-researchers from circuit design group at IMS Laboratory. I am also grateful to Ph.D students, Ph.D, engineers and interns with whom I shared discussions, experiences, many meals, good times... It was great sharing our work room with all of you during last three years. The group has been a source of friendships as well as good advice and collaboration.

Last but not the least, I would like to thank my companion and my family for supporting me throughout writing this thesis and my my life in general.

Abstract

***

The market for Internet Of Things (IOT) is on the rise [140]. It is predicted to continue to grow at a sustained pace in the coming years. Connected objects are composed of dedicated electronic components, processors and software. The design of such systems is today a challenge from an industrial point of view [168][49]. This challenge is reinforced by market competition and time to market that directly impact the success of a system [199]. In a current design process involves the development of a specification. Initially, the team in charge of hardware development begins to design the system. Second, the application part can be done by software developers. Once the first hardware prototype is available, the software team can then integrate their part and try to validate the functionality. This step may reveal defects in the software but also in the hardware architecture. Unfortunately, the discovery of these errors occurs far too late in the design process, could impacts the marketing of the system and potentially its success. In order to ensure that the hardware and software designs will work together as early as possible, methodologies based on the SystemC / Transaction Level Modeling (TLM) standard have been widely adopted. They involve the modelling and simulation of the proposed hardware architectures. During the initial phases of a product's design, they enable the software and hardware team to share a virtual version of the (future) system. This virtual version is more commonly referred to as a virtual platform. It facilitates early software development, test and validation; reduces material cost by limiting the number of prototypes; saves time and money by reducing risks. However, connected objects are increasingly incorporating hardware and software features. As the requirements have evolved, the SystemC / TLM simulation standard no longer meets all expectations. It includes aspects related to the simulation of systems composed of many functionality, disparate communication protocols but also complex and time consuming models during the simulation. Some works have already been carried out on these subjects. However, as the number of components increases, all forms of interoperability of models and tools become increasingly difficult to handle. Moreover, most of the research has resulted in solutions that are not inter-operable and can not reuse existing models. To solve these problems, this thesis proposes a solution for configuring SystemC / TLM models. It is now part of the standard Configuration, Control and Inspection (CCI). In a second step, the modeling of high-level abstraction communication protocols (TLM Loosely Timed (LT) and Approximately Timed (AT)) has been studied, as it relates to non-bus protocols. An evolution of the standard to improve support, interoperability and reuse is also proposed. In a third step, a change of the SystemC standard and more precisely of the behavior of the simulation kernel has been studied to support asynchronous events. These open the way to parallelization and distribution of models on different threads / machines. In a fourth step, a solution to integrate Central Processing Units (CPU) models integrated in Quick EMUlator (QEMU), a system emulator / virtualizer, has been studied. Finally, all these contributions have been applied in the modeling of a set of objects connected to a gateway.

Keywords: SystemC, TLM, Virtual Platform, Configuration, Communication, Parallelism

Résumé

***

Le marché de l'Internet des Objets (IdO) est en pleine progression. Il va continuer à croître et à se développer à un rythme soutenu dans les prochaines années. Les objets connectés sont constitués de composants électroniques dédiés, de processeurs et de codes logiciels. La conception de tels systèmes constitue aujourd'hui un challenge au niveau industriel. Ce challenge est renforcé par la concurrence du marché et le délai de commercialisation qui impactent directement sur le développement d'un système. Le processus de conception actuel consiste en l'élaboration d'un cahier des charges. Dans un premier temps, l'équipe en charge du développement matériel commence à développer le produit. Ensuite, la partie applicative peut être mise au point par les développeurs logiciels. Une fois le premier prototype matériel disponible, l'équipe logicielle peut alors intégrer sa partie et tenter de la valider fonctionnellement. Cette étape peut mettre en lumière des défauts dans le logiciel mais aussi lors de la conception matérielle. Malheureusement, la découverte ce type d'erreurs intervient beaucoup trop tard dans le processus de conception retardant la commercialisation du système. Afin de sécuriser au plus tôt les développements matériel et logiciel, des méthodologies basées sur le standard SystemC/Transaction Level Modeling (TLM) ont été proposées. Elles permettent de modéliser et de simuler du matériel. Durant les phases amont de conception d'un système, elles permettent de mettre en commun une version virtuelle du (futur) système entre les équipes logicielle et matérielle. Cette version virtuelle est plus couramment appelée plateforme virtuelle. Elle permet de tester et de valider le plus tôt possible lors du cycle de conception, de réduire le coût matériel en limitant la fabrication de prototypes, mais aussi de gagner du temps et donc de l'argent en diminuant les risques. Or, les objets intègrent de plus en plus de fonctionnalités aux niveaux matériel et logiciel. Les besoins ayant évolué, le standard de simulation SystemC/TLM ne répond plus à l'heure actuelle à toutes les attentes. Ces attentes concernent plus particulièrement les aspects liés à la simulation de systèmes composés de nombreuses fonctionnalités, de protocoles de communication disparates mais aussi de modèles complexes et consommateur de temps pendant la simulation. Des activités de recherche ont déjà été menées sur ces sujets. Cependant, elles ont pour la plupart abouti à des solutions qui ne sont pas interopérables. Les solutions existantes ne permettent donc pas de bénéficier de la réutilisation des modèles de la littérature. Afin de répondre à ces problèmes, une solution permettant la configuration de modèles SystemC/TLM a été recherchée. Cette dernière fait désormais partie du standard Configuration, Control and Inspection (CCI). Dans un second temps, la modélisation de protocoles de communication à un haut niveau d'abstraction (TLM Loosely Timed (LT) et Approximately Timed (AT)) a été étudiée, et plus précisément des protocoles de type non bus. Une évolution du standard actuel permettant d'améliorer le support, l'interopérabilité, la réutilisation a été proposée dans le cadre de la thèse. Ensuite, une évolution du standard SystemC et plus précisément du comportement du noyau de simulation a été étudiée pour supporter l'attente d'événements asynchrones. Ce type d'événement ouvre la voie à la parallélisation et la distribution de modèles sur différents threads / machines. Enfin, une solution permettant l'intégration de modèles de Central Processing Units (CPU) intégrés dans Quick EMUlator (QEMU), un émulateur / virtualisateur de système, a été étudiée. Finalement, toutes ces contributions ont été associées à travers la modélisation d'un ensemble d'objets connectés à une passerelle.

Mots clefs : SystemC, TLM, Plateforme Virtuelle, Configuration, Communication, Parallélisme

Contents

***

List of Figures xvii

List of Tables xxi

Acronyms xxiii

Introduction 1

1 Overview of System On Chip design flow 5
- 1.1 Introduction 5
- 1.2 System on Chip 7
  - 1.2.1 Introduction 7
  - 1.2.2 Methodology 9
  - 1.2.3 Modelling and simulation 11
  - 1.2.4 Virtual platforms 12
  - 1.2.5 Software development 12
- 1.3 Overview of SystemC and TLM 13
  - 1.3.1 The standard serialization library 13
  - 1.3.2 Scheduler 15
  - 1.3.3 Timing 16
  - 1.3.4 Overview of TLM-1.0 17
  - 1.3.5 Overview of TLM-2.0 18
    - 1.3.5.1 Transport 19
    - 1.3.5.2 Socket 20
    - 1.3.5.3 Payload 20
    - 1.3.5.4 Phase 21
    - 1.3.5.5 Timing and quantum 21
    - 1.3.5.6 Conclusion 22
- 1.4 Challenges for virtual platform modeling 23
  - 1.4.1 Virtual platform configuration 23
    - 1.4.1.1 Models and virtual platforms 23
    - 1.4.1.2 Interoperability and tools 24
  - 1.4.2 Models of SoC protocols 25
    - 1.4.2.1 Generic TLM-2.0 like interconnect standard 25
    - 1.4.2.2 Non unidirectional protocols 26
  - 1.4.3 Virtual platform simulation speed 27
    - 1.4.3.1 Improve platform simulation speed 27
    - 1.4.3.2 Simulation speed up requirements 28
- 1.5 Conclusion 28

2 Configuration, Control and Inspection 31
2.1 Introduction 31
2.2 Needs for simulation configuration features 33
2.2.1 Introduction 33
2.2.2 Configuration requirements 33
2.2.3 Without a configuration solution 34
2.3 Related works 37
2.3.1 Model configuration 37
2.3.2 Virtual platform configuration 37
2.3.3 Dynamic configuration 38
2.3.4 Backward compatibility 39
2.3.5 Conclusion 40
2.4 Configuration, Control and Inspection solution 40
2.4.1 Introduction 40
2.4.2 Overview 42
2.4.3 Parameter 43
2.4.4 Broker 45
2.4.5 Originator 47
2.4.6 Notification of read, write, creation and destruction of parameters 48
2.5 Performance analysis 50
2.5.1 Raw 51
2.5.2 Concrete usage 52
2.5.3 Callback 53
2.5.4 Conclusion on performance evaluation 53
2.6 Breadth of the standard 54
2.7 Limitations of the standard 54
2.8 Conclusion 55

3 TLM for non memory mapped protocols 57
3.1 Introduction 57
3.2 Modeling communications in virtual platforms 58
3.2.1 Introduction 58
3.2.2 Towards a definition of a transaction 60
3.2.3 OSI and TLM 60
3.3 Related works on abstract communications 62
3.4 Evaluation of protocols 65
3.4.1 Introduction 65
3.4.1.1 “One to One” protocols 66
3.4.1.2 “One to Many” protocols 67
3.4.1.3 “Many to Many” protocols 68
3.4.2 Modeling requirement summary 70
3.4.3 Interconnection 71
3.4.4 Conclusion 71
3.5 Proposed improvements of TLM 71
3.5.1 Introduction 71
3.5.2 TLM Transport 72
3.5.2.1 Socket and binding 73
3.5.2.2 Payload 75
3.5.2.3 Phases 78
3.5.3 Conclusion 78

3.6 Protocol configuration check with CCI standard . . . 79
3.6.1 Introduction . . . 79
3.6.2 CCI standard applicability . . . 79
3.6.3 Protocol configuration check . . . 80
3.6.4 CCI meta-data interoperability . . . 81
3.6.5 CCI meta-data limitations . . . 82
3.6.6 Conclusion . . . 82
3.7 Future works . . . 83
3.7.1 Software emulated protocol . . . 83
3.7.2 Pin functions . . . 83
3.8 Conclusion . . . 84

4 Parallelism in SystemC/TLM **85**
4.1 Introduction . . . 85
4.2 Related works . . . 86
4.2.1 Requirements . . . 86
4.2.2 Parallelism inside a SystemC kernel . . . 88
4.2.3 Multiple SystemC kernels without quantum . . . 89
4.2.4 Multiple SystemC kernels with quantum . . . 89
4.2.5 Asynchronicity . . . 91
4.2.6 Conclusion . . . 92
4.3 Asynchronous parallelization . . . 92
4.3.1 Asynchronous event based solution . . . 92
4.3.1.1 Introduction . . . 92
4.3.1.2 Asynchronicity and asynchronous event . . . 93
4.3.1.3 Modification of the SystemC kernel . . . 95
4.3.1.4 Conclusion . . . 96
4.3.2 Asynchronous channel solution . . . 97
4.3.2.1 Introduction . . . 97
4.3.2.2 Callback approach . . . 97
4.3.2.3 Asynchronicity and channel . . . 98
4.3.2.4 Formal function definitions . . . 99
4.3.2.5 Implementation in the SystemC kernel . . . 99
4.3.2.6 Conclusion . . . 100
4.4 Synchronization and quantum impact on parallelization . . . 100
4.4.1 Ordering and timing of the simulation . . . 100
4.4.2 Endless quantum keeper . . . 102
4.4.2.1 Notification system . . . 102
4.4.2.2 Quantum keeper improvement . . . 102
4.4.2.3 Conclusion . . . 103
4.4.3 Quantum based synchronization solutions . . . 103
4.4.3.1 Introduction . . . 103
4.4.3.2 Static quantum . . . 103
4.4.3.3 Windowed quantum . . . 104
4.4.3.4 Conclusion . . . 105
4.5 Experimental results . . . 105
4.5.1 Introduction . . . 105
4.5.2 Two SystemC kernels without time synchronization . . . 106
4.5.3 Two SystemC kernels with a quantum based synchronization . . . 106
4.5.4 Summary . . . 108

4.6 Conclusion . . . 108

5 Application 109
5.1 Introduction . . . 109
5.2 Requirements . . . 110
5.3 QBox: a SystemC CPU model based on QEMU . . . 111
5.3.1 Introduction . . . 111
5.3.2 Time and synchronization in QBox . . . 111
5.3.3 Multithread . . . 112
5.3.4 Impact of multithread for QBox SMP . . . 112
5.3.5 Conclusion . . . 113
5.4 The virtual platform . . . 114
5.4.1 Architecture . . . 114
5.4.2 Configuration . . . 115
5.4.3 Parallelism . . . 116
5.4.4 Protocols . . . 118
5.5 Experimental results . . . 118
5.5.1 Introduction . . . 118
5.5.2 Quantum . . . 118
5.5.3 Trace using CCI parameters . . . 120
5.5.4 Impact of CCI on the simulation execution time . . . 121
5.5.5 Exploration of the impact of the node CPU frequency . . . 121
5.5.6 Evaluation of the improved TLM standard . . . 122
5.5.7 Exploration of the parallelism in the simulation . . . 123
5.5.8 Enhanced quantum keeper . . . 124
5.6 Conclusion . . . 125

Conclusion 127

Appendix 131
A.1 CCI context . . . 131
A.1.1 The working group . . . 131
A.1.2 The CCI standard . . . 131
A.1.3 More about CCI callbacks . . . 134
A.1.4 The CCI standardization process . . . 134
A.1.5 CCI parameter lifetime: destruction and resurrection . . . 135
A.1.6 Broker details . . . 135
A.2 TLM-2.0 improvements . . . 137
A.2.1 OSI . . . 137
A.2.2 Protocols . . . 137
A.2.2.1 I2C . . . 137
A.2.2.2 CAN . . . 137
A.2.3 Improved TLM Quantum Keeper . . . 138
A.2.4 Improved TLM-2.0 blue print . . . 139
A.2.5 Software emulated protocol . . . 140
A.2.5.1 Introduction . . . 140
A.2.5.2 The definition of software protocol with TLM . . . 140
A.2.5.3 A first approach . . . 141
A.2.5.4 Limitations . . . 142
A.2.5.5 Conclusion . . . 142
A.3 Parallelism . . . 143

A.3.1 Two SystemC kernels without time synchronization 143

Bibliography **157**

List of publications **159**

Résumé étendu **161**

List of Figures

***

1 Conventional design flow and the design flow with a virtual platform [181] 1
2 Thesis breakdown 3

1.1 Microelectronic evolution - Yesterday's chip is today's function block! [128] 5
1.2 2008: iPhone 3G mainboard [169] 6
1.3 2016: iPhone 7 mainboard [70] 7
1.4 System on Chip (SoC) cost evolution [83] 8
1.5 OMAP 4470 SoC [180] 8
1.6 SoC V-Model [196] 9
1.7 SoC design flow [195] 10
1.8 Cortex M3 Virtual Platform 12
1.9 Linux OS layers [182] 13
1.10 SystemC architecture [2] 14
1.11 SystemC abstraction position [46] 14
1.12 SystemC models [46] 15
1.13 SystemC scheduler 16
1.14 Simulation time vs wall-clock time [113] 17
1.15 TLM-1.0 overview 17
1.16 TLM-1.0 UML 18
1.17 TLM-2.0 behaviour [135] 19
1.18 TLM-2.0 blocking and non blocking overview [135] 19
1.19 TLM-2.0 transport interfaces 20
1.20 The time quantum [135] 22
1.21 Intel Stratix 10 [86] 23
1.22 Intel Stratix 10 partial memory map [86] 24
1.23 Configurable models 24
1.24 Non memory mapped protocols in SoCs 25
1.25 TLM-2.0 router 26
1.26 Interconnection of different simulators 27
1.27 Complex SystemC models in the same host thread 27
1.28 The parallelism in different forms 28

2.1 Parameters of SoCs 31
2.2 SystemC simulation parameters 32
2.3 Requirements for a control and inspection standard 33
2.4 Example of configurable timer 34
2.5 CCI scope and initial focus 41
2.6 CCI parameters and broker 41
2.7 CCI configuration classes and use model 42
2.8 CCI Parameter and CCI Parameter Handle 43
2.9 CCI parameter and handle hierarchy 45
2.10 CCI broker and CCI broker Handle 46
2.11 CCI private broker hierarchy 46
2.12 CCI private broker registration 47

2.13 CCI originator mechanism 48

3.1 TLM-2.0 overview in virtual platforms 58
3.2 Zynq SoC 59
3.3 SystemC channel description 59
3.4 TLM socket principle 60
3.5 TLM-2.0 parts 61
3.6 OSI model layers 61
3.7 Example of non memory mapped communications 62
3.8 SPI CABA level vs TLM level [145] 64
3.9 Parallel input/output controller on SAM3X driving PIN input/output from multiple peripherals 65
3.10 One to one protocol scheme 66
3.11 UART model with TLM-2.0 66
3.12 UART frame, 8 bits, 1 stop bit 67
3.13 One to many protocol scheme 67
3.14 Many to many protocol scheme 68
3.15 A I2C frame structure 69
3.16 A CAN extended frame structure 69
3.17 Transport interfaces for memory mapped and non memory mapped protocols. Grey = existing. Yellow = added. Orange = modified. 72
3.18 Bidirectional socket scheme 73
3.19 Generic TLM initiator socket with backward compatibility 74
3.20 Bidirectionnal sockets and “One to Many” / “Many to Many” protocols 75
3.21 Base payload description. Yellow = added. Orange = modified. 76
3.22 TLM I2C payload inheriting from TLM Base Payload 76
3.23 Universal Asynchronous Receiver Transmitter (UART) protocol with the improved version of TLM 78
3.24 Example of protocol meta-data exchange 79
3.25 Protocol meta-data embedded in sockets 81
3.26 Meta-data interoperability 81
3.27 UART protocol example with the improved version of TLM and CCI 82
3.28 TLM and software protocol 83

4.1 NXP i.MX 8 heterogeneous SoC that contains ARM A53, A72 and M4F cores [131] 85
4.2 Multi-core heterogeneous platform 86
4.3 Multi-core heterogeneous platform based on different SystemC kernels and threads 87
4.4 Parallelism inside a SystemC kernel 88
4.5 Parallelism illustration between multiple SystemC kernels 89
4.6 Multiple SystemC kernels with a quantum 90
4.7 Asynchronicity and potential deadlock 91
4.8 SystemC scheduler with asynchronous mechanisms 93
4.9 How the asynchronous event waiting mechanism works 94
4.10 Updated and new classes in the SystemC kernel. Grey = existing. Yellow = added. Orange = modified. 95
4.11 Principle of the SystemC scheduler channel update 98
4.12 SystemC kernel with other TLM models in different threads 98
4.13 Execution example with a static quantum 104
4.14 Execution example with a windowed quantum 104
4.15 SystemC kernels that run as producer and consumer 106

5.1 IOT platform use case 109
5.2 IOT platform architecture 110
5.3 ARM926EJ-S virtual platform architecture with QBox 111
5.4 SystemC and QBox local times 112
5.5 Dhrystone runtime on four cores 113
5.6 Architecture of the system 114
5.7 Architecture of the node 114
5.8 Architecture of the gateway 115
5.9 Configuration of models in the gateway 115
5.10 System model based on different threads 117
5.11 Quantum impact on Linux boot (gateway only) 119
5.12 Impact of the quantum and the the number of nodes on the boot time 119
5.13 Execution time of the simulation in function of the number of nodes - IO intensive . 123
5.14 Execution time of the simulation in function of the number of nodes - CPU intensive 124
5.15 Quantum Keeper Plus callback mechanism 124
5.16 Quantum impact on Linux boot with Quantum Keeper Plus 125
5.17 Summary of contributions through virtual platforms 127

18 TLM and software protocols 140
19 Software protocol(s) connected to another software protocol(s) 141

List of Tables

***

2.1 1 billion write loop 51
2.2 1 billion read loop 52
2.3 1 billion TLM transactions with write 52
2.4 CCI untyped pre write callback 10 millions loop 53

3.1 non memory mapped protocols in regard of TLM 70

4.1 Comparison of asynchronous mechanisms (without synchronization) 107
4.2 Comparison of asynchronous mechanisms (with synchronization) 107

5.1 Comparison of state tracking between SystemC trace and CCI parameters 121
5.2 Impact of the node CPU frequency and the number of nodes. VP = Valid Packets, ET = Execution Time (s) 122
5.3 Evaluation of the improved TLM standard with UART 123

Acronyms

***

AADL Architecture Analysis and Design Language. 11

AHB Advanced High-performance Bus. 26, 58, 65

AMBA Advanced Microcontroller Bus Architecture. 25, 26, 58

AMP Asymmetric MultiProcessing. 9

AMS Analog/Mixed-Signal. 64

APB Advanced Peripheral Bus. 12, 26, 58, 65

API Application Programming Interface. 17, 35, 38–46, 55, 80, 82, 90, 92, 102, 111, 116, 120, 124, 136

ASB Advanced System Bus. 26

ASI Accellera Systems Initiative. 32

ASIC Application-Specific Integrated Circuit. 6, 11

ASIP Application-Specific Instruction set Processor. 6

ASSP Application-Specific Standard Part. 6, 140

AT Approximately Timed. vii, ix, 18–20, 25, 54, 57, 61, 70, 75, 139, 141, 165

AXI Advanced Extensible Interface. 25, 26, 58, 65

BCA Bus Cycle Accurate. 11

CABA Cycle Accurate Bit Accurate. 11, 63

CAN Controller Area Network. 25, 58, 61, 63–66, 69, 70, 72, 73, 137, 138

CASI Cycle Accurate Simulation Interface. 39

CCI Configuration, Control and Inspection. vii, ix, xvii, xviii, xxi, 2–4, 32–34, 37–48, 50–55, 58, 71, 79–82, 84, 102, 114, 116, 118, 120–122, 124–126, 128, 129, 131, 132, 134–136, 141, 162, 164, 165

CPU Central Processing Unit. vii, ix, 3, 23, 27–29, 54, 60, 86, 87, 90, 96, 100, 105, 110, 111, 113–115, 117, 121–123, 125, 126, 128, 129, 142, 162, 165

CRC Cyclic Redundancy Check. 69, 70, 138

CS Chip Select. 68, 83

DES Discrete Event Simulator. 14–16, 22, 91

DMA Direct Memory Access. 13

Acronyms

DMI Direct Memory Interface. 19, 63, 67, 71, 72, 90, 92

DTB Device Tree Binary. 113, 118

DVCon Design & Verification Conference & Exhibition. 55

EDA Electronic Design Automation. 24, 37, 39, 55

FIFO First Input First Output. 15, 59, 62, 63, 122

GDB GNU Project Debugger. 115

GIC Generic Interrupt Controller. 114, 115, 118

GPIO General Purpose Input/Output. 25, 58, 68, 83, 114, 137, 139–142

GPS Global Positioning System. 6

HDL Hardware Description Language. 14

I2C Inter-Integrated Circuit. 23, 25, 32, 58, 62, 64–66, 68, 69, 72, 76, 137, 140

IC Integrated Circuit. 3, 5, 6

IdO Internet des Objets. ix, 165

IEEE Institute of Electrical and Electronics Engineers. 2, 11, 13, 17, 97, 130

IO Input/Output. 112, 117, 118, 123

IOT Internet Of Things. vii, 4, 7, 110, 125, 129, 165

IP Intellectual Property. 23, 25, 27, 37, 53, 63, 79, 83, 140

IRQ Interrupt ReQuest. 66, 80, 111, 125

ISS Instruction Set Simulator. 87, 89, 90, 92

ITRS International Technology Roadmap for Semiconductors. 9

JIT Just In Time. 111

JSON JavaScript Object Notation. 44, 45, 131

LRM Language Reference Manual. 42, 73, 75

LT Loosely Timed. vii, ix, 18–21, 25, 52, 54, 57, 61, 75, 77, 87, 101, 110, 114, 115, 139, 141, 165

LWG Language Working Group. 82, 96, 100, 108, 129

MOSI Master Output Slave Input. 63

MPSoC MultiProcessor System on Chip. 38, 89

MTTCG Multi-Thread Tiny Code Generator. 112, 113, 116

NMMP Non Memory Mapped Protocol. 140

NVIC Nested Vectored Interrupt Controller. 114

NVP Name-Value Pair. 131

OCP Open Core Protocol. 25, 61, 139

OS Operating System. 13, 14, 115, 123

OSCI Open SystemC Initiative. 1, 13

OSI Open Systems Interconnection. 60–62, 66–70, 78, 81, 137, 139

PLL Phase Lock Loop. 38

POC Proof Of Concept. 2, 16, 51, 53, 55, 95, 96, 98, 99, 103, 128, 130

POSIX Portable Operating System Interface. 91, 96, 106, 124

QBox QEMU in a Box. 110–118, 122, 123, 126

QEMU Quick EMUlator. vii, ix, 3, 27, 28, 92, 110–113, 125, 129, 165

QKP Quantum Keeper Plus. 124–126, 138

RAM Random Access Memory. 114, 115, 117, 126

ROM Read-Only Memory. 114, 115, 117, 126

RS-232 Recommended Standard 232. 66, 67, 70, 72, 77

RS-485 Recommended Standard 485. 66, 67, 70, 72

RTL Register Transfer Level. 5, 11, 12, 17, 65, 139–141

SCL Serial Clock Line. 68, 137

SCML SystemC Modeling Library. 39

SDA Serial Data Line. 68

SDRAM Synchronous Dynamic Random Access Memory. 6

SLDL System-Level Design Language. 89

SMP Symmetric MultiProcessing. 9, 89, 112–114, 116

SoC System on Chip. xvii, xviii, 1–4, 6–13, 16, 18, 25, 27–29, 31, 32, 37, 38, 58, 59, 62, 65, 66, 71, 83, 85, 86, 109, 114, 125, 127, 128, 135, 140, 161, 162

SPI Serial Peripheral Interface. 12, 23, 25, 32, 34, 43, 63–69, 71, 72, 75, 79, 83, 114, 118, 129, 139, 140

TCP Transmission Control Protocol. 12, 115

TF Timed Functional. 11

TLM Transaction Level Modeling. vii, ix, xviii, 2–4, 11, 12, 17–23, 25–29, 31, 32, 52, 54, 55, 57, 58, 60–80, 82–84, 86–90, 92, 97, 98, 100–103, 105, 108, 111, 112, 114, 115, 117, 118, 120, 122, 124, 126, 128, 129, 139–142, 161–165

UART Universal Asynchronous Receiver Transmitter. xviii, 12, 13, 23, 25, 26, 32, 34, 58, 62, 63, 65–68, 72–74, 77, 78, 80, 82, 83, 114, 115, 117, 118, 120, 122, 126, 129, 140

UML Unified Modeling Language. 11

USART Universal Synchronous/Asynchronous Receiver Transmitter. 66

USB Universal Serial Bus. 58, 64

UTF UnTimed Functional. 11, 12

UVM Universal Verification Methodology. 50

VCD Value Change Dump. 120

VHDL VHSIC Hardware Description Language. 14

WG Working Group. 13, 32–34, 38, 40, 55, 131, 139

XML Extensible Markup Language. 24, 37, 38

Introduction

***

Context

Embedded systems are dedicated systems designed for a particular function in which they are embedded. In 2015, the global embedded system market revenue was valued at $ 159 billion [28] and it is expected to reach $ 225 billion by 2021 continuing to grow almost linearly for years. This increase is driven by a growing demand in the automobile industry as well as multi-core technologies. Thus, the number of connected devices will continue to grow and invade our daily life.

In this context, driven by cost reductions and time to market, SoCs has emerged and has quickly become a current solution. More features are integrated into a single chip and so complicates the design both of the hardware block and of the software application. A conventional design flow is presented in Figure 1. Hardware and software development are sequential. The integration is done late in the design. Unfortunately, it increases the risk of a design failure.

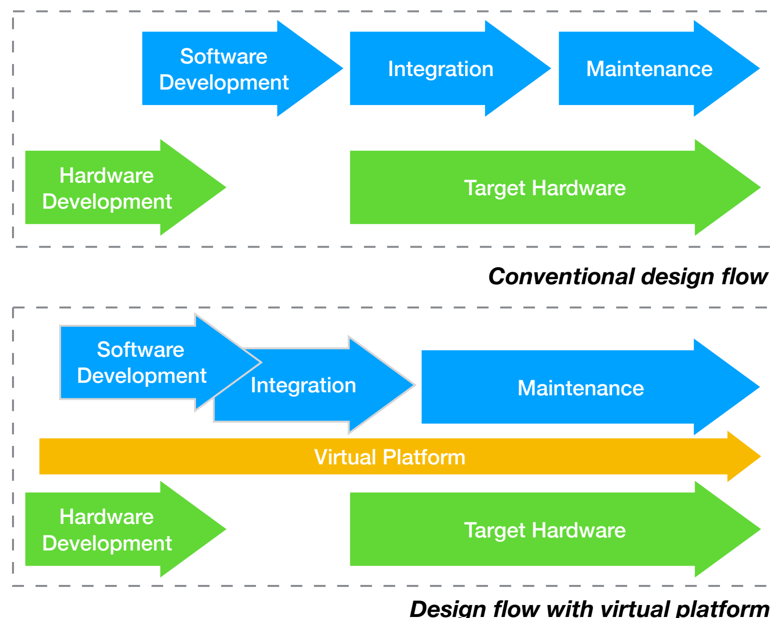


Figure 1 – Conventional design flow and the design flow with a virtual platform [181]

There has been significant research into improving the design flow of SoCs over the last few decades. In 1997, Synopsys and the University of Irvine published a paper [104] that detailed an efficient way to model hardware architecture. Then, along with other companies, a new language for modelling hardware emerged: SystemC [5]. Initially the added value of SystemC was a faster system level design and a better interoperability. Finally, in 2000, the Open SystemC Initiative (OSCI) group was formed to provide an industry neutral organization to host SystemC activities

and ensure the standardization and the adoption of the language.

In 2005, a transport layer standard has been introduced. It is called TLM. It aims to reduce details of communication between hardware models. However, this solution has shortcomings with respect to the modeling of memory-mapped buses and other on-chip communication networks. That's why, in 2009, the second iteration of TLM, also called the interoperability layer, has been released. It was initially based on [96]. TLM-2.0 standard has been presented with two abstraction levels. It contains more possibilities but also more complexity in the standard and its usage. *The true story is that two designers behind each abstraction level do not converge for a single solution. They finally decided in a bar around beers to have both levels.* This last standard finally ended to the introduction of virtual platforms. They aim to provide a complete simulation of the hardware like SoC at high execution speed. This evolution helps to improve the conventional design flow as in Figure 1. Software development and integration is done earlier in a global design flow. Moreover, the feedback between system, hardware and software teams is done during the development and not in a sequential manner.

Problematic and contributions

The history of the SystemC/TLM standard shows that it is constantly evolving. The last Institute of Electrical and Electronics Engineers (IEEE) revision of the SystemC/TLM standard was released in 2011. Moreover, a new version of the SystemC Proof Of Concept (POC), that is the basis for future developments of the IEEE standard, has been released this year. As systems incorporate more features, and requirements evolve, the SystemC/TLM simulation standard has had to meet new requirements. While requirements have evolved, SystemC and TLM IEEE standards has stood still with no major evolutions for six years. Consequently, it currently does not meet all the current requirements.

In this context, researches have been carried out to continue to improve the design flow of SoCs over the last decades. Finally, the problematic of this PhD is located in the improvement of standard for virtual platforms through various aspects.

With the increase of models in virtual platforms, the need of configuration has become de-facto a major requirement. In this PhD, an interoperable and standard solution to improve the configuration of virtual platforms is sought. The current TLM-2.0 standard focuses on memory mapped protocols. However, they do not constitute the only protocols that are available in SoCs. In this PhD, a way to improve the current TLM-2.0 standard to better model and also to support non memory mapped protocols in virtual platforms is studied. Next, the increasing number of cores and accelerators inside SoCs, mostly time consuming models, has an impact in term of the simulation execution time. In this PhD, a solution to better take advantage of the host machine that executes the simulation in a standard manner is examined. Finally, the PhD work aims to improve the efficiency of virtual platforms. Toward the above mentioned objective, the contributions of this PhD detailed in the manuscript are:

Together, this thesis aims to improve the efficiency of virtual platforms, the contributions are:

- **Standard configuration solution for SystemC**
  - An interoperable configuration solution
  - An efficient configuration parameter storage
  - An architecture that supports backward compatibility for existing configuration solutions
  - A standard part of CCI

- **Improved version of TLM-2.0 to better support non memory mapped protocols**
  - Definition of a transaction from a TLM point of view
  - A re-factor of TLM-2.0 to better support non memory mapped protocols
  - A proposal to use CCI standard to better handle the meta-data of a transaction
- **Standard mechanism to support asynchronous events**
  - A study of current solutions to speed up SystemC simulation
  - A first mechanism that adds asynchronous events in SystemC in a standard manner
  - A second mechanism that adds asynchronous event behaviour using a channel in SystemC standard
- **Solution for CPU emulation using QEMU**
  - A study on the importance of the quantum value on the virtual platform
  - An improved version of the TLM quantum keeper
  - An embedded version of QEMU doing CPU emulation and acting as a standard SystemC model

These different contributions have been presented in publications that are listed on the Publications page 159.

Thesis breakdown

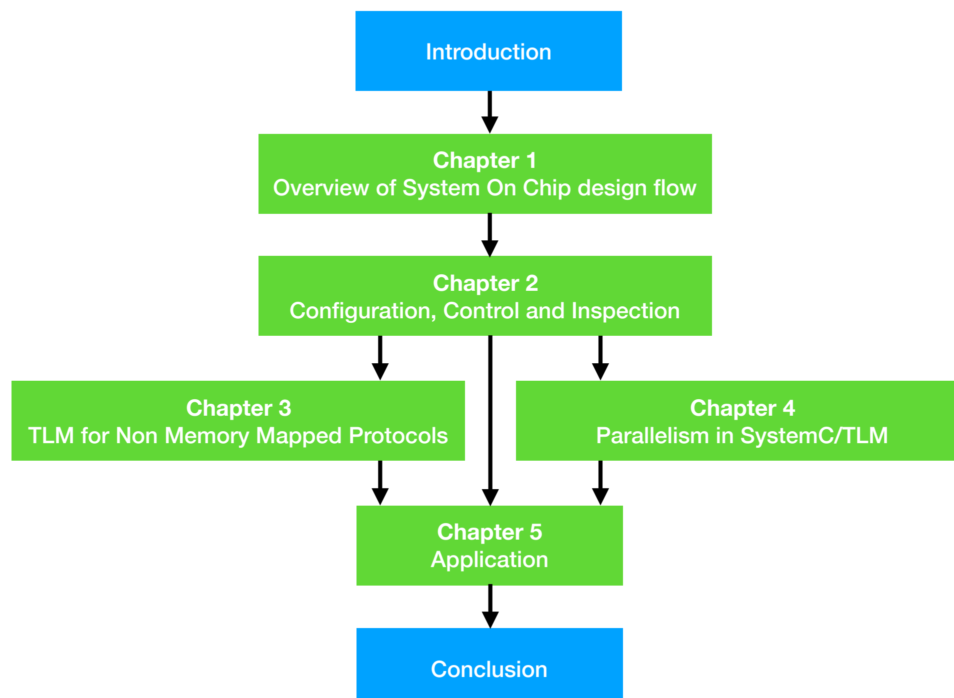


Figure 2 – Thesis breakdown

The thesis is summarized in Figure 2. It gives an overview of the organization of the chapters of the document. The principle five chapters and are described below.

In the 1st chapter, the general SoC design flow is presented. First, an overview of the evolution of Integrated Circuits (IC) into SoCs is given. Then, a SoC design flow is detailed, focusing on modeling and simulation with virtual platforms. Next, the SystemC [5] and TLM standard are presented. Finally, the last part of this chapter presents the different issues faced by designers

using models and simulate current SoCs with virtual platforms. The different problems that this thesis addresses will be identified.

In the 2nd chapter, the needs for simulation configuration features are justified. The related work on configuration and available solutions to configure models and virtual platforms will be examined. It will be shown where the available solutions do not meet all current requirements: a standard solution is then proposed with interoperability, backward compatibility and complete features. Efficiency of this solution is finally evaluated.

In the 3rd chapter, the support of non memory mapped protocols using the TLM-2.0 standard is discussed. Related works on abstract communications are presented. Then, common non memory mapped protocols and how they can be modelled with the TLM-2.0 standard are detailed. Next, a new architecture of the TLM standard is proposed, to better support non memory mapped protocols used both inside and outside SoCs. The CCI standard is used to better handle the meta data of a transaction and improve transaction efficiency.

In the 4th chapter, the simulation speed of multi core SoCs in virtual platforms is discussed. Related works on solutions to speed up virtual platforms are presented. However, no solutions are found that are inter-operable and for the most part the solutions are use case specific. A standard solution to enable the support of asynchronous events is described. A benchmark of this solution is also given.

In the 5th chapter, a concrete application of all the presented solutions from the last three chapters is presented. An Internet Of Things (IOT) platform along with a gateway has been studied. The configuration solution has been used to configure the frequency of modules, the memory map, etc. The TLM standard improvement has been applied for communications between the IOT platform and the gateway. The asynchronous event support has been considered to run platform processors on multiple host threads. Performance of the simulations as a whole was finally measured.

Chapter 1

Overview of System On Chip design flow

***

1.1 Introduction

There is a long history of developments in integrated circuits. The first IC appeared at the end of the 1950's [73][122][77]. They integrate multiple electronic circuits inside a single device, also called a chip. This integration resulted in smaller designs, cheaper solutions and a faster production. They represented a first step in the conception of large and complex electronic devices. Indeed, products was initially only composed of few ICs integrating low complexity functions. Since the conception of ICs, technology has evolved, decreasing the transistor size (from micrometers to nanometers), and enabling the number of transistors that can be integrated to increase dramatically [31][56][91] as illustrated in Figure 1.1.

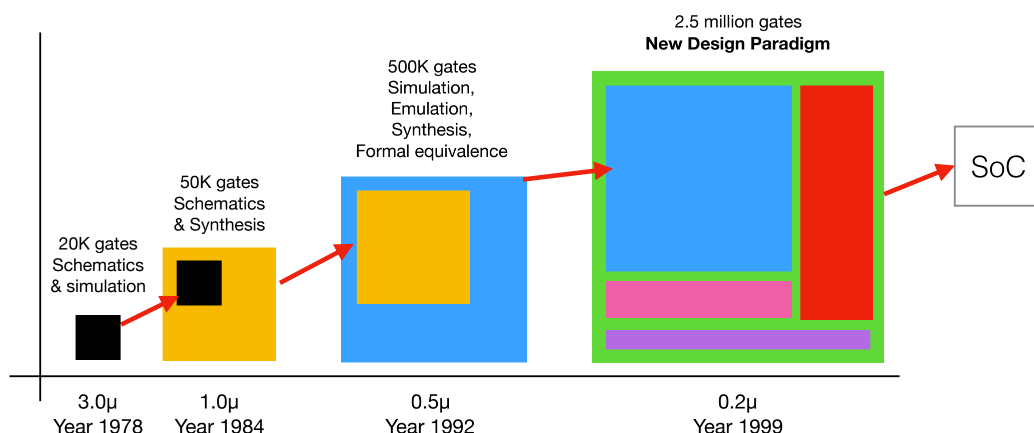


Figure 1.1 – Microelectronic evolution - Yesterday's chip is today's function block! [128]

With the integration of more and more transistors into a single chip, new applications emerged [34][32][167][117]. Naturally, the performance of chips has increased. However, in order to benefit from increasingly higher integration rates, that roughly followed Moore's law [159], the industry has had to solve significant design methodology issues.

Methodologies raise the level of abstraction from transistors to something more abstract, like Register Transfer Level (RTL), using tools to manage the synthesis, logical optimization and place and route. The level of abstraction has been pushed up. It was initially mainly based on formal methods [108][85][130] for the verification and the validation. Logic synthesis [55][103] has been introduced to speed up the development process and reduce the human effort.

However, ICs have continued to grow, integrating an increasingly large number of components and hence removing the latency penalties and reliability issues between ICs. Then, Application-Specific Integrated Circuits (ASIC) and Application-Specific Standard Parts (ASSP) emerged. They enable the design of complex systems composed of multiple circuits, e.g. one processor for software applications and peripherals. They are logic chips designed to perform a function for a specific application. The number of features to be tested and the number of ICs to be integrated has implied methodologies developed previously have become out of date. In order to address the growth in complexity of devices, notable improvements to methodologies are required.

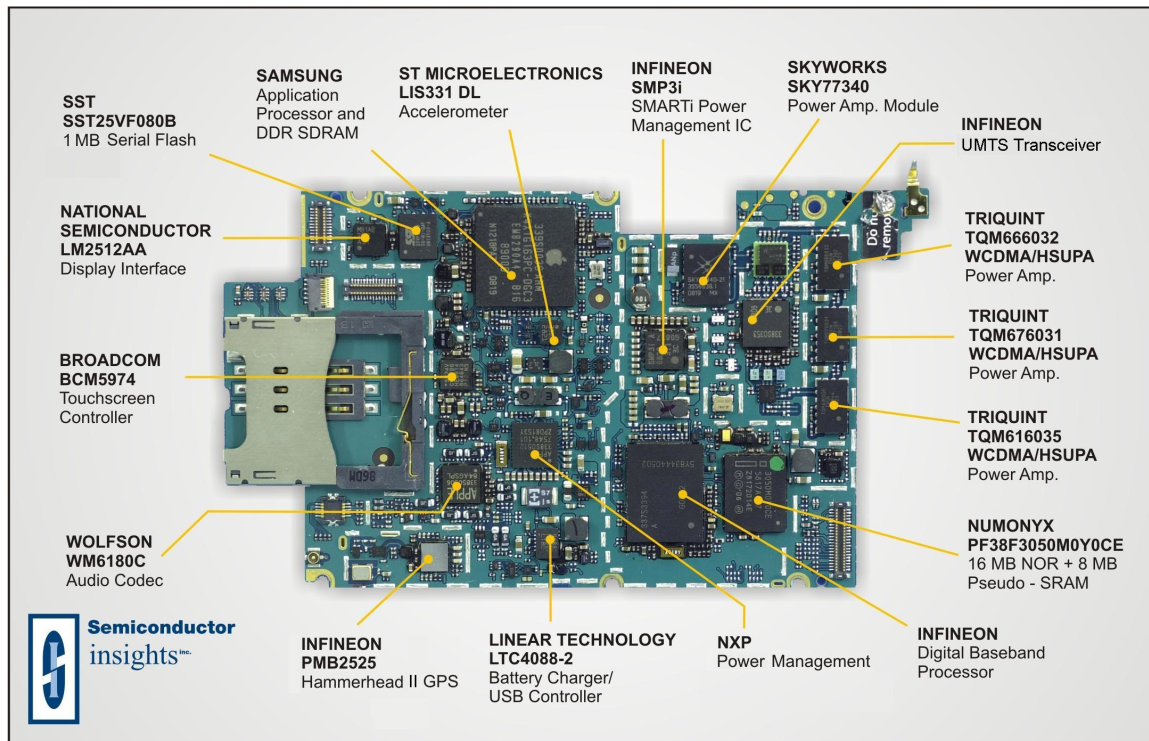


Figure 1.2 – 2008: iPhone 3G mainboard [169]

Nowadays, complex ICs are generally referred to as a System on Chip (SoC). The term SoC has been introduced by [146]. They are the logical evolution of the previous solutions base on communicating between distinct digital circuits. They constitute the integration of ICs and Application-Specific Instruction set Processor (ASIP) as functions inside a single chip. SoCs can be ASICs or ASSPs. Inside, all the components that are necessary for a complete system are grouped. The first SoCs were composed of millions of transistors, nowadays it is possible to integrate nearly twenty billion transistors on the die [193]. They enable a new era, mixing both complex hardware and software components in the same product [47].

In the first decade of the second millenium, things moved still faster. Previously, products had one feature. Now, increasingly more circuit features are included in the same design [149][101]. In 2008, one of the first smart-phone that has been presented used ICs and a SoC [84]. Its main board, presented in Figure 1.2, shows the high number of electronic components in a single system. Each component offered a specific feature (e.g. Global Positioning System (GPS), Bluetooth, audio). The SoC chip included the processor and the memory (Synchronous Dynamic Random Access Memory (SDRAM)). Chips came from different vendors and the software part running on the smart-phone was in charge of the orchestration.

Today, there has been a major paradigm shift. Components have moved into the SoC for energy

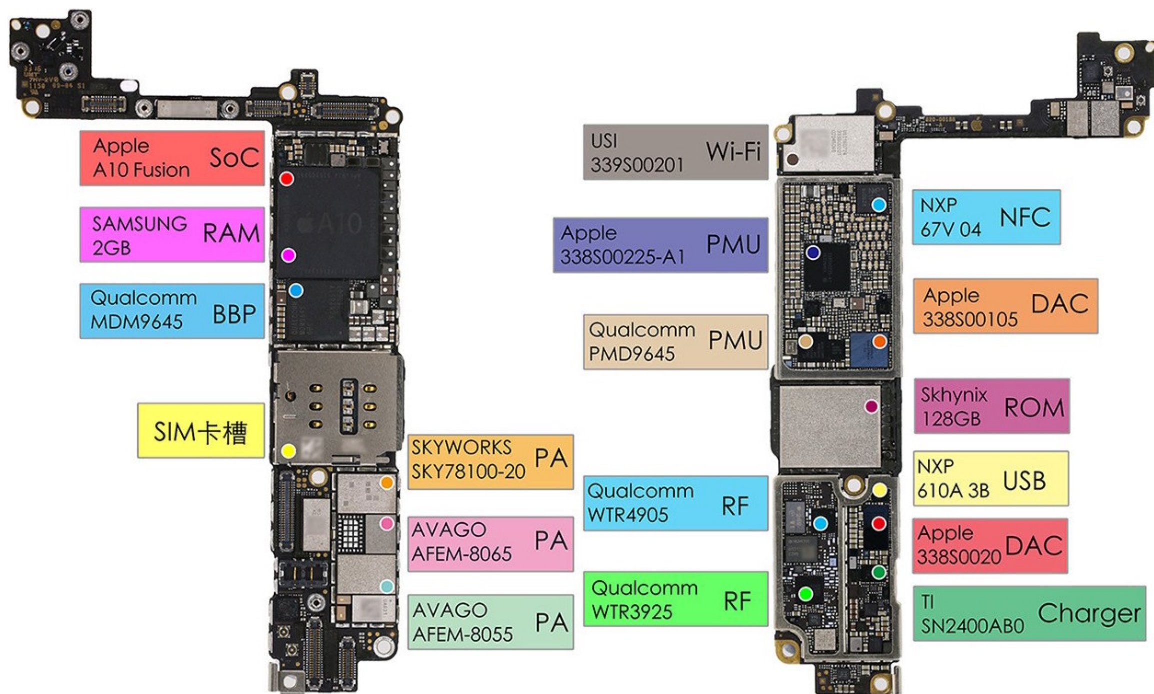


Figure 1.3 – 2016: iPhone 7 mainboard [70]

efficiency and / or cost reasons. More features are integrated in a single complex chip and the size is reduced, as showed in Figure 1.3. The scaling factor between the Figure 1.2 and the Figure 1.3 is about 1.2. The continuous reduction of the engraving technology enabled to go still further for the integration.

In the rest of this chapter, SoC design is presented in the Section 1.2. An overview of design will be presented, including methodologies, system design flow, software development and hardware conception. Then, an overview of the SystemC language, a solution for modeling and specifying software and hardware targets, is provided in Section 1.3. Features of the SystemC language such as timing and synchronization will be discussed. Finally, challenges for platform modeling involved in new system development are given in Section 1.4.

1.2 System on Chip

1.2.1 Introduction

In the context of embedded devices, So called IOT devices are the next generation of devices that will invade our life. The IOT domain will exponentially grow in the coming years [134]. More than 50 billions of IOT devices are expected by 2020 [54]. With technology growth, IOT devices will include more and more features, and become increasingly complex themselves. Specific SoCs for IOT devices have been designed [41][44][191][97]. More features increase the complexity and the number of interfaces used to communicate between components (both hardware and software). They constitute a critical key part of the development. Figure 1.4 shows the evolution of the SoCs cost split with time: SoC size and complexity are increasing in an exponential way. One can note that software development and hardware verification are the most important parts in the total design cost.

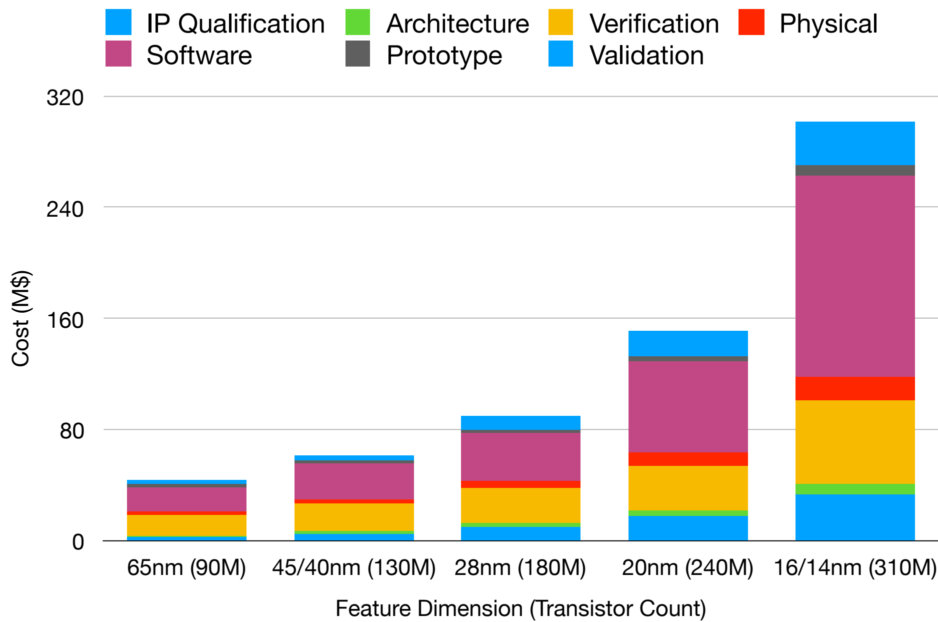


Figure 1.4 – SoC cost evolution [83]

First SoCs developed at the beginning of the 2000's were mainly composed of a processor core and its peripherals. They provided interfaces to allow software parts to communicate with hardware parts, commonly through registers. As the hardware complexity increases, likewise the interfaces, and software complexity increases too. Both hardware and software are clearly highly coupled. Methodologies that do not address the connectivity between hardware and software risk an increased number of issues as the hardware and software are integrated.

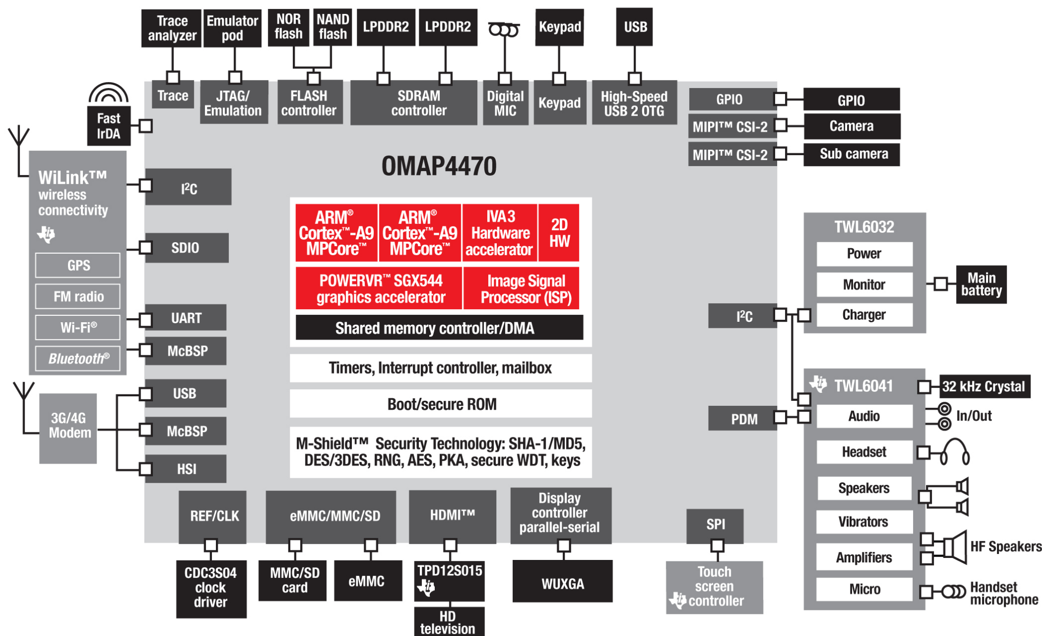


Figure 1.5 – OMAP 4470 SoC [180]

With the increasing computational demands of software running on SoCs, SoCs embed multiple cores in order to improve performance [78][152][203][43][175]. Multi-core processing has been established as a solution of choice in number of electronic industries. A SoC can include a

Symmetric MultiProcessing (SMP) system architecture where two or more identical processors share the same memory space. Figure 1.5 describes a SoC that contains two ARM Cortex-A9. However, a SoC can also include different architectures and so embed an Asymmetric MultiProcessing (AMP) system. If core architectures are different, this family of systems is called a heterogeneous system [114][142]. A powerful core can be used for intensive tasks while a lighter core can be chosen to perform smaller tasks (requiring less power).

In the industry, system cost is a major clear concern. Indeed, errors during SoCs design can be the cause of higher costs and could be reported to the ended price, by the customers or reducing the margins. Moreover, not only the cost, the time can be a brake to the sell if it does not meet requirements with the current time frame. A SoC could be in this case outdated before its sale. Finally, the SoC era requires robust methodologies in order to meet strong requirements. This concern is a clear challenge for the SoC industry.

1.2.2 Methodology

In order to meet new challenges of increasingly large and complex SoCs, different methodologies has been developed as described in [24]. According to the International Technology Roadmap for Semiconductors (ITRS) [90], a ten fold productivity increase will be required by 2020. Indeed, time to market pressure requires the reduction of development time from product idea to the hardware platform and the integrated software applications. Modeling and simulation methodologies are listed as especially important. System exploration and functional verification performed before hardware and software design should help to save development time. Design time is highly dependent of the complexity, design size, and requirements. SoC development can be based on the standard V-model [110][171] as presented on Figure 1.6. It is an improved version of the classic cascade model [15]. At each stage, the current process is monitored to ensure it is possible to move to the next level. With this model, tests begin at the stage of requirements writing. Each subsequent stage provides its own level of test coverage. Here, the development process is represented by a descending sequence in the left part of the V, and the testing stage by an ascending sequence.

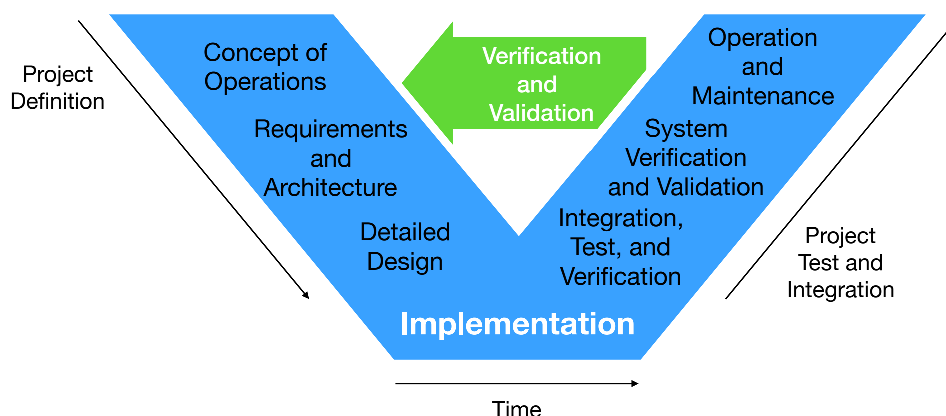


Figure 1.6 – SoC V-Model [196]

During the development of a SoC, requirements, architecture and detailed design of the hardware steps can be done in parallel with the software, without interaction. Once a complete hardware implementation is finally available the integration step starts, and only from this step, software is for the first time executed on the hardware. A software issue can be fixed without increasing too

much the final SoC design cost. However, a new SoC production due to a hardware error can drastically move spending upwards. This problem can happen more often as the chip becomes more and more complex as it increases the number of potential errors. However, another clear concern is the source of the issue. While an issue can be found, its origin (software or hardware) can be unknown until more investigation. It constitutes another time consuming task.

Finally, the V-model has some cons. This model is insufficiently flexible. It does not enable to secure the requirements early. It means new methodologies are required to ensure the SoC development earlier. It includes the hardware but also the coherence with the software part, and more precisely the integration between both worlds.

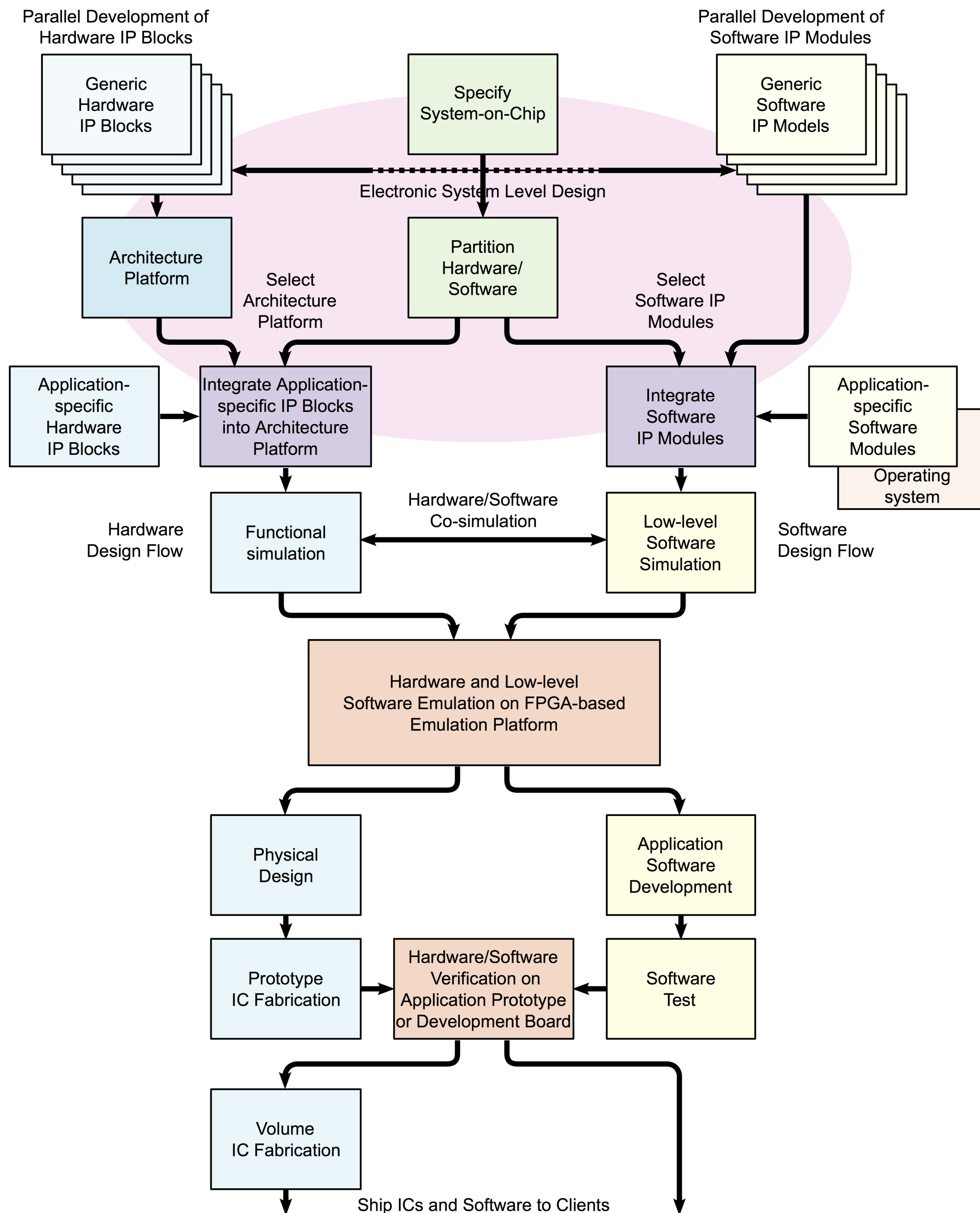


Figure 1.7 – SoC design flow [195]

1.2.3 Modelling and simulation

In order to improve the design flow, many methodologies that help to speed up development time from MATLAB [111] algorithm to ASIC [186][183][100][200][30] have been presented during the last decade. Different methodologies can be applied at different steps during the design of the hardware or the software parts as showed in Figure 1.7. As a result, there are some solutions to manage SoC development. Indeed, after years of industrial attempts and research activities, system design environments are now stable and mature enough to be applied in production [156]. These environments integrate tools for co-design, co-verification and refinement for software and hardware [111][121].

Usually, a complex design begins with the development of an algorithm called a gold model. From a SoC point of view, this sort of algorithm abstracts the actual hardware and software interface. Thanks to methodologies, it is then possible to progressively refine module descriptions at different abstraction levels. Different abstraction levels and their names are described below.

- UnTimed Functional (UTF): Modelling behaviour and communication interfaces of a system without any notion of time but only the order of events. Each event has a null time. Time is not relevant for the functional validation.
- Timed Functional (TF): Same as UTF except TF adds the notion of time. This notion can be used for process execution time, latency... This level is useful for temporal performance estimations.
- Bus Cycle Accurate (BCA): More accurate modelling than TF level adding a refinement of the transactions. Transactions are defined close to the cycle in order to validate time constraints. However, BCA does not add new information in transactions.
- Cycle Accurate Bit Accurate (CABA): In comparison with the BCA level, transactions and signals are defined at the bit level.
- RTL: architectural modelling of the entire system. Each bit, each cycle and each component are defined.

In 1997, a DAC paper [104] written by a tool vendor company called Synopsys introduced the key features to allow the system designer to model hardware blocks in C++ [88]. A software model enables to reproduce the behaviour of a piece of hardware. Models in general are typically used by designers to build early platforms, debug hardware, debug software and to verify the global design, before moving to the more concrete hardware implementation flow, at the RTL level.

Hardware modeling technology has been key to design flow evolution over the last two decades. It is embodied in standards like the SystemC IEEE industrial standard[6]. Other solutions for system modeling language exist like Architecture Analysis and Design Language (AADL) [87], SpecC [51], SysML [132] or Unified Modeling Language (UML) [133] applied to hardware like in [53]. However, SystemC is the principle standard applied as an industrial solution for modelling systems to enable joint hardware and software development.

As it is based on a software language, it eases hardware and software engineers interactions. In that way, SystemC enabled new methodologies [187][40][81], improving the design flow. However, simulation time for data exchange between modules e.g. processor core and memories was slow at a high abstraction level where designers focus on behaviour verification and not cycle / bit accurate properties.TLM-2.0 was introduced to provide a high performance means to simulate at two levels of abstraction. It enables hardware blocks to be modelled with a higher level of abstraction in order to build faster models. Finally, SystemC and TLM facilitate the modelling of a complete SoC that can be used by hardware and software engineers through what is called a virtual platform : a virtual representation of the SoC that reproduces its behaviour.

1.2.4 Virtual platforms

Virtual platform aim to solve various use cases. It enables software development and hardware/-software testing to begin before the real hardware is available and can also be used for later use when the hardware is available. It is an alternative to prototyping on real boards. It expands the software developers productivity allowing them to develop on it as their development platform, months before the real hardware prototype. This solution also enables the validation of both hardware and software interfaces. A virtual platform example is illustrated in Figure 1.8. It is composed of a collection of models which reproduce the behaviour of the real hardware. Models can be defined at different levels of abstraction. The abstraction levels can be mixed between models in the same virtual platform. In that way, some models can be accurate and potentially slow being described at e.g. the RTL level or potentially fast being described at e.g. the UTF level. Models commonly export a similar interface to the real hardware and so provide a reference model (or executable specification) for the software to use during development. For simulation time performance, TLM is commonly used in virtual platforms as detailed in Section 1.3.5.

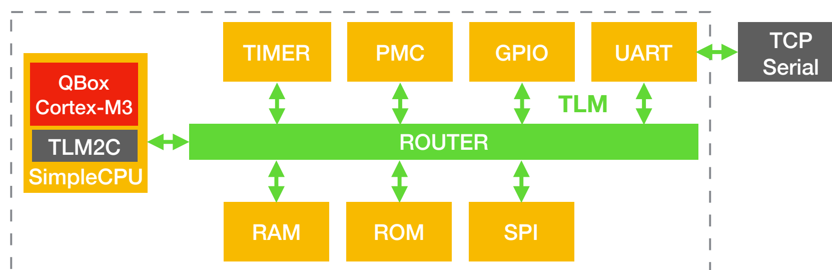


Figure 1.8 – Cortex M3 Virtual Platform

In order to help the software and hardware development, analysis and debug capabilities can be used. Combining these features in a rich set of functionality qualifies a virtual platform as being a truly complete prototyping platform. In Figure 1.8, the SoC is composed of one Cortex-M3 core and multiple peripherals. It includes a model of a UART, Serial Peripheral Interface (SPI), a memory, timers, etc. They are all connected to a system bus. The Advanced Peripheral Bus (APB) UART model is a model implementing a UART interface. On one side, it is connected to the system bus and on the other side it exports a UART interface. The UART interface is then connected to the Transmission Control Protocol (TCP) serial model. It is a special model that enables the designer to interact with the UART in the virtual platform as they would do with a physical device with a 'terminal'. The models in the virtual platform are configurable (setting their addresses, and other configurable parameters).

1.2.5 Software development

With the increasing complexity of hardware in SoCs, the cost of the software in a product has become an undisputed dominant factor in electronics design [83]. That is why software development has fundamentally impacted the development flow in the electronics industry.Software applications rely on hardware interfaces within the SoC to peripherals or accelerators. The development of software that communicate with the hardware, called drivers, is an integral part of the design process [189][197][92]. Thus, hardware designers interact directly with software developers in order to create low level drivers, which in turn affect complex end-user applications.

Software can be provided in different ways. An application can be built directly on the hardware

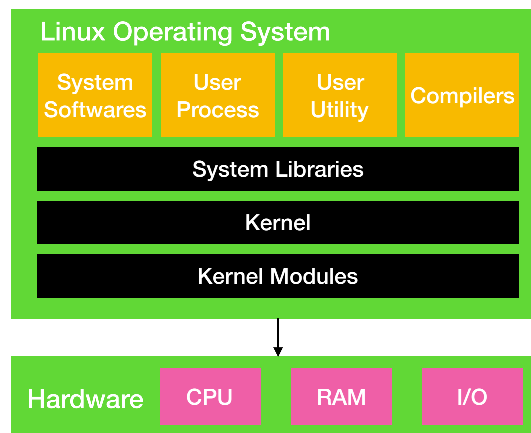


Figure 1.9 – Linux OS layers [182]

(so called 'bare-metal'), or can execute on an Operating System (OS), which itself contains driver, which themselves execute on the 'bare-metal'. OSs are now common in the embedded system world. Applications rely on OS services, relying on drivers, and relying themselves to hardware interfaces of the SoC. As showed in Figure 1.9, the OS is composed of different layers : system libraries and the kernel which is composed of kernel modules. The kernel uses drivers to communicate with the hardware. All of the different parts can be very complex software. These kernel modules can be drivers. The driver is a module which operates or controls a particular type of device that is associated to a hardware module, like the UART peripheral. It communicates with the hardware mainly through registers, or directly through memory which is accessed directly by the hardware (called Direct Memory Access or Direct Memory Access (DMA)). DMA is common for high speed data exchange like ethernet or video devices.

Devices included in SoC influences directly the interface complexity adding maybe more registers or way to use the device. Drivers are a critical part of the system whose bad usage lead to poor performances. The issue is making sure the software can see the device properly, access the registers in a right way and program it in an appropriate way. However, hardware debugging is not as easy as software debugging. The study of the behaviour takes time and the introspection in the real hardware can be limited due to physical constraints. That is where virtual platform are particularly useful. Finally, in order to allow virtual platforms to be used by software during the design and so to decrease common issues during the development, they should expose the same interface as the real hardware.

1.3 Overview of SystemC and TLM

1.3.1 The standard serialization library

SystemC is a C++ serialization library designed to describe system and hardware designs. After the publication of [104], it has been developed during few years by OSCI, an organization whose aim was to promote SystemC. SystemC is now an IEEE 1666 standard managed by the Accellera initiative and a number of Working Group (WG). The library adds notions to the C++ language for modelling and simulation as showed in Figure 1.10. Hardware data types are provided in order to do fixed point or low level bit operations through natural operators which are missing in C++. A

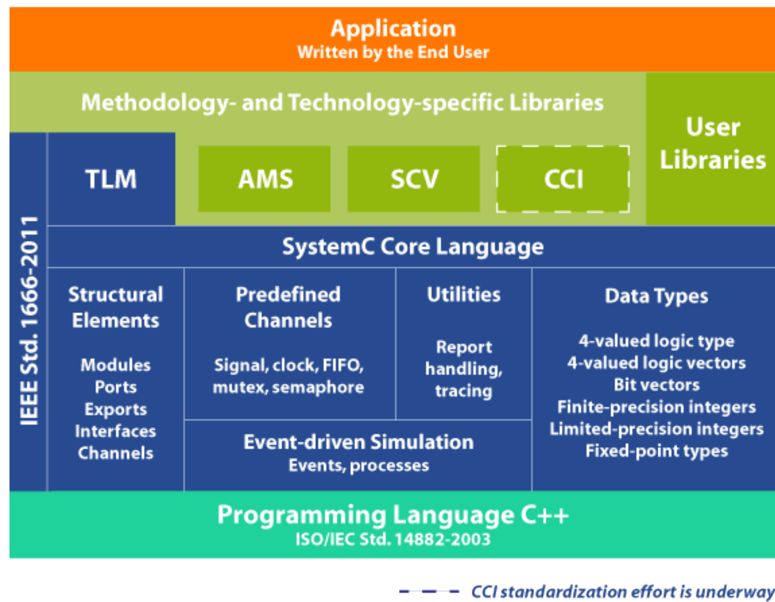


Figure 1.10 – SystemC architecture [2]

discrete time model is used to represent the simulation time. The simulation time is handled by a scheduler based on the Discrete Event Simulator (DES) principle. While OS offers solutions to run processes or threads in parallel, they do not enable to express concurrency as far as processes in VHSIC Hardware Description Language (VHDL). SystemC provides its own concurrency model that is used by the scheduler. Classes to describe models as modules are provided, they enable complex hierarchies of models to be build. Communication and synchronization mechanisms are provided to communicate between modules through interfaces, ports and channels.

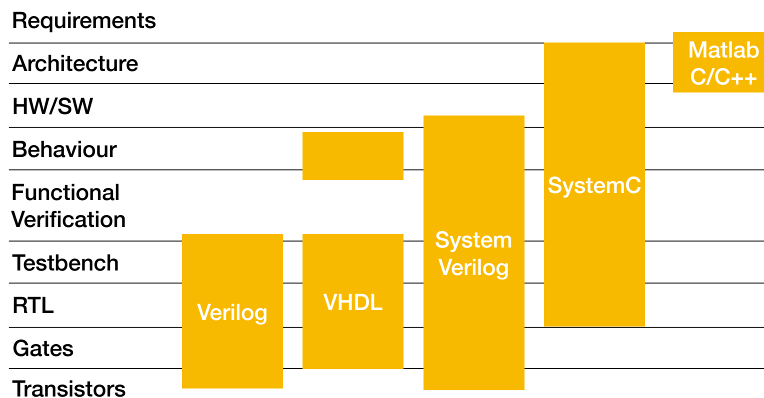


Figure 1.11 – SystemC abstraction position [46]

As shown in Figure 1.11, SystemC covers a broad range of abstractions. It can aptly be described as a system-level modeling language [17] while also be used as a Hardware Description Language (HDL) language.

The SystemC language is based on four kind of basic elements as shown in Figure 1.12. The modules are elements of the system as represented by white rectangles. The behaviour of modules can be described hierarchically through other modules or using specific methods, called SystemC processes. Processes define the logic. SystemC provides two kind of processes : methods (`SC_METHOD`) which are sensitive to an event and run one time or threads (`SC_THREAD`)

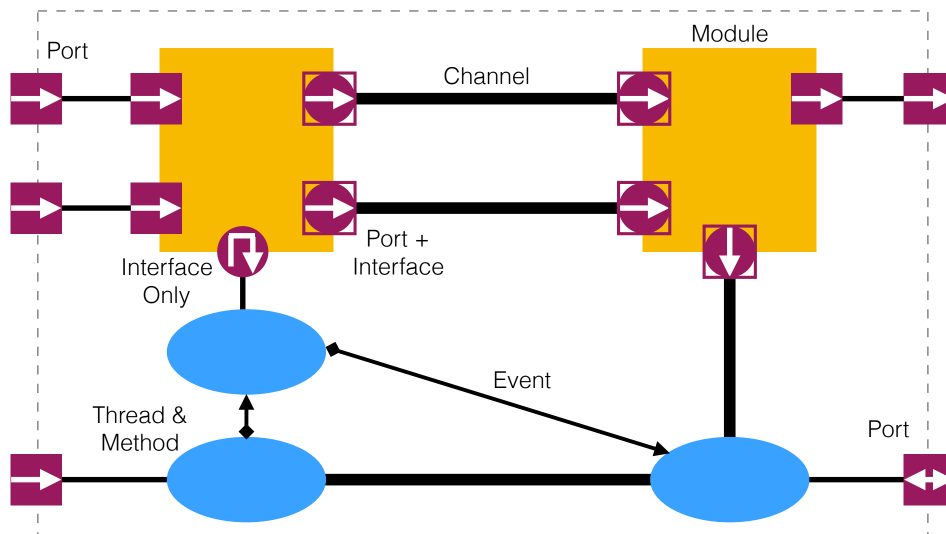


Figure 1.12 – SystemC models [46]

which are standalone and can wait on events. The SystemC kernel enables support for static and dynamic processes, that are added or removed during the simulation. In order to communicate between modules, models can use either ports (with an interface) or direct interfaces. Interfaces define method(s) available in order to communicate. Ports are responsible to forward interface method calls to a channel to which the port is bound. Finally, ports enable the connection to channels through an interface, providing the bridge between ports. Channels can be First Input First Output (FIFO), signals or created by the designer.

The specification of the SystemC modules (ports and the internal behaviour) can be described at different levels of abstraction. Of course, the choice has an impact on the accuracy and on the simulation speed. This choice also has an impact: the interoperability between modules can be an issue. Mixing different abstraction levels inside the same system is complex because of the non-homogeneous interfaces. Finally, the SystemC standard provides the bricks to build modules interfaces but does not provide standardized protocol interfaces. This is one of the main issues addressed by this thesis and detailed in the Chapter 3.

1.3.2 Scheduler

The scheduler included in the simulation kernel of SystemC is a DES and its behaviour of is illustrated in Figure 1.13. The scheduler is responsible for SystemC process execution and data transfer between modules in a coherent way. Its behaviour is as follows. All modules are created and connected through ports and channels during the elaboration step. During this elaboration step, processes that are scheduled to execute are listed. Then, the delta cycle starts, which is a loop around the evaluation, update and delta notification steps. All runnable processes are evaluated once during the evaluation step. Then, if the process running implies a data transfer with another module, signals values and other channels are updated during the update phase. Related processes (e.g. to receive data) are triggered and marked runnable. And so on until no more processes are runnable. It has to be noted that due to DES properties of the scheduler, when there are no more events available, the simulation ends. Moreover, the order in which runnable processes are executed is undefined.

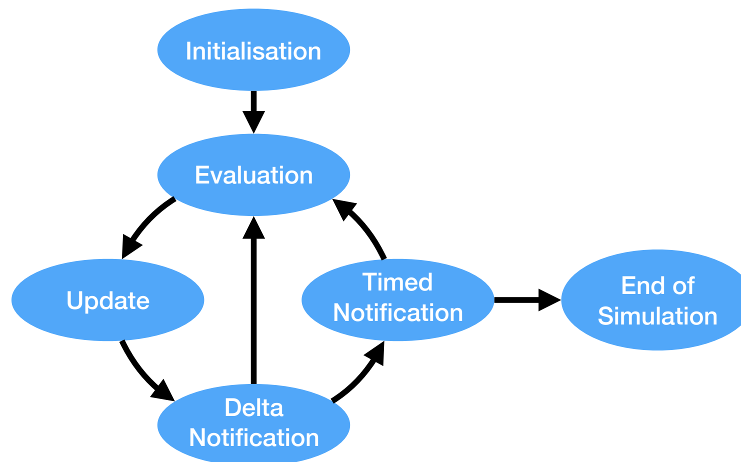


Figure 1.13 – SystemC scheduler

The SystemC scheduler is cooperative and only one SystemC process (SC_THREAD or SC_METHOD) is executed at a time even if many SystemC processes could be executed in parallel. This guarantees data coherency but slows down the simulation. This is increasingly an issue as SoCs become more complex. A more detailed discussion of the research into simulation speed is provided in Chapter 4.

In the SystemC POC, the free implementation provided by Accellera, the SystemC processes are associated with a host thread, which can be either a QuickThread [93] which is a tiny implementation of threads or pthreads [129], a POSIX implementation of threads. However, SystemC POC is only one implementation of the standard but others exist like SystemCASS [105].

1.3.3 Timing

The DES included in SystemC supports untimed and timed simulations. In order to support timed simulations, the SystemC kernel has its own base of time that is uncorrelated to host clock or wall-clock time. To clarify more, some terms and conventions should be defined. Host time is the time on the system, also called the wall-clock time, on which SystemC is running. Guest time, also called simulated time, is the time inside the simulated system. To summarize, guest time is the time that the simulated module(s) see. Guest time and wall-clock time are distinct.

Simulation time can be slower or faster than the wall-clock time. Indeed, simulation time evolves according to events and can advance in a non-constant manner. For instance, a step from 0 to 10ms for the simulation time can take 1s wall-clock time while the step between 10ms and 20ms can take 2 hours wall-clock time. It depends of the simulation speed of events to process between time steps. In Figure 1.14, this difference is illustrated. The evaluation of the processes P1, P2 and then P1 takes a total of 20s wall-clock time while the simulation time only increases by 2 milliseconds. The evaluation of the process P2 and P3 take a total of 15s wall-clock time while the simulation time increases by 13 milliseconds.

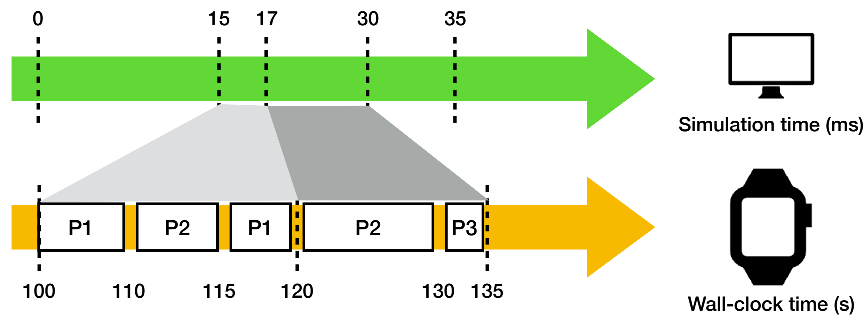


Figure 1.14 – Simulation time vs wall-clock time [113]

1.3.4 Overview of TLM-1.0

To meet the needs for increase simulation speed, new solutions have been required in order to speed up communication within and between models but also to support a change in abstraction. Fast data transfer approaches were proposed in the language beginning with TLM-1.0. TLM-1.0 is a SystemC extension that provides a common solution to model generic data transfer coherently, thanks to a message passing mechanism. It aims to provide a common Application Programming Interface (API) to support various protocols and architectures. TLM also aims to provide model before its definition at the RTL level with simulations time faster than a model described at the RTL level. TLM-1.0 is still part of the IEEE 1666 standard even if TLM-2.0 is now available and detailed in the next section. It remains interoperable with TLM-2.0.

Data transfer with TLM-1.0 happens between an initiator and a target as showed in Figure 1.15. An initiator is a module that acts as the initial sender for a transaction. A target is a module that acts as the final destination for a transaction. The data transferred between both modules is called a payload. A TLM-1.0 `put` is used to put queues data from the initiator to the target. A TLM-1.0 `get` is used to consume data from a target to an initiator. TLM-1.0 did not intend to support a specific range of protocols, the payload of the transaction is not standard, the designer is free to specify its content [177]. This means that a single protocol can be implemented in different ways [144]. The TLM-1.0 standard did not adequately address the issues of interoperability [96].

Figure 1.15 – TLM-1.0 overview

It can be used for architecture exploration and is commonly used in virtual platforms. Indeed, it provides capabilities to connect models together. Blocking data transfers enable the model to invoke the kernel `wait` function, which pauses the calling module waiting for a specific time or event to occur. Non blocking data transfer enables their owner to continue its execution without consuming simulation time. TLM-1.0 unidirectional core interfaces and standard channels for communication are represented in Figure 1.16. The core interfaces inherit from SystemC interfaces while providing a standard common layer of TLM modeling.

TLM-1.0 defines the basic to build transactions but it does not standardize the layout of protocols, letting to the user the free-choice of fields and decreasing the interoperability. Even if TLM-1.0 provides a simple communication mechanism based on SystemC, transactions can be unidirectional

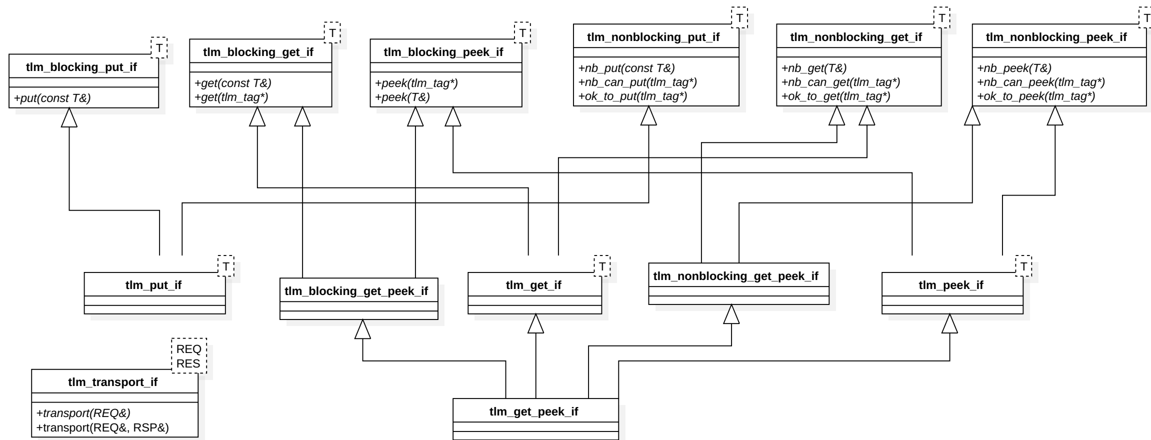


Figure 1.16 – TLM-1.0 UML

and bidirectional. As TLM-1.0 uses channels, data is passed by value. This means there is an initial copy of the data into the channel and subsequently another copy on the other side of the channel. This can have a non negligible cost for big transactions.

Designer can implement their own channels from core interfaces. However, the content of transactions is not standard with TLM-1.0. The issue of simulation performance, level of abstraction, and interoperability were all addressed (with various degrees of success) by TLM-2.0, which is presented in the next section.

1.3.5 Overview of TLM-2.0

While TLM-1.0 aimed to improve simulation time spent in data transfers between simulated models [177] but also a common approach to interfaces, it does not help to resolve model interoperability issues. The standard itself was not badly-formed and could support a large range of protocols, including many interfaces commonly used in SoCs.

However, as the implementation was left to the designer, it can result in incompatible interfaces between two models for the same protocol from different companies. For example, a same TLM-2.0 extension with fields using different names, some combined differently. In order to solve this issue, in 2009, SystemC is delivered with TLM-2.0 [6]. TLM-2.0 aims to provide a fast data exchange layer at a higher abstraction level including a set of interfaces, classes, and solutions to model at a higher abstraction level.

Contrary to TLM-1.0 or SystemC, TLM-2.0 communications between modules can be done directly through function calls without using any channels as showed in Figure 1.17. Transactions are sent by reference in the function calls avoiding data copy. Transactions always happen between an initiator and a target.

TLM-2.0 specify two abstraction levels : AT and LT supported by two types of function: blocking and non blocking, as showed in Figure 1.18. The levels are theoretically inter-operable allowing the designer to mix them in the same design [204].

- AT level allows the designer to continue to add timing point during a communication in order to keep a certain degree of accuracy. It is aimed at architectural exploration with timing points.

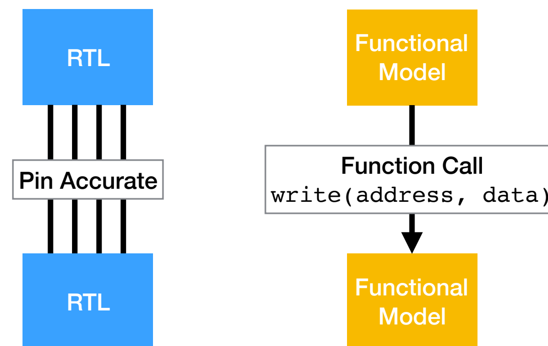


Figure 1.17 – TLM-2.0 behaviour [135]

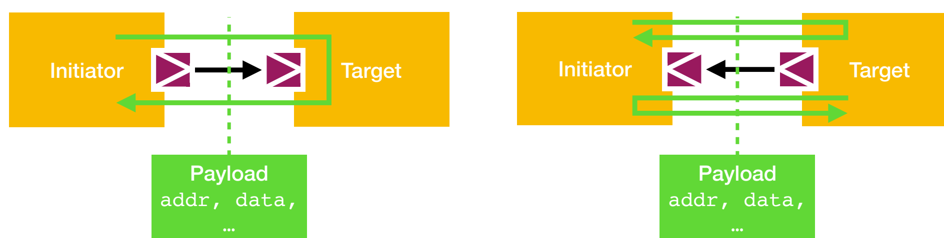


Figure 1.18 – TLM-2.0 blocking and non blocking overview [135]

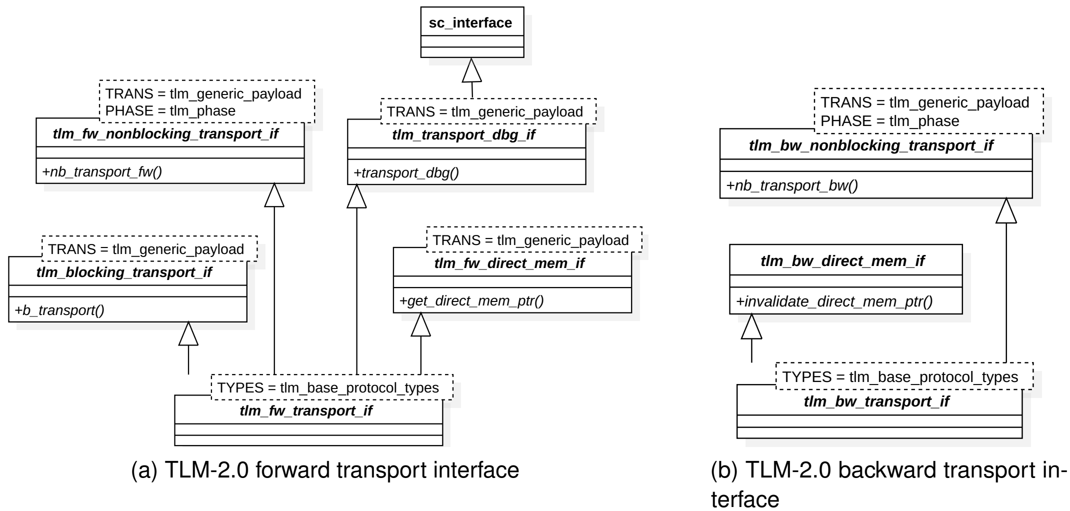
- LT aimed to remove these timing points. It focuses only on the functionality in order to speed up communications during the simulation.

TLM-2.0 focused on modeling memory mapped communication like buses protocols. Fields in a TLM-2.0 transaction are standard. Basically, the payload includes address, data, read/write... fields. The payload in TLM-2.0 is called the generic payload. The TLM-2.0 standard proposes the bus protocol as the most interoperable layer of the standard and leads designers to believe that TLM-2.0 only addresses such buses [60]. Calling the payload "generic" re-enforced this notion. It was expected that protocol owners would build their own protocols. A more detailed review of the work related to TLM-2.0 is given in Chapter 3.

1.3.5.1 Transport

At the core of TLM-2.0 is the transport mechanism. It defines the SystemC interfaces which are used for the communication between SystemC ports and exports. TLM-2.0 introduced six interfaces as showed in Figure 1.19. Each interface has only pure virtual method which need to be implemented. The blocking interface is intended to support the LT abstraction level. The non-blocking interface is intended to support the AT level. As the entire protocol transaction is split into multiple interactions between the initiator and the target, two interfaces are defined: one for the forward path and another one for the backward path. In the non-blocking interface, pure virtual methods also contain a phase argument.

The Direct Memory Interface (DMI) interface is also defined, intended to decrease the number of TLM transactions, it is implemented as two interfaces: one to get a direct memory pointer, and another one to allow the target to invalidate the pointer. Finally, a debug interface is available in order to run transactions without any impact, delays, waits, event notifications, or side effects associated with a regular transaction.



(a) TLM-2.0 forward transport interface

(b) TLM-2.0 backward transport interface

Figure 1.19 – TLM-2.0 transport interfaces

1.3.5.2 Socket

SystemC uses ports in order to connect different modules separated by channels. As seen before, channels introduce transaction details that are not necessary at the AT and LT levels. That is why TLM-2.0 enabled communication directly through sockets based on ports and exports (`sc_export`). Sockets are available in two families: initiator and target. The initiator socket, called `tlm_initiator_socket` uses the SystemC interface `tlm_fw_transport_if` as defined above. The target socket, called `tlm_target_socket` uses the SystemC interface `tlm_bw_transport_if`. They are standard sockets. Non hierarchical initiator sockets must be connected to a target sockets. This enables a forward and a backward path through an internal mechanism.

Sockets can also be used to bind hierarchically modules according to the SystemC hierarchy. Sockets are parametrized with the protocol types as discussed below. A protocol is defined by its payload and its phase. The standard also provides *util* classes which can be used to ease the use of sockets in a SystemC module. It includes support for a multi socket, allowing multiple sockets to be bound to the same socket.

1.3.5.3 Payload

The generic payload is a class offered by the standard for transaction objects primarily intended for memory-mapped bus modeling. This class contains the fields that define the payload. It includes the command (read/write), the address, the data, the byte length... All fields do not need to be set in order to build a valid transaction.

The generic payload also includes an extension mechanism. This mechanism enables ignorable fields to be added. In addition, the standard also explicitly allows designers to inherit from the generic payload class to build a more complex payload. If a completely novel payload class has to be defined, the standard does not guarantee the interoperability : it is left to the protocol owner. That is why the standard recommends using the generic payload. However, it does not offer a clear solution to easily extend or define a new payload without re-implement facilities like the

extension mechanism.

1.3.5.4 Phase

The phase is also a class, called `tlm_phase`, and offered by the standard for transaction objects passed through the interfaces. It is also part of the TLM protocol. The standard provides a default phase type that is only available for non-blocking transport. Phases can be represented by an unsigned integer. They represents the phase of the transaction split in different steps.

Phase can be extended in a similar fashion to the payload (with the help of a convenience macro). However, for maximum interoperability, the standard advises to use the default four phases. Similarly to the generic payload, if the phases have to be totally redefined, then it is not possible to easily re-use the mechanism provided to add extensions. Instead, it has to be implemented again from scratch.

1.3.5.5 Timing and quantum

In order to improve simulation speed, TLM-2.0 introduced new time mechanisms to decrease context switching with the SystemC Kernel. Indeed, TLM-2.0 transactions can be sent and received between modules from direct function calls. This avoids expensive context switching with the SystemC kernel. To improve the timing accuracy, transport interfaces can also annotate transactions with a delay. However, in case of TLM LT, another way to manage the time has been introduce: temporal decoupling. This enables SystemC processes to run ahead of simulation time. Running ahead of simulation time, initiator modules must manage their own local time. The time that an initiator could be ahead is limited to a know amount called the quantum. This value is user-configurable.

The Figure 1.20 illustrates the local time evolution of a module based on the quantum. In this case, the quantum is set to 1µs. The initiator sends transactions to the target annotating the delay relative to its local time. When the local time of an initiator reaches the quantum value, the initiator must synchronize with the SystemC simulation time.

The length of the quantum has an effect on simulation performance. The smaller the quantum, in general the more accurate the simulation. The bigger quantum the faster the simulation is as less synchronization with the SystemC kernel happens. While this assumption sounds logical, it will be shown in the Chapter 5 that it is not always the case and the optimal quantum for speed is not always the biggest value. The quantum is defined globally and the standard encourages to use it in all initiators. However, the standard does not necessarily impose the use of it. Instead, it is possible to have for each initiator a different time quantum. In that case, the initiator using the smallest quantum may be the bottleneck of the simulation as it will require more synchronizations. It also means that it is possible to have more synchronizations during one quantum if required in order to improve the accuracy for example.

In order to facilitate the management of the local time in each initiator, the standard provides an utility class called TLM Quantum Keeper. The quantum keeper provides a set of methods for managing and interacting with the time quantum. Each initiator uses an instance of the quantum keeper. It is normally used each time the time has to move. The quantum value is not stored in the quantum keeper. Instead, it uses the global value. Finally, the quantum keeper is not directly part of the initiator module. In that sense, it cannot be retrieved from other modules or external tools

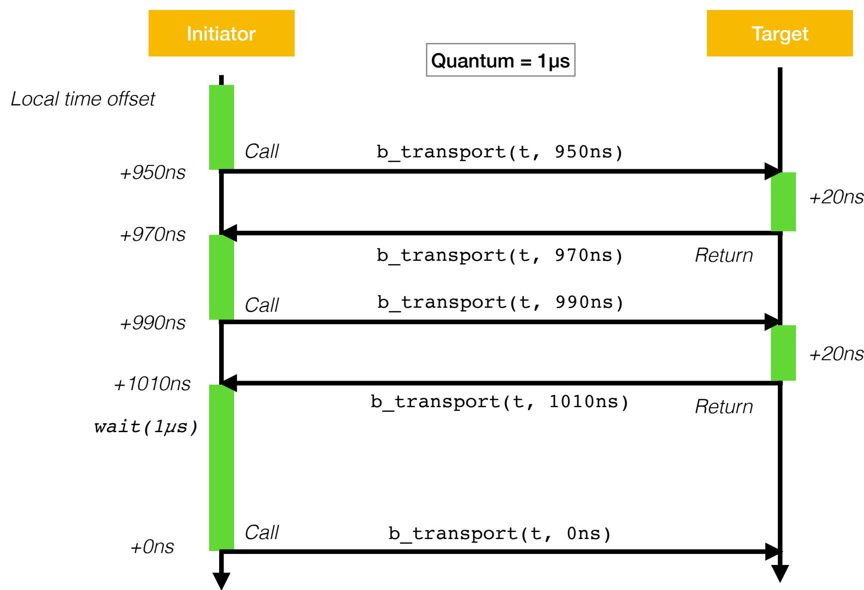


Figure 1.20 – The time quantum [135]

for other new applications. The usage as well as the implementation is left totally to the designer as it is a utility class. It decreases the interoperability. In case of complex virtual platforms with multiple cores and so multiple initiators, it can be useful to tune the quantum with different values in order to debug specific local parts while enabling the maximum available speed for other parts. As the quantum keeper uses only the global quantum value, it cannot be used in that case.

Tuning the quantum is important [74] [72]. An explicit example that shows possible issues is a timer generating periodically interruptions. The timer uses SystemC time as its source of time. If the quantum value is much larger than the period of the timer, then multiple interruptions could be issued at the same simulation time. Indeed, during the synchronization with the SystemC time, interruptions could be generated as a whole to catch the SystemC time. The source code of the timer model will not change if the quantum changes but the generated behaviour can be different. Currently, no metrics are available to measure the impact of a bad quantum in a simulation.

1.3.5.6 Conclusion

SystemC, TLM-1.0 and TLM-2.0 introduced standard solutions to model platforms at high levels of abstraction. Modeling can be built at different abstraction levels and mixed inside the same simulation. Due to its DES properties, the SystemC scheduler suffers in terms of simulation speed in large virtual platforms. On the other hand, TLM-2.0 includes the first inter-operable solution for memory mapped protocols that does not require kernel interaction. The time decoupling with quantum enables new opportunities to decrease context switching with the SystemC Kernel and so further improve the simulation speed, but it requires careful quantum tuning. Finally, SystemC and TLM-2.0 offer standard solutions to meet some virtual platform modeling requirements.

1.4 Challenges for virtual platform modeling

While SystemC and TLM support most commonly required features for modeling a platform, some of them need to be improved. TLM-2.0 targeted memory mapped protocols, but platforms do not exclusively use bus like protocols. Indeed, a platform can include many distinct communication protocols like UART, SPI, Inter-Integrated Circuit (I2C)... as can be seen in Figure 1.21. Platforms also include more and more processor cores and complex Intellectual Property (IP). Both the architectural exploration phase and the debug phase are becoming pressing issues [102]. Virtual platforms are great debug tools which can facilitate this step specifically.

1.4.1 Virtual platform configuration

1.4.1.1 Models and virtual platforms

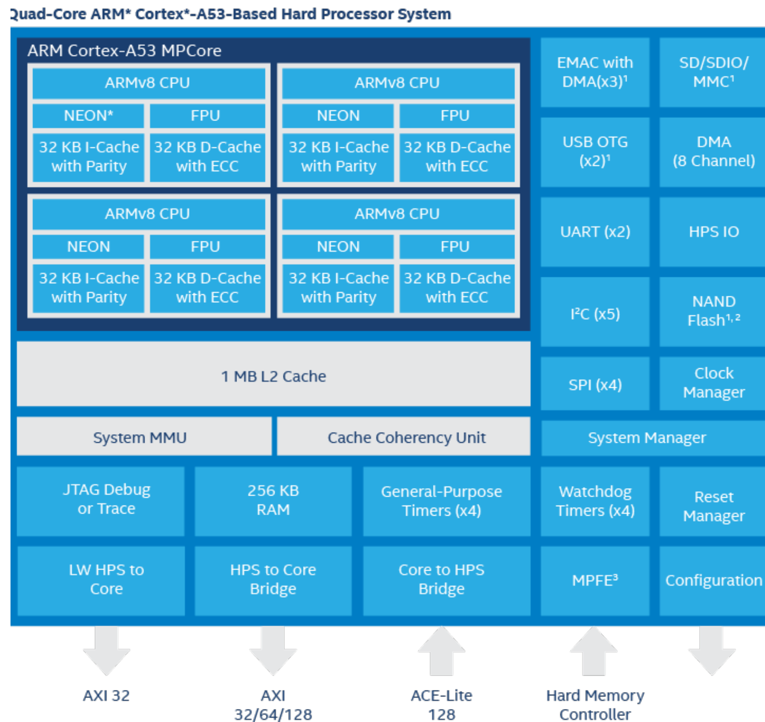


Figure 1.21 – Intel Stratix 10 [86]

A virtual platform helps design space exploration [202][190][166][22]. Models can be just drafts and their functionality partially undefined. Resources allocated in models can be unknown and random values can be employed. Some models can be tuned manually at coarse grain in order to meet the recommendations.

In the Figure 1.21, the complexity of a hardware design can be seen. Each part of the design needs tuning and configuration. Peripherals and CPUs are bridged together through buses which themselves need tuning and configuration. Indeed, in order to exchange information, peripherals are commonly placed on a memory map. Each peripheral has a base address and a size that are used to identify the transactions received. Virtual platforms based on TLM-2.0 may use the quantum. The quantum is a time value and can be configured globally. Chapter 2 will give a

Slave Identifier	Slave Title	Base Address	Size
GPIO0	GPIO0	0xFF708000	4 kB
GPIO1	GPIO1	0xFF709000	4 kB
GPIO2	GPIO2	0xFF70A000	4 kB
L3REGS	L3 Interconnect GPV	0xFF800000	1 MB
NANDDATA	NAND controller data	0xFF900000	1 MB
QSPIDATA	Quad SPI flash data	0xFFA00000	1 MB
USB0	USB0 OTG controller	0xFFB00000	256 kB
NANDREGS	NAND controller	0xFFB80000	64 kB
CAN0	CAN0 controller	0xFFC00000	4 kB
UART0	UART0	0xFFC02000	4 KB
...			

Figure 1.22 – Intel Stratix 10 partial memory map [86]

complete overview of the related work and exact requirements for configuring models, and propose better solutions.

1.4.1.2 Interoperability and tools

As we will see in Chapter 2, configuration is important, it includes the name and value of parameters that should be set when simulation of each module starts. A clear concern with configuration in general is interoperability. A configuration file can be used to tune models. Common configuration formats include Extensible Markup Language (XML) [109], INI [194] or LUA [107]. However, even if the basic structure of the file is standard, the layout for each usage is not. While configuration files can provide values at a certain time in the simulation like the initialization, parameters cannot be set dynamically during the simulation according to some conditions. However, configuration files cannot be ignored. The format and the content of a configuration file will be discussed in the Chapter 2.

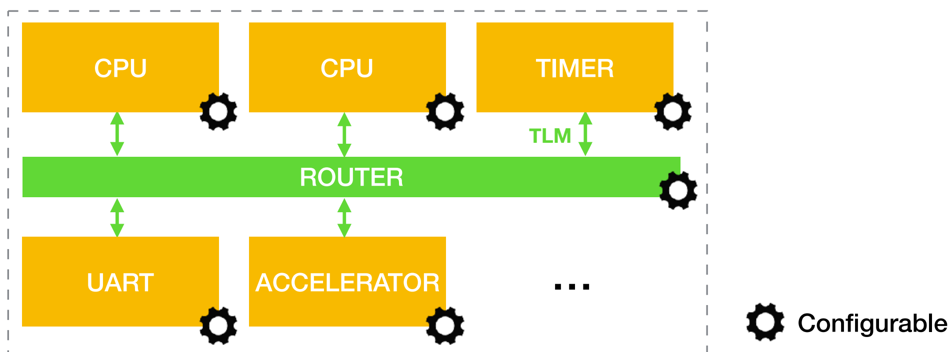


Figure 1.23 – Configurable models

Electronic Design Automation (EDA) vendors provide tools to ease design and validation tasks. Thus, they need to connect to the SystemC models, including support for SystemC based virtual platforms. The connection with the simulation kernel must be done in a standard way in order to have vendor independent solutions. Currently, a solution like [75] is able to manage parameters.

However, this solution is not a standard and totally tool dependent. Furthermore, the high investment companies have made in model IP means that there is considerable inertia to new standards which will involve re-writing that IP. Finally, a standard solution is required in order to solve all these issues. Chapter 2 will present such a solution.

1.4.2 Models of SoC protocols

1.4.2.1 Generic TLM-2.0 like interconnect standard

TLM-2.0 introduced two abstraction levels (AT and LT) that enable the description of memory mapped protocols and fast data transfers. The generic payload has mainly focused on memory mapped protocols. In that sense, the generic payload includes fields for bus transactions. TLM-2.0 supports officially bus protocols and is typically used for buses like Advanced Microcontroller Bus Architecture (AMBA), Advanced Extensible Interface (AXI) as presented in [35] and [162], Open Core Protocol (OCP) for which an official TLM-2.0 kit is provided [7]. The TLM-2.0 generic payload includes most common fields of a bus. However, specific fields of AMBA bus can be embedded inside a TLM-2.0 extension.

Indeed, the aim of the generic payload is to enable interoperability between bus like protocols but all buses are not equal. In that sense, TLM-2.0 extension enables to add a new set of fields transported along with the transaction object. In some way, as extensions are optional, they do not decrease the interoperability. However, the interoperability between extension implementations of a same protocol is an issue. Different vendors can implement different versions of the extension according to their understanding of the protocol. It is part of TLM-2.0 corner limits.

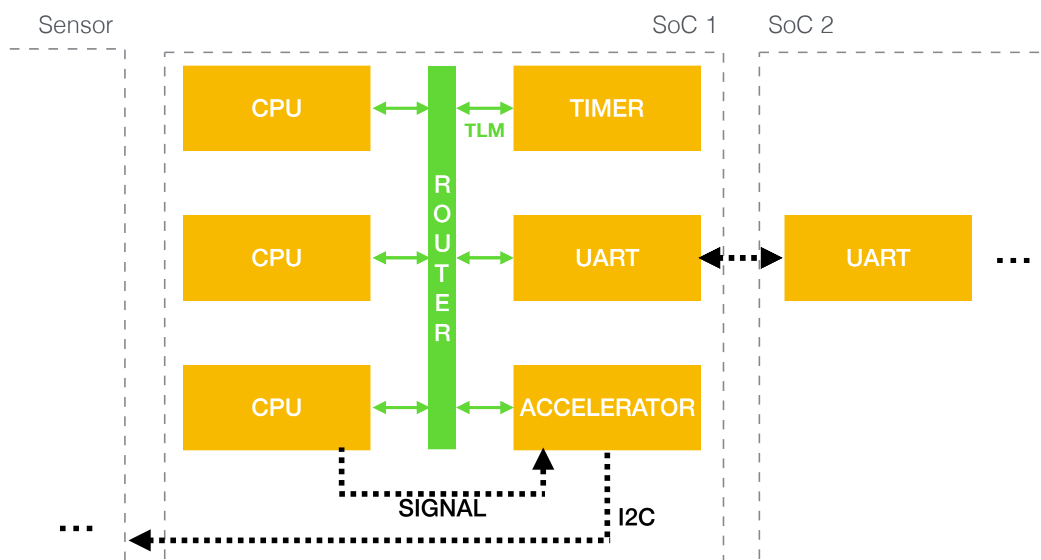


Figure 1.24 – Non memory mapped protocols in SoCs

Buses are highly common in SoCs but they are not the only protocols as showed in Figure 1.21 and 1.24. Non memory mapped protocols include General Purpose Input/Output (GPIO), UART, SPI, I2C or Controller Area Network (CAN). TLM-2.0 transactions happen between two modules, an initiator and a target. However, in platforms, transactions can be exchanged on buses for the connection of more initiators and targets together. Each bus protocol implements its own way to route the transaction from the right initiator until the right target. It can be done, for example, using

broadcast. Modeling broadcast implies more model interactions as all targets must check if they are the real recipient of the transaction. In order to avoid this, TLM-2.0 introduced an interconnect concept. An interconnect is a module that accesses a transaction but does not act as an initiator or a target.

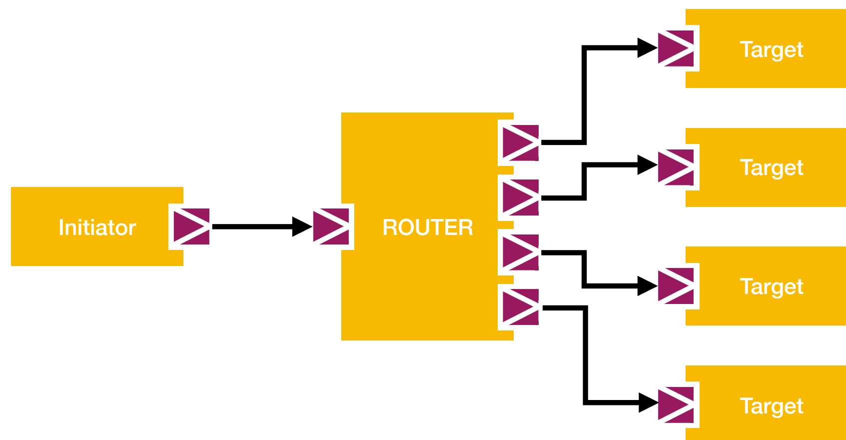


Figure 1.25 – TLM-2.0 router

An interconnect can be used to route transactions from an initiator directly to the right target (the real recipient) as showed in Figure 1.25. Interconnects can be virtual components and do not necessarily reflect the real hardware. They help to improve the simulation speed.

1.4.2.2 Non unidirectional protocols

TLM-2.0 aimed to model buses protocols. However, non unidirectional protocols were not specifically considered. The first AMBA generation of buses protocols included Advanced System Bus (ASB) and APB. Then, Advanced High-performance Bus (AHB) has been introduced for the second generation. It is a high performance bus. Finally, the third generation introduced AXI. AHB supports both master and slave interfaces as well as AXI interfaces. Master and slave interfaces are available with the protocol. The last one enables to have bidirectional exchanges of data simultaneously. None of these protocols support bidirectional exchanges with a same interface. It means that there is no master and slave interface at the same time.

Non memory mapped protocols can include bidirectional interfaces, for example, UART. A device can initiate and receive transactions at the same time. The standard does not provide support for bidirectional sockets which would support this. The only solution to build a bidirectional socket is currently a socket aggregation between the initiator and the target socket. This solution is currently used in [75]. Adding native bidirectional support to TLM-2.0 sockets will trigger limitations like the interoperability with non bidirectional sockets. We will examine how to add bidirectional support, and it's limitations, in Chapter 3.

1.4.3 Virtual platform simulation speed

1.4.3.1 Improve platform simulation speed

Currently, most SoCs include more than one CPU and many complex hardware parts as showed previously in Figure 1.21. From a modeling point of view, the virtual platform can so be composed of many complex models, having a non negligible impact on the simulation speed. That is why, in order to speed up execution of virtual platforms, TLM-2.0 has been introduced. Its aim is to reduce context switching with the simulation kernel during data transfers.

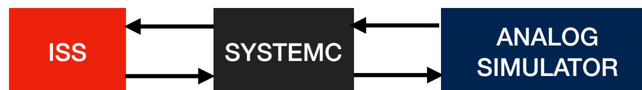


Figure 1.26 – Interconnection of different simulators

Inside virtual platforms, CPUs are typical complex and time consuming models. The increase of the number of CPU model instances can lead to extremely long simulation time. However, while SystemC provides the basic standard brick, CPU model design is left to designers and some design choices can highly impact the simulation speed. While SystemC provides the basic building blocks for models, CPUs and similar complex IP blocks may be better implemented using other technology. Other simulators for CPU emulation are nowadays also available. QEMU and GEM5 [23] are some of the technologies available. Both emulate different CPU architectures at different abstraction levels. As a CPU contains complex parts, it would be useful to be able to reuse these parts directly inside a SystemC simulation as showed in Figure 1.26. While the QEMU has no official support for SystemC, the GEM5 includes naively a wrapper to enable the co-simulation with SystemC [112].

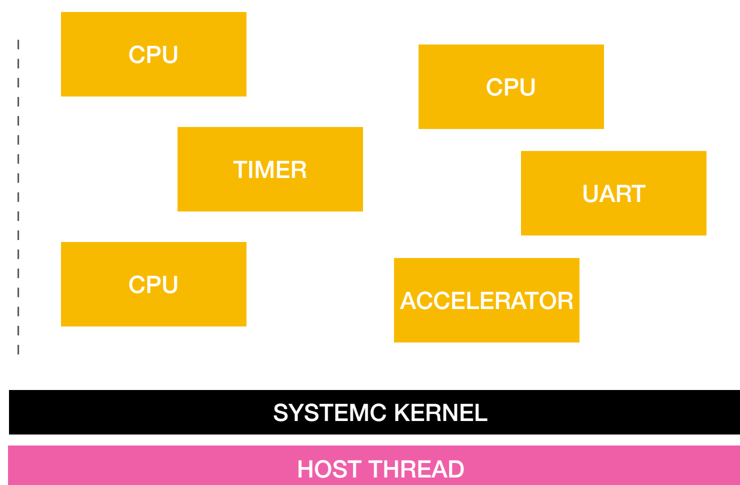
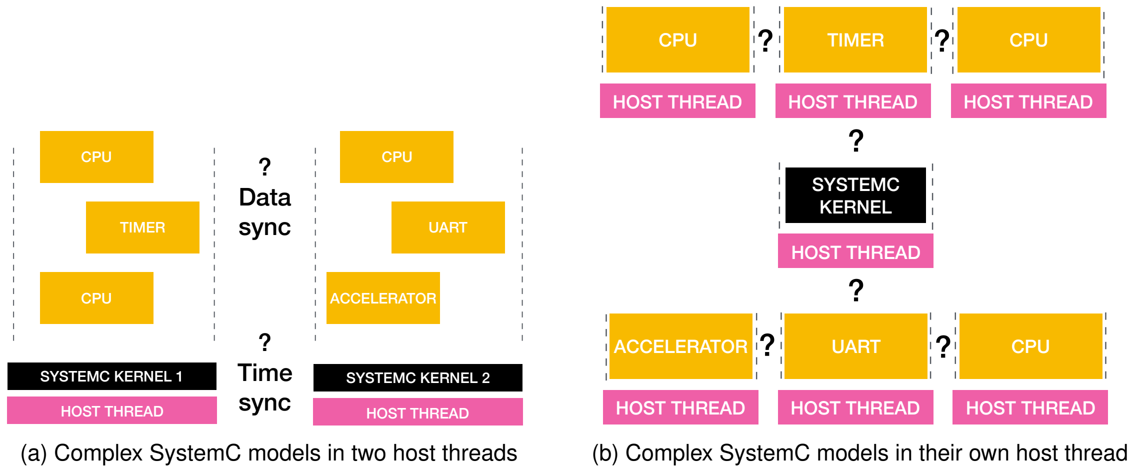


Figure 1.27 – Complex SystemC models in the same host thread

Currently, most improvements in SystemC have been done conserving the mono-thread property of SystemC simulation kernel [19]. Virtual platforms can be summarized as in Figure 1.27. While SoCs include more cores, host computer processor cores are also more numerous. It would be interesting to take advantage of this computation power. More details of related work around the different ways to speed up simulation is given in Chapter 4, which analyzes different available solutions and how the parallelism can be applied. There are major issues with parallelism,

including the data and time coherency between parts (complete models, processes,...) which are executed in different threads, and the interoperability between different solutions as showed in Figure 1.28.



(a) Complex SystemC models in two host threads (b) Complex SystemC models in their own host thread

Figure 1.28 – The parallelism in different forms

1.4.3.2 Simulation speed up requirements

A TLM-2.0 initiator can have its own base of time using the quantum while a TLM-2.0 target does not maintain a local time but instead can consume it. Initiators produce transactions while targets consume and answer them. A CPU, a time consuming model, can be modeled as an initiator and a target, commonly initiating transactions on the system bus, and responding to interrupt requests. In the case of an accelerator, also a time consuming model, it can be modeled as a TLM-2.0 target. Finally, both initiators and targets can be time consuming models. Simulation speed up should be supported by both types of models.

While CPUs could be modelled with SystemC, existing models available for example in QEMU and GEM5 can decrease modeling time. In that case, external simulators support is required in order to use models from other simulators. However, SystemC has its own base of simulation time. External simulators have also their own base of simulation time. A mechanism to synchronize the times between both is required to maintain the consistency. Current approaches will be presented in the Chapter 4 and 5. This also directly enables the support of multiple SystemC simulators. In that case, other SystemC instances are seen as external simulators. Another remaining constraint is the backward compatibility. The industry and academic worlds can be highly discouraged if a complete rewrite of models is required.

1.5 Conclusion

With the evolution of SoCs, new solutions are required in order to meet new requirements during the design process. Modeling and simulation are an answer to help to meet bigger challenges in the SoC cost : hardware, software and verification. SystemC, TLM-1.0 and TLM-2.0 have been introduced as standards for modeling and simulation. While SystemC offers a complete solution

for modeling and simulation, TLM-2.0 aims to speed up data transfers and so the overall simulation but also to higher levels of abstraction.

However, there are issues with the IEEE 1666 standard in terms of supporting modern SoCs important requirements are missed. Configuration solutions for models and virtual platforms are required and their standard integration with tools is currently missing. TLM-2.0 introduced a standard solution for memory mapped protocol but does not officially support non memory mapped protocols although these protocols are commonly used in SoCs. With the multiplication of CPUs in SoCs, simulation speed can drastically decrease due to SystemC scheduler properties. Some solutions to handle parallel CPU simulations are required to meet new challenges. These issues are examined by this thesis and solutions are detailed in the next three chapters. The last chapter will include a concrete application of all previous contributions.

Chapter 2

Configuration, Control and Inspection

***

2.1 Introduction

Nowadays, SoCs can contain dozen of processor cores and hundred of hardware peripherals [137][115]. Due to the increasing complexity, methods and tools for exploration, simulation and reusability during the SoC development are key concerns [90]. With time, SystemC and TLM have become the simulation and modelling standard for both design and verification of systems to help manage this complexity. With the benefit of simulation, the SoC architecture can be modelled and explored with a virtual platform during the design phase. Increasingly, virtual platforms are typically constructed from a number of sub-components, typically sourced from different suppliers. They can contain complex models, themselves composed of models, all of which are configurable, and interconnected as in Figure 2.1. Typical systems are characterized by memory size, software image file name, number of processor cores, bus frequency, etc. In a way, modules are characterized by their parameters. During the design phase it is necessary to control, configure and inspect the entire simulation for exploration, early validation, etc.

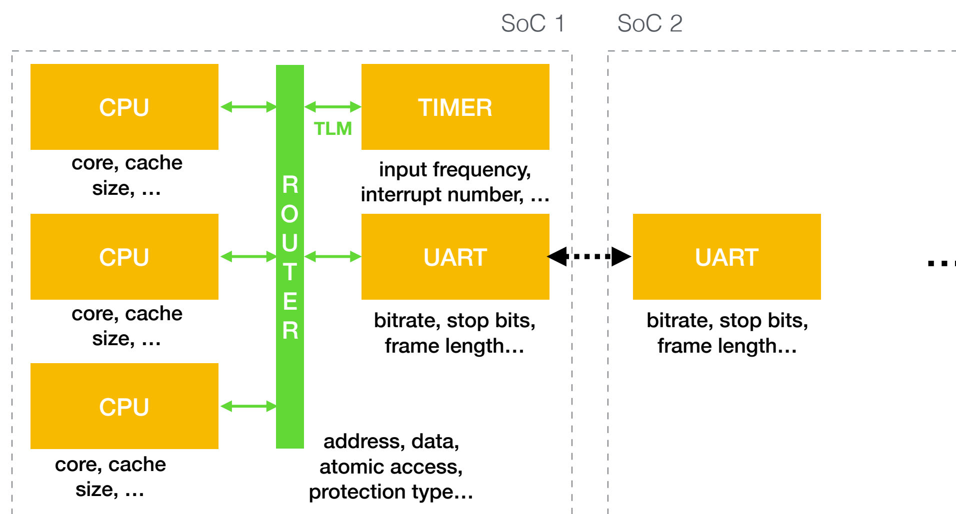


Figure 2.1 – Parameters of SoCs

Inside SoCs, buses are commonly used. They could be modeled with the TLM-2.0 standard as showed on the Figure 2.1. However, TLM-2.0 standard only focused on memory mapped communication. Indeed, even if an interface between different models is somehow the same everywhere, some run-time specific protocol negotiations standards could be missing. Moreover,

the static and binding part of the interconnect is indeed ensured by standard TLM-2.0 interfaces but the associated meta-data is not ensured. Meta-data includes for example the position of a peripheral in the memory-map. While TLM-2.0 addresses communication, it doesn't address the configuration.

Outside SoCs, models can communicate with communication standards like UART, SPI or I2C as showed on the Figure 2.1. All of them have some key features which are configurable, like their bit rate, frame length, stop bits, etc. In order to guarantee that the value sent from one model can be read by another model, the key communication parameters have to be the same. A configuration problem on one side of the communication should lead to a bad data decoding problem on the other side.

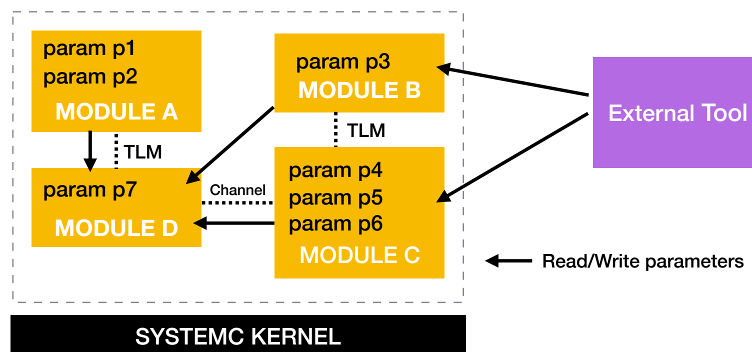


Figure 2.2 – SystemC simulation parameters

Figure 2.2 shows three modules A,B and C that require to read a key parameter from another module: the parameter p7 in the module D. Additionally, an external tool also requires to change parameters of the module B and C in order to try different configurations of the system for the purpose of exploration. In order to allow an external tool but also other modules to read and write parameters inside models from different suppliers, a common solution is required.

In 2009, the CCI WG was created as part of the Accellera Systems Initiative (ASI). It aimed to provide some solutions to issues listed below. Its global objective was to elaborate a standardized answer to issues related to model and tool interactions for configuration, control and inspection. My work and contributions in these fields have been introduced since 2016 into the WG activities. From 2016, the CCI WG aimed to change the requirements according to the change in the industry between 2009 and 2016 in order to elaborate a standard solution. More details and an history of the CCI WG are available in the Appendix A.1.1.

Section 2.2 introduces the needs for simulation configuration features. Next, section 2.3 summarizes the related works in the domain of configuration. Section 2.4 discusses the solution provided by the CCI standard solution. Then, the performance of the CCI solutions is analyzed in the Section 2.5. The breadth of the standard and use cases are discussed in Section 2.6. Section 2.7 examines the limitations of the standard and propose some solutions. Finally, a conclusion of the CCI standard is given in Section 2.8.

2.2 Needs for simulation configuration features

2.2.1 Introduction

Since the first development of virtual platform, an approach for configuration and inspection has been a missing feature as reported in [37]. At the simplest level, a configuration can happen in different places in a SystemC simulation: inside models and globally. Generally, models come from many vendors. They aim is for models to be configurable in order to enhance the model re-use. Figure 2.3 illustrates a typical configurable system. In this case, the system clock speed, the size of the memory, the address and data bus widths, the input clock of the timer, the memory maps are example parameters. When designers combine models together, they expect to configure the complete system in one coherent and consistent place; typically a tool. Until now, it has not been possible, as each model has it is own way of receiving and dealing with configuration information.

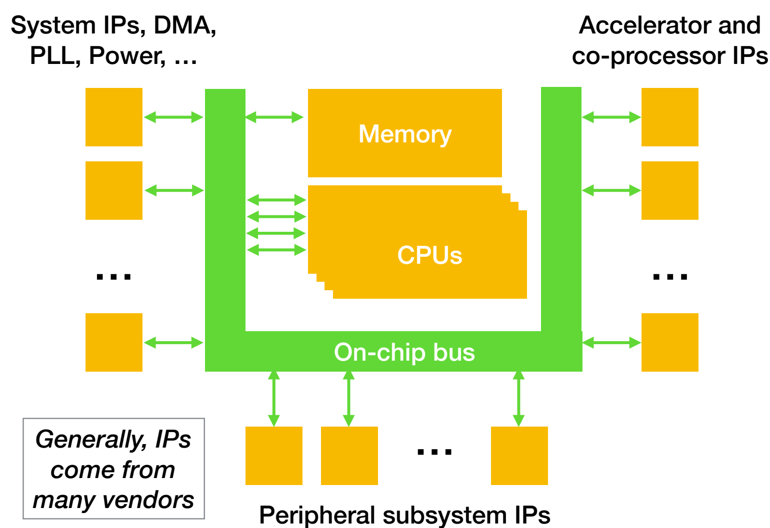


Figure 2.3 – Requirements for a control and inspection standard

2.2.2 Configuration requirements

In order to facilitate configuration and inspection of virtual platforms, SystemC models have to control, check, set and get essential key parameters from different models within the SystemC simulation. Configuration not only avoids model source code modifications, it is also helpful for debug, an essential part of the exploration design process. It also aids protocol negotiation between models from different sources.

During the exploration of virtual platforms, platforms should be stress testable with many different configurations. They should be both in terms of the configuration of models blocks, but also potentially the inclusion of different models blocks. The configuration solution should be usable not only by models of the simulation but also by tools. In that way, a common solution that can be used by tools and models to provide configuration means inspection and control over the entire simulation.

The CCI WG has summarized its initial requirements in [38]. The initial motivation was focused on the instrumentation of models from tools of different companies. However, one of the critical

issues for the parameters in the initial CCI WG was the storage. 6 years ago, the expectation was that storage for a parameters value would have to be provided by a tool. The expectation was that tools would be able to manipulate, interrogate and display parameters dynamically if their storage was held within the tools own structures as defined in [38]. However, the solution should be vendor independent.

As part of the SystemC language, the naming of parameters is also important. The parameters have to respect the SystemC name hierarchy. However, the naming of parameters is more complicated than one would imagine. Models may often have entire collections of parameters which are not physically instantiated in their hierarchy. Their names, and their inherent internal hierarchy may be only understood by the model itself. Nonetheless, parameters should be findable in any level of the simulation hierarchy.

Equally, models may be configured before being delivered to customers (both internal and external to a company). In some cases, it is important that the pre-configuration is captured. But, end users are not presented with that part of the configuration in their tool environment. A notion of privacy is required for some models eager to hide some parts of their design. In the end, the need for a standard configuration solution is not just about a simple standard parameter class. It is about an agreement on what is, and what is not critical for interoperability.

This flexibility allows both the designers and tools to check the validity of parameters. For instance, when SystemC models communicate through a standardized protocol like UART or SPI, they aim to be notified of any change in the protocol configuration on both sides. In that way, a notification mechanism is required to be notified of protocol changes to verify that the protocol configuration is valid.

2.2.3 Without a configuration solution

Let's study the case of the configuration of a timer without a standard solution but trying to answer to the requirements. Figure 2.4 provides a simple timer model that can be used as a processor peripheral. The timer model described in SystemC is parametrized here by two parameters: the input clock frequency and the interrupt number. Different ways to configure the model without configuration feature are presented.

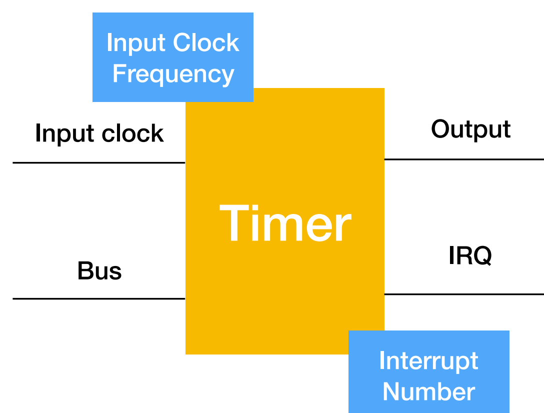


Figure 2.4 – Example of configurable timer

The source code description of a first solution is given in the Listing 2.1. Both parameters are defined as plain types as public class member variables of the SystemC module in order to allow

other models and tools to access to them. However, in order to reach the timer model instance to read parameters, other modules need to retrieve the `sc_object` associated with the module owning the parameters. This solution does not enable tools to list parameters available in the model if they are initially unknown. Class members could have been declared differently in order to meet more of the requirements. The SystemC standard has functionality to enable the attachment of attributes to a `sc_object` called `sc_attribute`. Attributes are composed of a name and a value and the type of the value is user-defined. Attributes can be set and retrieved through a method call on a `sc_object` which limits their scope. However, in contrary to simple class members, `sc_object`s can be found by name with the SystemC API `sc_find_object`. However, the API offered by SystemC attribute is limited. It does not offer a tool connection, neither a notification mechanism, nor tracking of changes or an initial value in case value are not set from another model or tool. In the end, this simple solution meets only basic needs.

Listing 2.1 – Simple example without a (standard) configuration mechanism

```
1  SC_MODULE(Timer) {
2      SC_CTOR(Timer) {
3          // ...
4      }
5  public:
6      double m_input_clock_frequency;
7      unsigned int m_interrupt_number;
8  }
9
10 int sc_main(...) {
11     Timer timer("my_timer");
12     timer.m_input_clock_frequency = 50.00;
13     timer.m_interrupt_number = 7;
14     // ...
15     sc_start();
16 }
```

A second approach consists in the use of preprocessor definitions as presented in [39] and examined below. An approach with the timer is presented on the Listing 2.2. In that case, parametrizable variables can stay privates and the configuration can be centralized in a header. However, a configuration change leads to a new compilation of the entire system. It naturally implies that this solution does not support dynamic modifications at runtime which limits its applicability. Finally, it implies that the SystemC model provider shares the entire source code of their IP in order to allow their customers to modify a simple configuration in their model.

Listing 2.2 – Timer example based on a preprocessor configuration mechanism

```
1  // Configuration.h
2  #define TIMER_INPUT_CLOCK_FREQUENCY 50
3  #define TIMER_INTERRUPT_NUMBER 7
4
5  // Main.cpp
6  #include "Configuration.h"
7
8  SC_MODULE(Timer) {
9      SC_CTOR(Timer):
10         m_input_clock_frequency(TIMER_INPUT_CLOCK_FREQUENCY),
11         m_interrupt_number(TIMER_INTERRUPT_NUMBER) { ... }
```

```
12  private:
13      double m_input_clock_frequency;
14      unsigned int m_interrupt_number;
15  }
16
17  int sc_main(...) {
18      Timer timer("my_timer");
19      // ...
20      sc_start();
21  }
```

A third approach to compensate for the lack of dynamic change support would have been to standardize a file format to carry configuration information of parameters. The configuration file could be a header file with preprocessor definitions. Configuration files can also be used to preload and initialize models with their content. A configuration file approach is presented in the Listing 2.3. However, it was quickly established in [38] that configuration is not just dynamic. It can be driven not just from a tool, but also from other parts of a model as discussed in [16].

Listing 2.3 – Timer example with a configuration file

```
1   // Configuration.xml
2   <?xml version="1.0" encoding="UTF-8"?>
3   <configuration>
4     <module name="my_timer">
5       <parameter name="input_clock_frequency">50</parameter>
6       <parameter name="interrupt_number">7</parameter>
7     </module>
8   </configuration>
9
10  // Main.cpp
11  SC_MODULE(Timer) {
12      SC_CTOR(Timer) {
13          // ...
14          // Load Configuration.xml
15          m_input_clock_frequency = readConfiguration("
              input_clock_frequency");
16          m_interrupt_number = readConfiguration("interrupt_number");
17      }
18  private:
19      double m_input_clock_frequency;
20      unsigned int m_interrupt_number;
21  }
22
23  int sc_main(...) {
24      Timer timer("my_timer");
25      // ...
26      sc_start();
27  }
```

Simple solutions to configuration do exist, but their limits are quickly reached. Moreover, the requirements listed above are difficult to achieve. Nonetheless, SystemC has some interesting parts like the `sc_attribute`. Below we will examine the related works in the configuration domain to see what it is possible to do.

2.3 Related works

2.3.1 Model configuration

The authors compare in [39] different solutions to configure SystemC models according to different requirements. They initially examine how SystemC models can be configured with preprocessor macro combinations or by modifying the source code. They go onto show how models can be dynamically configured with a configuration file or command line parameters. The first approach is less flexible and does not enable the delivery of a configurable model to another designer without providing the complete or the partial source code. The second approach enables the deliver of some models that a customer can then modify.

The solution detailed in [163] enables the configuration of SystemC models. This work is the precursor of the CCI standard presented below, which has been addressed during this PhD. However, this solution missed a default implementation for parameters, also called the parameter storage. Indeed, for instance, a critical part of the precursor of CCI was the belief that EDA companies would provide the parameter implementation. However, as reported in [158],[118] and [75], authors have to use parameter's of their own making. This means that the challenge is to integrate those implementations, rather than relying on the EDA tool to provided one. Nonetheless, the solution in [163] provides a working configuration mechanism implementation with SystemC/TLM available as open source. It provides the support for configuration file and command line parameter support. This implementation constitutes an interesting starting point because it already considers use cases, a mechanism to integrate with other existing solutions.

2.3.2 Virtual platform configuration

Configuration solutions could be used for the adaptation of models but also for the entire virtual platform. In the domain of software for embedded systems, a framework for performance evaluation was detailed in [42]. The authors use XML parameter entities to facilitate the exploration of the architectural design space. This enables more freedom to explore and discover an efficient design. The requirement of a solution to manage complex design exploration is given in the conclusion. Although the presented solution is not compatible with SystemC and TLM, this paper shows a concrete requirement for exploration that could be applied to virtual platforms. A similar approach is given in [99]. The authors have implemented a configuration manager, which is a SystemC module. It allows a designer to simulate complete dynamic reconfiguration scenarios. This solution is use case specific and is not intended to be extended, however this paper also highlighted a need of a solution for the exploration.

In virtual platforms, models can be moved in the memory map during design exploration. The position of models in the memory maps constitute some of the key parameters. A solution to speed up memory map exploration in SoC designs using SystemC models is presented in [16]. SoC memory maps are (typically) fixed in the real hardware architecture. Thus, the addresses of different models cannot be changed during run-time. However, during the prototyping phase, the memory map configuration can be partially unknown. With their solution and during the initialization of the simulation, each model sends information about the address map to the bus arbiter. The latter stores it as an address map in order to forward transactions to the right IP thanks to the transaction address. This solution enables automatic reconfiguration and a faster evaluation. However, this configuration mechanism is specific to memory mapped elements. It is

not intended to be extended for other use cases. Others, like the authors of [123] also use the parameter mechanism introduced in [163]. They use it for the configuration of the virtual platform memory map to store the state of a model in order to save and restore it later. With a global API, models that contain parameters can be controlled from the same place without too much complexity in the models themselves. They noticed that parameters in the simulation introduced a little drawback of 2% on performances. However, it increased the flexibility of the overall platform. In their work, parameters are used for configuration but also for model control.

More globally, a framework for MultiProcessor System on Chip (MPSoC) generation is presented in [36]. An analysis flow with configuration files is used to test different sizes of MPSoC for the same application. Based on various inputs in the first stage of their analysis flow, they generate the complete platform and run tests in order to validate different scenarios. It helps to stress their platform on the same application with different configurations. This paper shows the interest of a configuration solution in order to stress the various combinations of their MPSoC. Another solution, called SoCRocket, is detailed in [166]. It is a virtual platform environment. It shows the application and the flexibility offered by a configuration mechanism. Again, it is based on the solution presented in [163]. Parameters are used to initialize the number of processors and all the related cache models. Values, ranges and descriptions of the parameters are stored in an XML notation. It provides support for LUA[107] configuration files. However, another file format is introduced in [166] as well as in SoCLib [106]. This is further evidence that different file formats are common within the industry. The file format itself should not be the subject of standardization.

2.3.3 Dynamic configuration

Even if some parameters can be set during the initialization phase with a virtual platforms, like the position of peripherals on the memory map, other parameters can be changed during the execution. In the case of a timer model, the input clock frequency parameter can be updated from another model like the Phase Lock Loop (PLL) according to an update of a configuration register in the model. The authors of [158] have demonstrated the benefit of dynamic configuration features in the simulation of complex SoCs. They propose to add attributes to models that can be easily configured. Their parameter implementation provides similar features to the one provided by [163]. However, their implementation adds support for extended attributes. They implemented a mechanism to support:

- user-defined types which can be used for SystemC types like `sc_core::sc_time`,
- a callback system,
- a portable string format using `Boost::Lexical_Cast`,
- a default and initial value

Parameters used as registers are also considered but not used. However no mechanism to track changes is available. In many other ways, the features provided by this implementation are similar to those required by the CCI WG. One point to note is that designers that use this attribute implementation do not want to update all their modules when the CCI standard becomes available. Hence a critical requirement for our work is a mechanism to support vendor-defined parameters.

The solution in [163] includes configuration support for native SystemC types and user types without being specific to a use case like the memory map. It also contains a global registry to track changes and the originator of modifications, default value mechanism, callback notifications and regular expression support for parameter search in the hierarchy. It enables the dynamic change of parameters during the simulation run-time. However, the current implementation is missing a

uniform name mechanism well integrated with SystemC in order to avoid parameter name conflicts and support for user parameter types. It was overly complicated because it supported vendor implementations.

The authors of [118] are waiting for the CCI standard. They write that CCI will provide all the required interfaces for introspection and reflection as required for their framework. Moreover, they built their framework to easily integrate with CCI as soon as possible. The work presented in the article is related to reports of logging data collected from hardware and software. It also considers a debug report in order to ease debugging and support evaluation. However, in order to log parameters from models, a standard is expected in order to ensure interoperability. They have built their reporting framework to be easily extendable in order to integrate the CCI standard when released.

2.3.4 Backward compatibility

As explained before, some incomplete or not interoperable solutions exist for configuration. These solutions cannot be completely deprecated to avoid a major rewrite of models to support the new configuration standard. Rather, backward compatibility should be supported. An interfacing approach with existing configuration solutions is presented in the precursor of CCI. The interfacing with the ARM Cycle Accurate Simulation Interface (CASI) [14] (one solution evaluated in [163]) is based on a callback mechanism. The key idea is to mirror the parameters available in the CASI database with the parameter implementation presented in the paper. This is done by forwarding the requests (read / write) to the original parameter with read and write callbacks. This solution enables an integration of existing parameter systems without the addition of a lot of specific code. However, it has some drawbacks. First, it assumes that no other SystemC models registered callback(s) on these mirrored parameters (which could interfere with the mirroring of callbacks). Second, this solution adds an overhead that triggers a callback when the parameter is used. Third, the API available through the mirrored parameter is limited. For example, extra data associated with a parameter, called meta-data, is not handled. Meta-data can be helpful, for example to know the unit of a parameter. A similar interfacing is provided by the CoWare's SystemC Modeling Library (SCML) [45] properties library. SCML properties are already managed in a database. Instead of adding support to the parameter for a unified registry, a database wrapper has been added along with a new implementation of the SCML properties. Even if the implementation was straightforward, the usage required some modifications in the user model. The header file associated with the implementation have to be updated. Hence, this solution is not ideal.

A lot of efforts was put into building a mechanism where-by the API to the parameter was fixed for the user, and the underlying storage could be provided by a tool. However, since that time, the reality is that the problem of CCI has had to be solved by each model writer. Typically large IP companies have their own (or many) 'CCI like' parameters [16][75][36][154]. Previous solutions are not standard. In the absence of a standard, quick solutions were required. Equally, in many cases, EDA companies have included support for one of more parameter's in their tools. This means that EDA companies are now able to provide the sorts of parameter introspection that is expected whether the parameter's storage is held within the tool or not. [118].

2.3.5 Conclusion

While it is clearly stated in the literature that the use of a simple configuration solution is necessary, no paper address the issue as a whole. As a large number of models with different built-in parameters exist, any proposed solution must take care of interoperability with current solutions. This issue has not been sufficiently addressed. For instance, it is clear that the modification of parameters should not involve the re-writing of the whole platform. The domain of SystemC simulation configuration is not entirely mature. This has lead to custom solutions provided by various vendors and users. Current solutions use the precursor of CCI or other libraries with similar features. Additional solutions added new features. But, there is no interoperability between solutions. What seems universally agreed within the literature is that a complete standard for better interoperability is required.

The significant issue brought up by the literature is not to build a suitable parameter mechanism. The critical issue is to provide a mechanism that enables interoperability across different user solutions. No solution has been proposed to this solve problem, and it is where my research has focused. The literature shows that an instrumentation standard for SystemC that complements TLM-2.0 and enable models from different providers to be fully exploited in any SystemC environment is essential.

Hence, over time, the emphasis has significantly moved from the support of tool specific storage mechanisms to support a vast number of legacy parameter types. One can assert that the original requirement was perhaps miss-guided. But equally 6 years ago there were not as many implementations of parameter like objects. The proliferation of those objects is evidence of the requirement for a standard. It is equally evidence that the standard is very late in materializing. None the less, the change in a fundamental understanding of the 'landscape' in which the proposed standard should operate has, clearly, further delayed the standardization process.

2.4 Configuration, Control and Inspection solution

2.4.1 Introduction

It is necessary to configure, inspect models and be able to change and track values in external tools. These features enable the capture and change of the simulated design during the simulation time. The new features, which are presented below, have been based on results of the precursor of CCI. Figure 2.5 shows an overview of the CCI WG activities. The focus of this work has been to define the parameters and the configuration mechanisms of CCI. The key configuration components of the presented solution are the broker and the parameter.

The broker manages access to parameters that are registered with it and so handles a local database of registered parameters as showed on the Figure 2.6. Its API provides various mechanisms to find parameters in the SystemC hierarchy, add or remove parameters, etc. A simulation has only one global broker, which is the default broker. However, there is another kind of broker. There is the global (public) broker, which is the default broker. Private brokers can also be used, potentially throughout the hierarchy. Both kinds of broker can be used within the same SystemC simulation. Private brokers can be used to hide sets of parameters from a specific model. The reason to have a notion of private broker is also to be able to handle specific configuration files with different file formats as stated in the literature. A specific broker handle is attached to

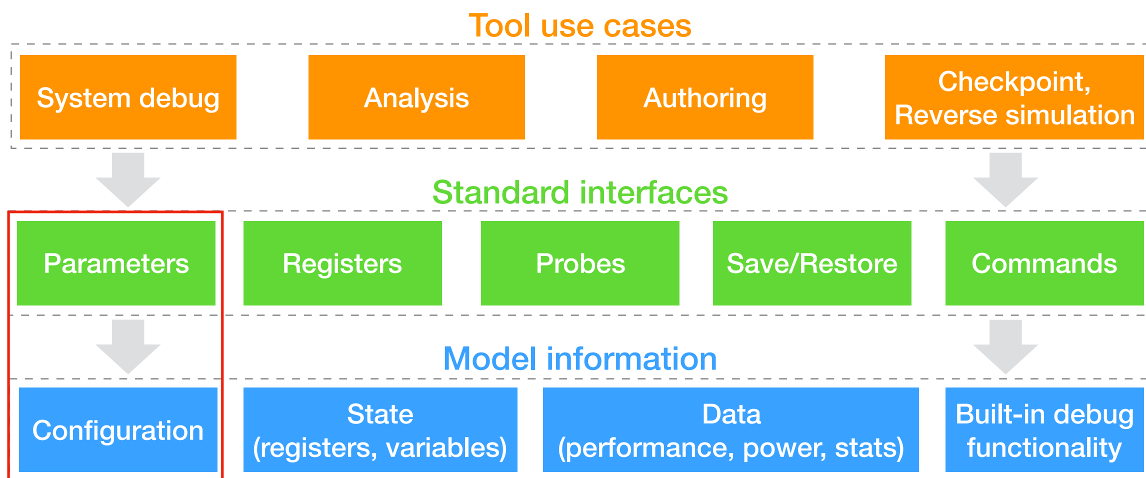


Figure 2.5 – CCI scope and initial focus

each instance of a module. Broker handles proxy the requests to the global (public) broker or a private broker.

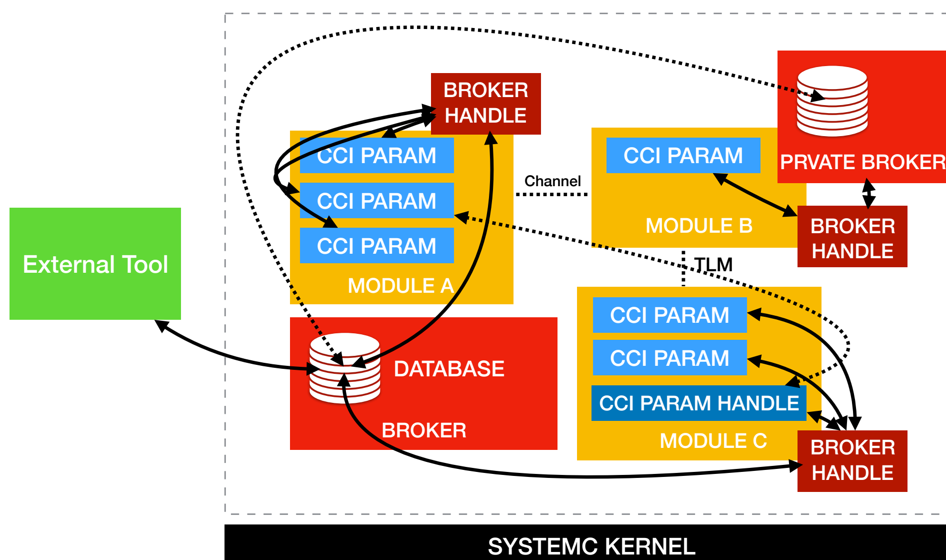


Figure 2.6 – CCI parameters and broker

The parameter is identified by a string name, that is unique in the SystemC hierarchy, and a value. They are declared in a SystemC module with an instance of a CCI parameter. Each parameter is registered to a broker during their construction. The parameter provides an API for type independent setters and getters. They are themselves important to improve interoperability with external tools and different models. The type can remain unknown but a way to interact with parameter is still required. Parameter handles are used to access parameters from other modules. They are requested to the broker handle through the broker API. They proxy the requests to the original parameter. They are used to track who accesses the parameter

2.4.2 Overview

Figure 2.7 provides a more detailed overview of the CCI architecture. It mentions two interfaces : `cci_param_if` and `cci_broker_if`. These API define the standard. The interface of the parameter `cci_param_if` solves the problem of interoperability between user defined parameters and a default implementation. It enables support for previous solution as long as the interface is implemented. Parameters are owned by SystemC modules (also called 'parameter owners'). Their name follows the SystemC hierarchy in which they are instantiated. The CCI precursor introduced a parameter class proxying the request to the tool parameter implementation through a pointer. However, this solution added a level of redirection due to the proxy pattern. It also added complexity that requires a parameter factory mechanism. In the end, the final solution that is presented below fills the hole that the precursor of CCI opened for the interoperability between different parameter implementations.

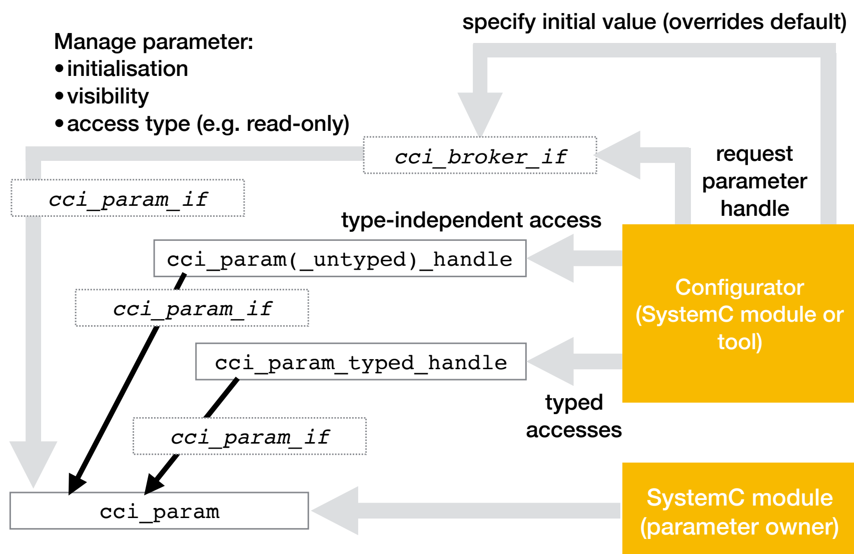


Figure 2.7 – CCI configuration classes and use model

Our solution to ensure interoperability is that the parameter implementation is separated from its interface. Implementing the parameter interface results in complete compatibility with the CCI standard as long as it follows the rules defined in the Language Reference Manual (LRM) [8]. This interface allows existing parameter implementations to be supported. A `cci_param` class is provided by the standard. This class is the default implementation of a parameter. `cci_param` class is templated. This template represents the parameter value type. Initially, this class was not part of the standard and an implementation was required in order to run the standard. However, in order to ease the use and the deployment of the standard, a simple default implementation has been added. This enables the use of `cci_param` in SystemC modules.

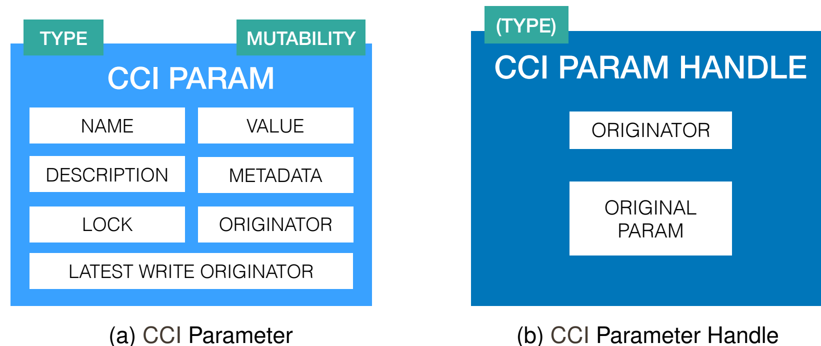
The broker interface is directly used by parameter instantiations and by tools or other SystemC modules that request parameters. On instantiation, a parameter is registered with the broker associated with the SystemC module owning the parameter (the owner). This enables tracking the identity of the module creating the parameter, the model causing value accesses or changes... The identity of a parameter owner is called an originator. It is described in Section 2.4.5.

Other modules or tools can request a parameter handle. A parameter handle shares almost the same API as a CCI parameter interface except that a handle tracks the owner requesting an access to a parameter. This means that the broker returns parameter handles, aliasing

the original parameter declared in the parameter owner. They are returned by the broker in response to a search by name for a parameter. The broker returns parameter handles that are type-independent (not typed) : `cci_param_untyped_handle`. However, if the tool or the module knows the type of the parameter, the type of the parameter can be restored by casting thanks to the `cci_param_typed_handle<T>` class.

2.4.3 Parameter

The key parts of a CCI parameter are summarized in the Figure 2.8a. Basically, a parameter is composed of a name value pair. The value type is specified during the parameter construction through a template parameter. Parameters are not SystemC objects themselves. Instead, their name is registered in the SystemC hierarchy through an API which has been proposed and added in SystemC 2.3.2. The API guarantees that parameter names follow the SystemC name strategy. Thus, their name is not in conflict with other SystemC objects. A parameter handle shares the same features as a parameter excepts it acts as an alias and redirects the methods to the original parameter as showed in Figure 2.8b.



(a) CCI Parameter

(b) CCI Parameter Handle

Figure 2.8 – CCI Parameter and CCI Parameter Handle

A description could also be attached to a parameter or a parameter handle. The description is a free form string defined at construction time and/or subsequently altered with a setter. In addition, a meta-data mechanism has been added. The meta-data field is a string and value pair. It enables model writers to add additional text, explanations of how to use the parameter, units, limits etc.

While parameter values can be modified during the simulation, some parameter values may need to be fixed during their construction and then remain constant later. For instance, the number of available chip selects on a SPI controller. In order to differentiate these parameters, a notion of mutability has been added. The mutability defines the capability of a parameter to be modified during different simulation phases. By default, parameters are mutable. Their value can evolve dynamically during the simulation. The mutability of a parameter is specified by a template parameter during its creation. This means that mutability is a static locking mechanism specified during the build time. It can be used to defined static read-only parameters. However, a dynamic locking mechanism is also required. It is provided in order to change mutability during the simulation. The mechanism can be used to dynamically define parameters as read-only or not. This mechanism is called the lock. A parameter is locked with a password. This password is required in order to unlock it. When a parameter is locked, all attempts to write to the parameter will be rejected. If a selective read-only mechanism is required in order to enable partially read/write to some modules and read-only to others, a callback mechanism could be used as explained in

the Section 2.4.6. To sum up, the mutability and lock mechanisms solve different issues. Mutability is a static mechanism. It is set in the source code as a template parameter. It is also set in the module owner. On the contrary, the lock is a dynamic mechanism which can be modified during the simulation. A parameter can be locked or unlocked from various places in the SystemC hierarchy.

The instantiation of a CCI parameter in a SystemC module is as simple as a C++ class instantiation as shown in the Listing 2.4. Each parameter must be declared with a name and a default value during its construction. The default value is the value that is used by the parameter until a new one is set. This default value is statically defined in the module code as it is defined in the parameter constructor. The default value cannot be changed during the execution.

Listing 2.4 – Timer example with CCI parameters

```
1  SC_MODULE(Timer) {
2      SC_CTOR(Timer):
3          m_input_clock_frequency("input_clock_frequency", 50),
4          m_interrupt_number("interrupt_number", 7) {
5          // ...
6      }
7  private:
8      // ...
9      cci::cci_param<double> m_input_clock_frequency;
10     cci::cci_param<unsigned int> m_interrupt_number;
11 }
```

However, this static default value does not fit all requirements. Indeed, in order to test various combinations of a module, it is unnecessary to be able to change the initial state of parameters. While a hand edit of module to change default value could be done, a more robust solution was required to change the initial state dynamically. An initial value notion has been added. Contrary to the default value which is specified in the constructor, an initial value can set to override the default value. It is done through the broker API. To work, during the parameter construction, the parameter checks if an initial value has been provided to the broker to set its current value instead of the default value. Contrary to a static default value, the broker allows the definition of the initial value before the parameter registration. In the end, in order to be taken into account during the initialization of the parameter, the initial value has to be provided to the broker before the parameter construction. Otherwise, the default value is used. To sum up, the default value is set during the parameter construction. It is statically defined in the source code. The initial value can be set from any place in the SystemC hierarchy. It can only be changed by the module itself (by editing the code). It can happen before the parameter construction (otherwise it will not be taken into consideration). It overrides the default value. The broker is responsible to store the initial values. The CCI parameter implementation is responsible to check that an initial value has been set or not in the broker during its construction.

As discussed above, a parameter supports two kind of accesses: typed or untyped. Untyped parameters use the JavaScript Object Notation (JSON) [179] standard representation. However, in order to abstract this representation to the designer and tool, a new variant type called `cci_value` has been introduced. This class has been added to avoid the duplication of conversion code that needs to be written by the user between untyped and typed values. This new type is the core interface for serialization and deserialization between untyped and typed values. The core idea of this class is to enable an easy extensibility to support conversion for custom data types.

Internally, it is a thin wrapper around the RapidJSON [148] variant data type. It simplifies the usage and follows the SystemC naming styles. Full JSON support is automatically available from/to `cci_value`. Consequently, the designer only needs to define one set of conversion functions. This new type also avoids confusion with the type-safe access through setters and getters. The class represents a type-safe union for all plain types, SystemC types and user defined types. The CCI value class enables the handling of all types, including the complex types using maps or vectors of `cci_value`. Finally, it supports complex data types as listed in [16] with a better support for user-defined types. Typed parameters are specified from a template parameter to the CCI parameter / parameter handle class during the construction. Access to the actual value is more direct and efficient when the value type is know as it does not require a conversion to/from the JSON (string based) `cci_value`. The performance impact of this conversion is discussed in Section 2.5. It is envisaged that the `cci_value` class will be integrated into the core SystemC language as a `sc_variant` class in order to be used by other parts of the SystemC standard.

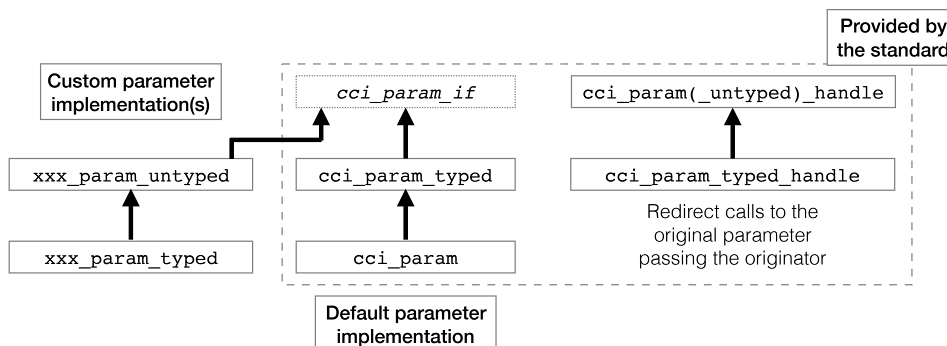


Figure 2.9 – CCI parameter and handle hierarchy

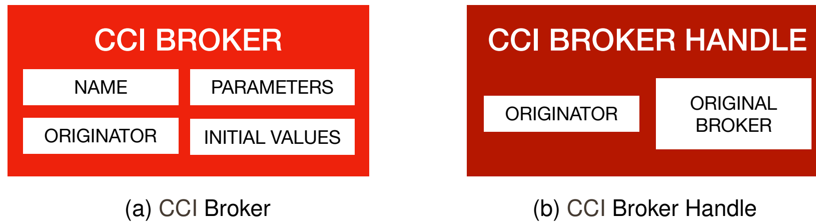
Finally, the parameter class hierarchy is summarized in Figure 2.9. The highest level of the hierarchy is the parameter interface. It includes all API that a parameter implementation must provide, including setter, getter, initial values, lock, callbacks, etc. The detailed API is available in the LRM [8]. All parameter classes which plan to be used by the CCI standard have to inherit from this interface. The interface does not have any specific typed API.

The hierarchy can be split into two levels : untyped and typed. The CCI parameter untyped class provides all the untyped API using the CCI value type abstraction class. The CCI parameter typed class has a template argument which specifies the type of the value. It adds the typed part and the more direct access API to the actual value. Both the CCI parameter untyped or typed handle class is part of the standard and does not need to be provided by a vendor implementation. The parameter handle uses a reference to a 2.9 as a class member in order to forward request to the real parameter implementation. Hence the parameter handle enables support for user implementations of the parameter interface, as well as the default implementation. For convenience, `cci_param` and `cci_param_handle` are aliases of `cci_param_typed` and `cci_param_untyped_handle`, respectively. They are the most used classes in the hierarchy, so shorter names are provided. However, for a better understanding of the hierarchy, complete names have been specified in the Figure 2.9.

2.4.4 Broker

The broker is the conductor of parameters. It is illustrated in Figure 2.10a. It manages parameters initialization, access-type, visibility etc. The broker is characterized by name, the database of

parameters, initial values and the originator that is explained in Section 2.4.5. Each time a model tries to get a parameter from the broker, the broker returns a parameter handle. Parameter handles are returned by value to simplify memory management issues. In case a parameter cannot be found, an 'invalid' handle will be returned whose methods will always fail. The SystemC module should check the validity of a parameter handle. Like parameters, a default implementation of the broker is provided by the standard but the vendor is left free to implement their own implementation as long as it implements the broker interface.



(a) CCI Broker (b) CCI Broker Handle

Figure 2.10 – CCI broker and CCI broker Handle

A SystemC module may want to hide some of its parameters for reasons of privacy. In that case, a module needs to use a private broker. Private brokers are necessary to deny or restrict access to their parameters. Hiding parameter can be done by instantiating and registering a private broker before any parameter creation. In Figure 2.11, parameters B and C in the module B are hidden and registered to the private broker. During the private broker creation, the broker is registered in a broker registry. This is a global registry storing references to private brokers in the simulation. By instantiating and registering the private broker, it becomes the default broker for all child modules. A private broker propagates down the SystemC hierarchy. The private broker can be retrieved through the standard API. After registration of the broker, nested parameters automatically register with the private broker. Private brokers are not accessible outside their associated module hierarchy (including by tools).

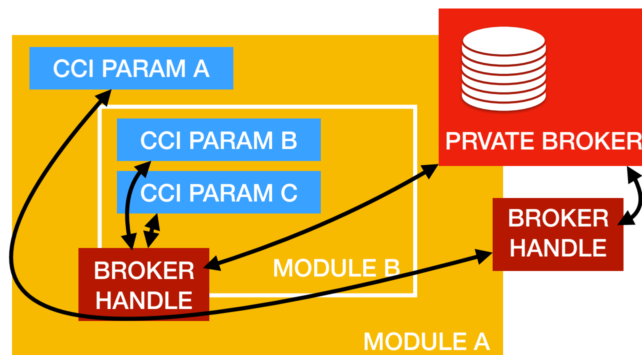


Figure 2.11 – CCI private broker hierarchy

Private broker not only support hidden parameters but they can also set public parameters thanks to a 'drift' upward mechanism following the hierarchy. In the default implementation, a list of parameter names can be specified in the private broker constructor. During parameter construction and registration to the broker, if the parameter name matches in the list, the broker drifts upward the registration request. If the parent broker is public, the parameter becomes available publicly. If the parameter name does not match with the list, it is initialized and registered to this private broker.

The main idea behind the broker registry is to ensure the private broker hierarchy within SystemC

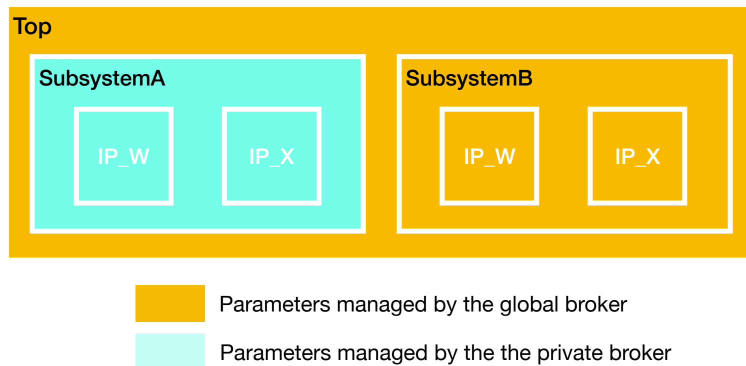


Figure 2.12 – CCI private broker registration

modules. It also means that a private broker which cannot successfully answer to a request forwards the request to the broker above in the hierarchy. It can be forwarded to a private broker or the global broker, depending on the hierarchy. The principle is described in Figure 2.12. Only the module itself can register its own broker. It is not possible to 'impose' a broker to another module. This means that it is not possible for a broker to be overwritten in the hierarchy. This ensures the registered broker will not change during execution. The registration must happen before any parameter instantiation in order to avoid parameters with different brokers which can lead to complexity for parameter destruction and resurrection as explained in Appendix A.1.5. Registered parameters in a broker can be accessed from tools or other models. The mechanism allows tools to track any changes or accesses to a parameter through callbacks. More details on the broker are given in Appendix A.1.6.

2.4.5 Originator

While the parameter handle has been briefly introduced in Section 2.4.3, a mechanism to track the SystemC module requesting the parameter through the broker, is described here. This mechanism is called the originator. It can be compared to the "identity card" of a SystemC module. It is used to track owners, handles and value providers of parameters. It also enables tracking of the identity of modules causing a parameter update. Listing 2.5 shows an example of the originator. In the example, the originator associated to both parameters is the timer module. As the module name is "timer", the originator name is the same. If the module was included in a subsystem, the originator name would follow the hierarchy.

Listing 2.5 – Originator illustration

```
1  // Timer.h
2  SC_MODULE(Timer) {
3      SC_CTOR(Timer):
4          m_input_clock_frequency("input_clock_frequency", 50),
5          m_interrupt_number("interrupt_number", 7) {
6          // ...
7          std::cout << m_input_clock_frequency.get_originator().name() <<
              std::endl;
8          std::cout << m_interrupt_number.get_latest_write_originator().
             name() << std::endl;
9      }
10 private:
```

```
11      // ...
12      cci::cci_param<double> m_input_clock_frequency;
13      cci::cci_param<unsigned int> m_interrupt_number;
14  }
15
16  // Main.cpp
17  int sc_main(int sc_argc, char* sc_argv[])
18  {
19      Timer timer("timer");
20      sc_start();
21  }
```

An originator is a specific class in the CCI standard: `cci_originator`. A parameter handle owns an originator. It is initialized during the handle construction. By default, an originator is associated with the current SystemC hierarchy within which it is created. For tracking or debugging purposes, the name of the originator can be retrieved. When a parameter is updated from a tool, outside of the SystemC module hierarchy, an originator containing an explicit string name must be set. In other words, for tools using CCI and not part of the SystemC hierarchy, any name can be chosen.

Figure 2.13 shows the originator mechanism used in the parameter handle. During the request for a parameter handle to the broker, the current module is necessary to create the originator. This information never changes during the lifetime of a parameter handle which is why parameter handle must not be exchanged by pointer or references between modules. Instead, they should explicitly be requested through a broker in order to maintain the correct originator. If the module "Top.SubsystemB.IP4" updates the parameter "Param_2_IP1" through the parameter handle "ParamHandle_1_IP4", the latest write originator of the parameter "Param_2_IP1" is the originator associated to the module "Top.SubsystemB.IP4".

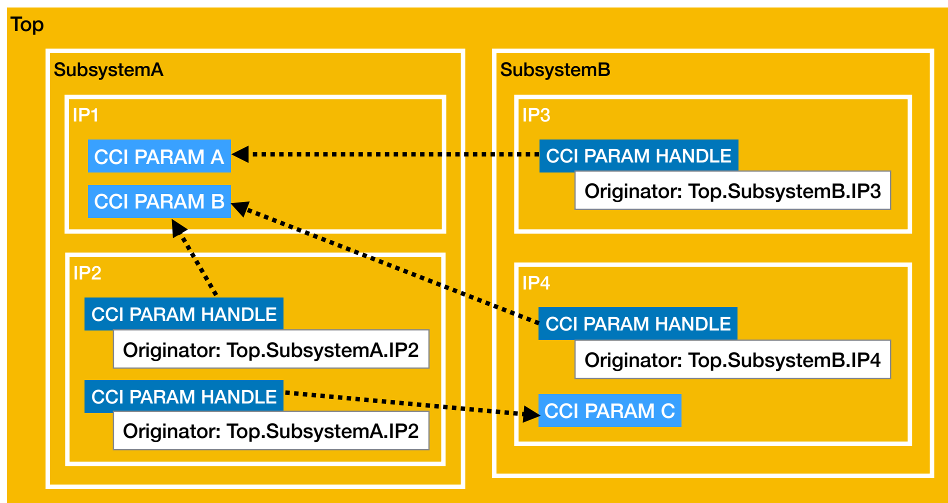


Figure 2.13 – CCI originator mechanism

2.4.6 Notification of read, write, creation and destruction of parameters

Notification of read/write accesses to parameter are necessary to track parameters changes and lifetime. Notifications have been implemented with a callback mechanism. Notifications can happen due to a modification in the parameter itself or more globally because parameters are

created or destroyed. Two different trigger groups have been provided:

- The first trigger group enables parameter changes and accesses to be tracked. It includes pre-read, post-read, pre-write and post-write. Listing 2.6 shows the usage of two callbacks "listening" for write changes in the "timer" module. Both callbacks are post-write.
- The second trigger group is provided to track parameter registration and/or destruction in the broker. Tools should be able to dynamically track all parameters which are created during the simulation. A handle for the parameter is made available in the event. This enables the tool to directly register value change callbacks on parameter creation.

Listing 2.6 – Callback illustration

```
1  // Timer.h
2  SC_MODULE(Timer) {
3      SC_CTOR(Timer):
4          m_input_clock_frequency("input_clock_frequency", 50),
5          m_interrupt_number("interrupt_number", 7) {
6          // ...
7      }
8  private:
9      // ...
10     cci::cci_param<double> m_input_clock_frequency;
11     cci::cci_param<unsigned int> m_interrupt_number;
12 }
13
14 // Observer.h
15 SC_MODULE(Observer) {
16     SC_CTOR(Observer):
17         m_broker(cci::cci_broker_manager::get_broker()),
18         m_timer_input_clock_frequency_handle(m_broker.get_param_handle(
             "timer.input_clock_frequency")),
19         m_timer_interrupt_number_handle(cci::cci_param_typed_handle<
             unsigned int>(m_broker.get_param_handle("timer.
             interrupt_number"))) {
20         // ...
21         m_timer_input_clock_frequency_handle.
             register_post_write_callback(&Observer::
             untyped_post_write_callback, this);
22         m_timer_interrupt_number_handle.register_post_write_callback(&
             Observer::typed_post_write_callback, this);
23     }
24 private:
25     // ...
26     void untyped_post_write_callback(const cci::cci_param_write_event<>
           & ev)
27     {
28         std::cout << "Parameter Name : " << ev.param_handle.get_name()
29                   << " Originator : " << ev.originator.name()
30                   << " New Value : " << ev.new_value
31                   << " Old Value : " << ev.old_value << std::endl;
32     }
33
34     void typed_post_write_callback(const cci::cci_param_write_event<
         unsigned int> & ev)
35     {
```

```
36         std::cout << "Parameter Name : " << ev.param_handle.get_name()
37                   << " Originator : " << ev.originator.name()
38                   << " New Value : " << ev.new_value
39                   << " Old Value : " << ev.old_value << std::endl;
40     }
41
42     cci::cci_broker_handle m_broker;
43     cci::cci_param_untyped m_timer_input_clock_frequency_handle;
44     cci::cci_param_typed_handle<unsigned int>
         m_timer_interrupt_number_handle;
45 }
```

Notifications (callbacks) are composed of a callback event that is passed as an argument to the registered callback method, function or lambda. This argument represents the data of the callback as in a TLM transaction. Depending of the kind of callback, callback events are different. This allows callbacks to track the exact change and the origin of the change. In case of a pre and post write callback, the event contains the parameter name, the old value (before the write), the new value (the value which will be set), the originator doing the write and a parameter handle. The difference between the previous and the new value can then computed while the parameter handle can be used to get information on the parameter. In the case of a pre or post read callback, the event contains the value, the originator and a parameter handle. It can also be used to track the value read, get the originator of the change and to do some operations through the parameter handle. Listing 2.6 shows all elements of the payload in the callback definition.

As described in the requirements, it can be necessary to reject changes. In that case, the return value of the callback can be used to dynamically reject writes to specific parameters. The choice to reject or accept the change can be done dynamically with the callback event. This mechanism can be used, for example, to validate the value in a specific range, filter on the originator of the change, or overwrite it if the specified value is not expected. Callback order has also been considered. In order to allow tools to register callbacks in a specific order, the callback execution follows the callback registration order. This enables the tools to lock parameters before any other callbacks by positioning themselves at the highest hierarchical level. The same behaviour can be found in the Universal Verification Methodology (UVM) standard [3]. More information on the implementation of the CCI notification is available in the Appendix A.1.3.

2.5 Performance analysis

The performance, flexibility, features and robustness of a SystemC simulation is vital to many use cases. Before the CCI standard, parameters were already part of simulations but implementations varied, as was seen in Section 2.3. Some parameters can be just plain raw data and for others C++ class providing various features. One aim of the CCI standard is that the impact of the new features should not add a significant overhead to the existing simulation compared to the plain data types.

The features that have been tested are reads and writes to typed parameters and to `cci_value` (hence the untyped/typed conversion). Tests have been run with and without callback mechanisms. These have been tested in a 'raw' environment with no activity other than the parameter itself, in order to evaluate a raw performance metric. The performances in a concrete case to show the real impact on a complete simulation platform have also been compared. Each test below has

been run ten times and an average has been applied.

2.5.1 Raw

The first benchmark of the CCI POC has been done with a simple read/write loop. Different kinds of parameters have been tested: typed, untyped, typed handle, untyped handle. In the case of the write test, parameters are assigned to a value equal to the index of the loop. The type of the parameter is an `unsigned long`. In the case of read test, parameters are read in each step of the loop through the typed or untyped getter. The test is composed of a billion writes or reads to the plain data type. The same loop has been applied to a `cci_param` with the typed setup and the untyped setup, and to both typed and untyped `cci_param` handles.

Table 2.1 – 1 billion write loop

Data type	Runtime (ms)	Overhead
unsigned long	1699	
cci_param_typed<unsigned long>	3332	2
cci_param_untyped	35536	20.9
cci_param_typed_handle<unsigned long>	19419	11.4
cci_param_untyped_handle	37939	22.3

Table 2.1 shows the result of the write loop test. The `cci_param_typed` overhead can be explained by the use of method calls and hence redirection prior to the actual write as done with the plain data type `unsigned long long`. The untyped write, `cci_param_untyped`, overhead is related to the use of `cci_value`. In this case a `cci_value` instance is created. To be initialized, the typed unsigned long value is first converted to a string value. Then, inside the setter, the value is converted back from string to an unsigned long. Similar behaviours are observed with the untyped handle parameter, `cci_param_untyped_handle`. Meanwhile, the typed parameter handle is slower than the original parameter. Parameter handles act as a mirror doing the redirection of each function call to the original parameter adding the originator information. This level of redirection adds a small overhead. Nonetheless, as the originator of the write is not the same as the original parameter, the latest write originator has to be updated for each write.

Globally, these results can be explained by the overhead of various features provided by CCI: pre and post callbacks, locks and the mutability. However, the CCI POC is not optimized for performance. Instead, its focus is on features and ease of maintenance. Initially, results were worse but a cache mechanism has been implemented to avoid callback checks when a parameter does not use these features. It introduces a kind of "fast path" doing a first check of the cache value before doing the usual write. This helps to reduce the overhead by a factor of 7 for the `cci_param_typed` results. However, in case of `cci_param_untyped_handle`, the "fast path" is not used as the originator of the modification has to be updated for each write.

Similar results for the read test are given in Table 2.2 . The overhead is smaller as there are less possible branches. For instance, a lock only related to writes, hence less conditions are checked during the getter execution.

Clearly, overall, the raw results show a non negligible overhead. This is exacerbated by the CCI POC that focuses on features and not on performance. However, a concrete SystemC module

Table 2.2 – 1 billion read loop

Data type	Runtime (ms)	Overhead
unsigned long	1808	
cci_param_typed<unsigned long>	2363	1.3
cci_param_untyped	21259	11.8
cci_param_typed_handle<unsigned long>	8749	4.8
cci_param_untyped_handle	24551	13.6

does not typically run intensively reads or writes on a CCI parameter. Such a module would not involve anything else except for a direct parameter read/write. At the very lease, TLM-2.0 transactions between reads or writes (for instance a TLM-2.0 router) with CCI parameters as addresses to route transactions to the right modules in the memory map can be imagined. To highlight this point in the next subsection, the overhead of CCI is estimated on a real use case.

2.5.2 Concrete usage

In the case of a SystemC on Chip, parameters are involved in many models with various usages. Some parameters will never be written except during their initialization like the parameters for the configuration of a static model. Others are necessary to capture model state. They are written and read often like in a counter.

A performance measurement is provided in the precursor of CCI [163]. The benchmark is composed of two TLM-2.0 devices that exchange transactions and writes to a parameter. Unfortunately, it is not possible to reproduce the overhead observed in the paper, even though it has been attempted to reproduce the exact conditions, including the parameter itself, identified in the paper, the results are significantly worse. Compared to a plain data type, at the very least, a function call is performed through a virtual table which adds some overhead. Though, of course, this comes with the benefit of adding more flexibility.

In order to be closer to the concrete usages of parameters in a virtual platform simulation, a more complete benchmark has been constructed. This test computes the time to run a billion TLM transactions using the LT level. Each transaction in the TLM target module produces a write to a CCI parameter with the value provided in the payload of the transaction. The benchmark has been executed 10 times. The result corresponds to the average. Results are given in Table 2.3.

Table 2.3 – 1 billion TLM transactions with write

Data type	Runtime (ms)	Overhead
unsigned long	7199	
cci_param_typed<unsigned long>	13070	1.8
cci_param_untyped	47483	6.6
cci_param_typed_handle<unsigned long>	29997	4.2
cci_param_untyped_handle	53838	7.5

Of course, the intensity of reads and writes is related to the usage. On the one hand, when

parameters are used to specify peripheral addresses in the memory map, reads will be more frequent than writes. The router uses this value to dispatch transactions to the right IP. When CCI parameters are used to store the value of registers, callbacks can be applied to mimic the real-hardware behaviour which can add more latency to each read and write. A complete virtual platform using CCI parameters will be examined in detail in Chapter 5.

2.5.3 Callback

In this benchmark, the pre-write callback mechanism has been measured. This callback can deny a write with its return value. In this benchmark, the write is always approved. The callback payload is also relatively large containing the old current value, the new value, the originator and a parameter handle. In principle, the callback contains logic to handle the various cases: denied write, new value overwritten... This adds overhead. For the purpose of the benchmark, the callback itself has been left empty in order to only measure the mechanism overhead itself. Unfortunately, the actual callback mechanism performances itself was not provided by the precursor of CCI paper.

Table 2.4 contains the results. The overhead is compared with a parameter write without any callback enabled in order to isolate the callback mechanism impact. The impact is not negligible. As a 'fast path' mechanism was deployed, the difference between a parameter write and a parameter write with a callback is larger than without the 'fast path'. As the 'fast path' disables the check of different features (including callbacks), the overhead is explained by these checks as well as the callback mechanism itself.

Table 2.4 – CCI untyped pre write callback 10 millions loop

Data type	Runtime (ms)	Overhead
cci_param_typed<unsigned long> (no callback)	52	
cci_param_typed<unsigned long>	5598	107.7
cci_param_untyped	5829	112.1
cci_param_typed_handle<unsigned long>	5482	105.4
cci_param_untyped_handle	5926	114

In order to reduce this impact in the global simulation, a custom implementation of the parameter can be used in order to speed up specific cases. Such a parameter could also avoid branching in the setter and getter triggering the callbacks. However, contrary to the "fast path" solution quoted in the previous section, this would reduce functionality. The branching is run-time dependent and the usual read or write depends of the callback return. It cannot easily be predicted.

2.5.4 Conclusion on performance evaluation

Although CCI has introduced a non negligible impact on parameter access performance, it should be seen in the context of the wider simulation. Plain data types are certainly faster and can be useful in some cases, but, they are inaccessible from outside the model. They have none of the features that the CCI parameters provide. Furthermore, it has been shown that parameters can be optimized in the case that a parameter does not use callbacks with a software branching avoidance path. The CCI POC has initially focused on features and the standardization of the

interface. Final implementation is left free to vendors. In the end, for the most part, designers need the setter and getter features that parameters offer. Moreover, in my opinion, the overhead remains acceptable, given the features that parameters bring.

In the case of a read or a write happens in the same module as the parameter owner, callbacks can be replaced with a direct handling of the change. However, if parameters are just used before the end of the elaboration phase and only impacts the initialization, then a degree of flexibility can be benefit in regard of the small drawback of an increase of the initialization time. Finally, the overall performance and the features provided by the CCI standard give a right balance between performance, features, and interoperability.

2.6 Breadth of the standard

The initial purpose of a CCI parameter was simply to enable the configuration and inspection of a model. Increasingly, it has become apparent that a parameter of this sort can be used to achieve a wide variety of purposes. One possibility is to use parameters to track all the state variables of a model. This has some key advantages, both in terms of debug and analysis of a model. But, it also means that the model can potentially be saved and restored to a known state. This mechanism has been used to save and restore SystemC models. It is the fundamental mechanism behind reverse-execution, a useful technique for complex debug as in [123]. The other key area in which CCI parameters can be used is that of register. A similar approach has been proposed in [75].

Another key area of CCI is TLM-2.0. Transactions initiated from CPU(s) are routed through a TLM-2.0 router to the right module reading the address of the transaction and the addresses of module(s). In order to do the right routing, the router need to be able to read the position of each module on the memory map. CCI parameters can be used to specify in the TLM-2.0 sockets their base and high addresses on the memory map. This value can then be retrieved by the router. Currently, no standard mechanism is available and solutions are introduced in Chapter 3.

TLM-2.0 transactions are characterized by their payload and their delay in case of LT and also their phase in case of AT. A payload class (and the phase) can be extended in order to add new attributes to a transaction. New fields can characterize a protocol like the bit-rate, the clocking mode, etc. If these characteristics do not match on each side, then an error should be triggered. It is often synonymous of a bad software configuration. However, some characteristics are not changing frequently. They are only set during the initial configuration of the module. In this case, they can be considered as part of the protocol negotiation. A solution based on the breadth of CCI will be presented in the Chapter 3.

2.7 Limitations of the standard

The single most obvious issue with the CCI standard is that it only standardizes the parameter types and associated interfaces. It does not standardize the names, types and meanings of those parameters. While it is inconvenient, it is surmountable. An obvious case of this, is to add parameters to inform a memory map. Given the way in which standardization works, what is required here is a way of agreeing on names, types and meanings alongside the CCI standard. Effectively, in order to exploit the standard, users need a place to record the names, types and meanings of the parameters. Where those are common, ideally, those records should be public.

The second issue with the CCI standard is that, while an implementation of a parameter is provided by Accellera, no means by which the parameters can be initialized from configuration files or command lines is provided. This makes sense, as the POC simulator is just that, a proof of concept. It is not intended to be a fully fledged EDA tool. None the less, there is an utility that can read from configuration files, and initialize parameters like in [75]. It can be used directly with the SystemC POC simulator to allow configuration data to be read from a number of different configuration files types.

As discussed before, SystemC attributes enables the attachment of elements to a SystemC object but the API does not support CCI parameters. An evolution of the attribute mechanism that can be replaced by CCI parameters would be more flexible and offer fuller features. It has been proposed. Note that this is under consideration by the SystemC WG.

2.8 Conclusion

Accellera introduced TLM-2.0 has introduced a standard layer of interoperability between models for memory mapped communications. It facilitates interconnection between models through buses. However, neither SystemC, nor TLM provide a mechanism to configure, control, and introspect the models through an unified and standardized API.

This chapter introduced a literature overview of existing configuration mechanisms. It also described their use cases and features they provide and a detailed analysis of the precursor of CCI. This is where my actual PhD work started. The chapter showed that requirements have evolved and that the CCI standard should be completely redesigned to be more robust and flexible given the new requirements.

My contributions introduced a new architecture that has been detailed in this chapter. All these proposals have been implemented and evaluated by the CCI WG. The work includes parameters, brokers and notification mechanisms. For the first time, extensibility and backward compatibility with existing configuration mechanisms has also been considered as a primary objective. The results have been presented during Design & Verification Conference & Exhibition (DVCon) Europe 2016 [68] through a tutorial session . Performances of the POC showed that the standard introduced an overhead over raw variables. However the POC is not optimized for performance. The overhead is likely to be negligible in a more concrete case. The standard and details about the standardization process are given in Appendix A.1.2 and Appendix A.1.4.

In order to use the CCI standard for TLM-2.0 non memory mapped protocols, some rework of the TLM-2.0 standard is required. Backward compatibility should be considered for systems not using the CCI standard. The next chapter will introduce in details the issues with non memory mapped protocols and solutions taking advantage of the CCI standard.

Chapter 3

TLM for non memory mapped protocols

***

3.1 Introduction

Increasing the level of abstraction, by removing time or simplifying functionality helps to increase model performance which enables models to be used for a wider number of use cases. The TLM standards objective is to enable models at a high abstraction level to communicate between themselves. It is based on a technology, namely the use of transactions for simulations. The TLM abstraction levels are designed to ease model creation discarding different amounts of detail data transfers. However, abstraction levels are therefore relatively broad, and cover a wide range of use cases. This is potentially one of the problems with the TLM-2.0 standard. Indeed, it tries to satisfy a wide range of requirements.

The TLM standard is built on interfaces between two kinds of elements : initiator and target. A single element remains statically assigned to a specific role. However, the standard enables models to possess the two types of interface, as illustrated in Figure 3.1. In this configuration, an initiator is in charge to communicate with the target through the interconnect. A set of data is directly exchanged.

The scope of TLM-2.0 is wide as illustrated in Figure 3.5. The TLM-2.0 standard includes two abstraction levels called LT and AT. These names do not help to identify implicitly the use cases they cover. The TLM abstraction levels remove different amounts of detail from the communication protocols. Both are implemented using transactions and function calls to communicate the transactions. The transaction remains common between the two types of interface. The LT abstraction level aims to support virtual platforms for software development, with only as much timing information as the programmer needs. It is implemented with a blocking function call at the initiator side. The initiator has to wait until the target finishes its processing. Transactions are completed in a single function call. Conversely, AT aims to provide more accurate timing, that can be used to drive architectural exportation, synthesis, etc. It is based on a non-blocking call mechanism and a sequence of phases which are protocol specific. The phases define the timing points of the protocol. The additional modelling of time has a cost in terms of simulation performance.

While the ambition for the TLM-2.0 standard is wider, in reality it has primarily focused on enabling interoperability for memory mapped bus based protocols [5]. Thus, it standardizes the simulation of system that contains bus models. However, current virtual platform do not contain only memory mapped protocols. Consequently, individual designers have to define their own TLM protocols, which are often non-interoperable. Consequently, the aim of this chapter is to propose some

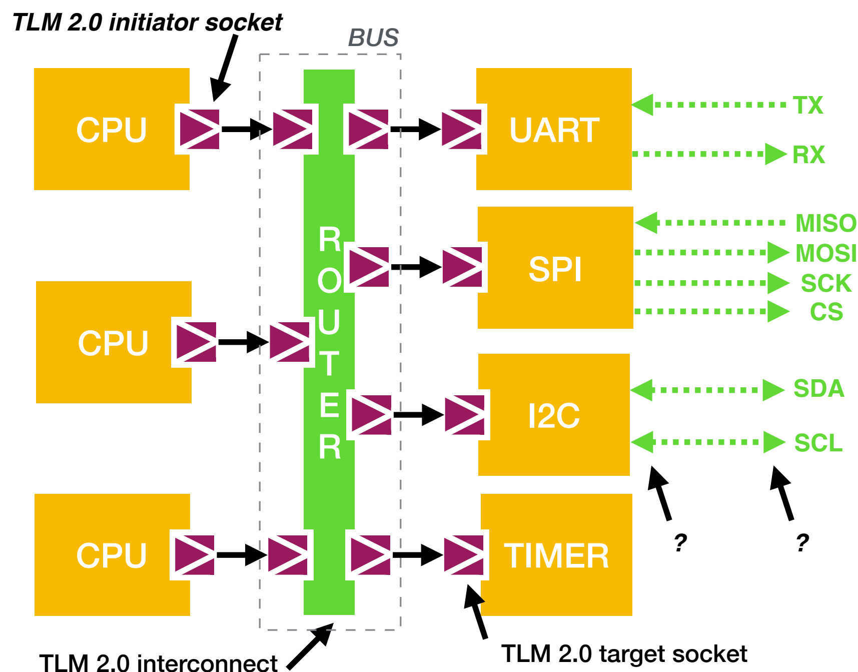


Figure 3.1 – TLM-2.0 overview in virtual platforms

solutions:

- improvements to the existing TLM-2.0 standard,
- a methodology around the TLM standard with the objective to facilitate the definition of a wide range of protocols in a consistent manner.

The rest of the chapter is organized as follows. An introduction to the modeling of communications is presented in the Section 3.2. Then, related works is given in the Section 3.3. Some non memory mapped protocols are then detailed to extract the key points in the Section 3.4. An evolution of TLM is proposed in the Section 3.5. The Section 3.6 describes the interoperability between TLM and CCI and how CCI can help for the protocol configuration check. Finally, a conclusion is given in the Section 3.8.

3.2 Modeling communications in virtual platforms

3.2.1 Introduction

SoC complexity has increased. SoCs are now composed of many processors, accelerators and peripherals as showed in Figure 3.2. This figure illustrates the architecture of the SoC Xilinx Zynq 7000 [198]. Communications between SoC elements are done through an AMBA interconnect. On one side, elements are connected to the interconnect. On the other side, they can also be connected directly to other peripherals like the I/O mux, which enables communication outside of the SoC (with various protocols). Internal communications are performed using bus communications like AXI, APB, AHB, interrupt signals, etc. External communications are commonly performed with GPIO, UART, I2C, CAN, Universal Serial Bus (USB), Ethernet...

While processor and peripheral behaviours can be modeled with SystemC modules, the com-

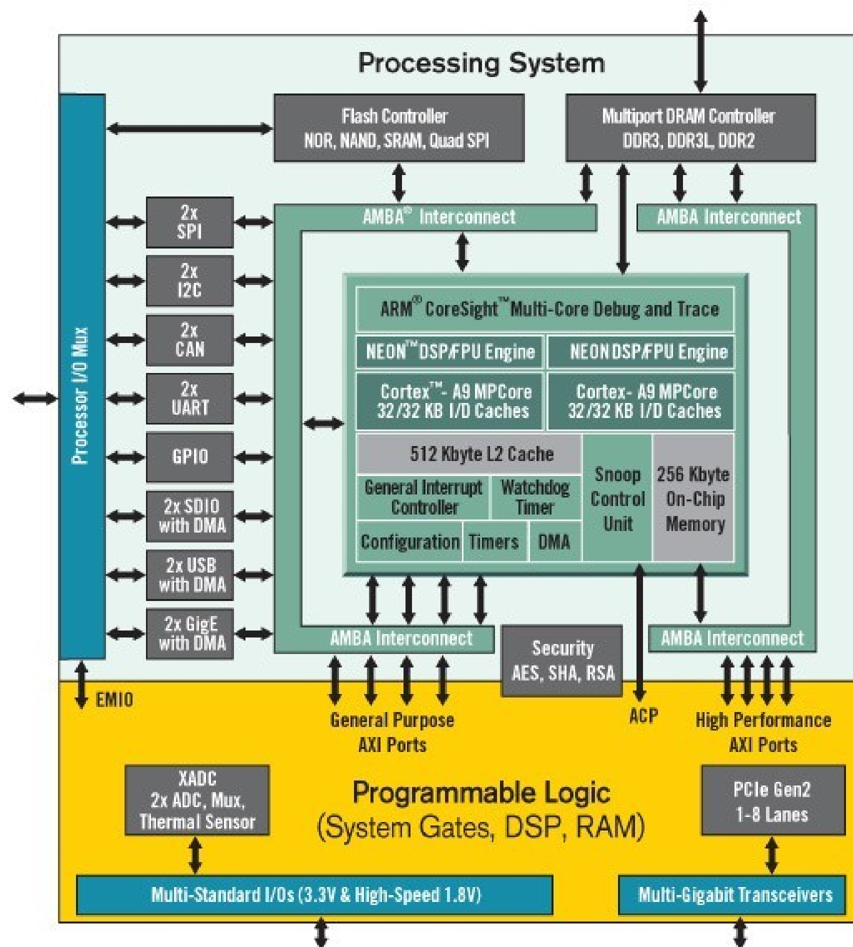


Figure 3.2 – Zynq SoC

munication between elements can be done in different ways. The SystemC channel mechanism can be used as showed in Figure 3.3. Channels are generic and can be used to model hardware communication like SoC communication. In Figure 3.3, a FIFO channel is used for the communication between a sender and a receiver. As presented in Chapter 1, the user can define their own SystemC interfaces for ports and channels. This enables a level of customization for the designer in order to define specific behaviours. Furthermore, it also enables the designer to adapt the abstraction level of the communication model to their requirements.

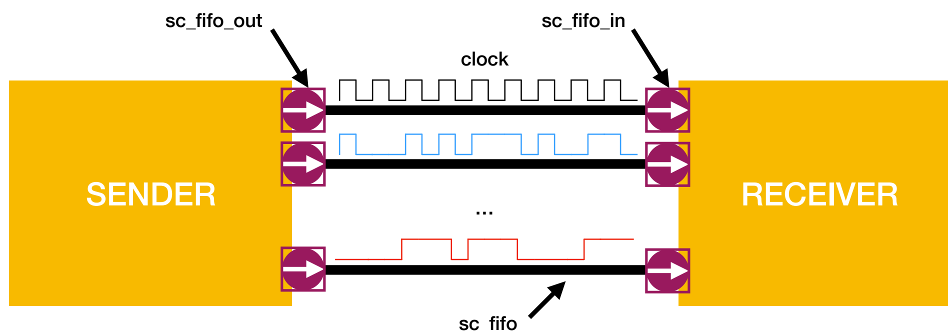


Figure 3.3 – SystemC channel description

However, communication channels are time consuming both in terms of the time and effort it takes

to model them and the time they take to execute. At execution time, read and write transactions involve SystemC kernel calls that are penalizing. SystemC kernel calls are used to guarantee data consistency and determinism according to simulation time. Data exchanges at low abstraction levels can represent more than 90% of the simulation time. SystemC ports and channels use data copy between each step of the data exchange. Data copy can be non optimal and CPU consuming.

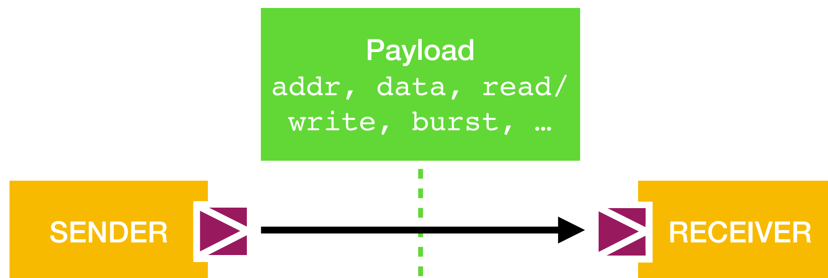


Figure 3.4 – TLM socket principle

To reduce the amount of SystemC kernel calls and the data copy, TLM was proposed. In the example given in Figure 3.3, different channels can be regrouped into a TLM socket. The data exchange can also be moved into the TLM payload as showed in Figure 3.4. This drastically speeds up the simulation but reduces the accuracy of the communication model. Depending on the abstraction level and the required degree of accuracy, the content of the payload of the transaction is constant but the phases change. This choice is left to the designer choice. The next section purposes a definition of a transaction.

3.2.2 Towards a definition of a transaction

A transaction is not clearly defined in the TLM standard and is left to the understanding of the designer. What has and has not to be modeled at different abstraction levels is not clear. A misunderstanding can decrease the level of interoperability between two modules that have to communicate with the same protocol. Within the context of a wider scope of protocols support in TLM, there is no clear agreement of what constitutes a transaction. Unfortunately, little literature on this subject is available. It is crucial to understand what exactly is meant by a “transaction”, what should be held in a transaction, and what should be left out as mentioned in [69]. This leads designers to make radically different choices about how to model interfaces. One example of this is how different people have addresses the subject of modeling a simple wire. A wire and signal modeling is given in [178] and [176] respectively. These two approaches are not interoperable, although they attempt to solve essentially the same problem.

3.2.3 OSI and TLM

In making a proposal to define a transaction for TLM on an existing standard, the Open Systems Interconnection (OSI) communication model [89] is interesting. A similar approach was taken in [69] [155]. The OSI model divides communication protocols into multiple layers of abstraction as illustrated in Figure 3.6. A detailed description of the three first OSI layers is given in the Appendix A.2.1. This model is extensively applied to describe network communications. However, it is also

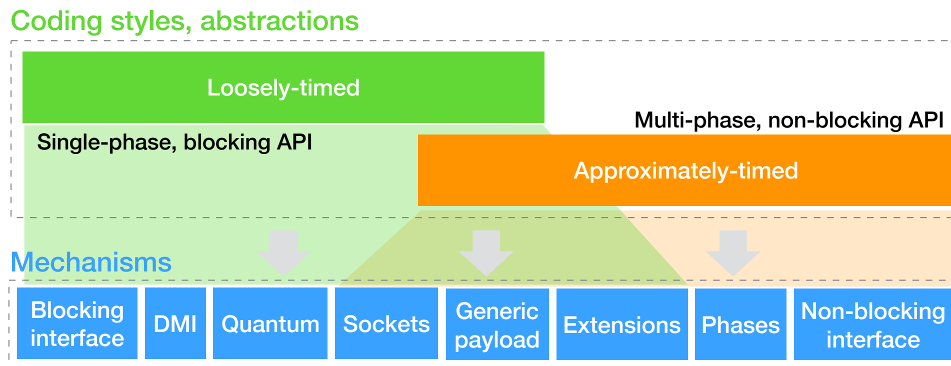


Figure 3.5 – TLM-2.0 parts

designed to have wider applicability. For instance, the CAN bus standard refers to the OSI model to describe the protocol [1].

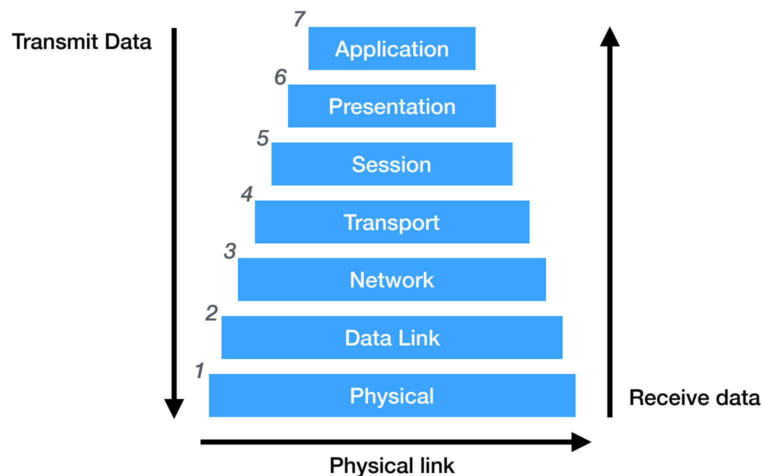


Figure 3.6 – OSI model layers

A transaction for a memory mapped protocol is universally assumed to be a read, a write or in some cases a read-modify-write operation. The transaction carries all the data that are necessary for a complete operation. For buses that are capable of performing bursts, the data covers the entire burst [7]. In other words, the notion of transaction is the full operation. This fits perfectly with the OSI layer three. The packet contains the address and the data, just as does a TLM generic protocol transaction, and is routable.

The proposal here is an abstraction level to expose the details of the network layer, and to use something akin to the OSI packet as the transaction is expected. In that way, a direct mapping of the OSI data link and / or network layer of the protocol is applied onto the transaction. The data layer is dealing with frame and buffer congestion. The third OSI layer maps onto the interconnect dealing with routing and packets.

The OSI network layer includes resource conflicts along with routing. In the case of the LT abstraction level, that is more abstract in its timing, time-based resource conflicts may be ignored or not modeled all. A more naturally belong at the lower level of abstraction in terms of TLM is considered. The protocols and associated timing points are exactly akin to the burst packets. Protocols are modelled at the AT abstraction level in a complete bus modelling kit, such as the OCP modelling kit referenced above.

In order to guide designers choice about how to write interface models, it is critically important to have agreement on what constitutes a transaction. The adoption of the OSI model is considered as not only technically robust, but also comes with the benefit of an existing, well documented, and well understood standard. A related work about modeling of non memory mapped protocols is detailed in the next section.

3.3 Related works on abstract communications

To reduce SystemC kernel calls to speed up simulation and to facilitate the modeling of virtual platforms, TLM communications have extended the SystemC transport mechanism. Coarse grain modeling of memory mapped communication protocols is then enabled. However, some features are missing for virtual platforms modeling, including, for example, an easy extension mechanism for adding new protocols support; a better interoperability for these new protocols; the modeling of communication protocols other than buses as presented in Figure 3.7. In this example, communication protocols include buses, signals, UART, I2C, etc. They are common protocols in SoCs. Previous work is discussed in this section.

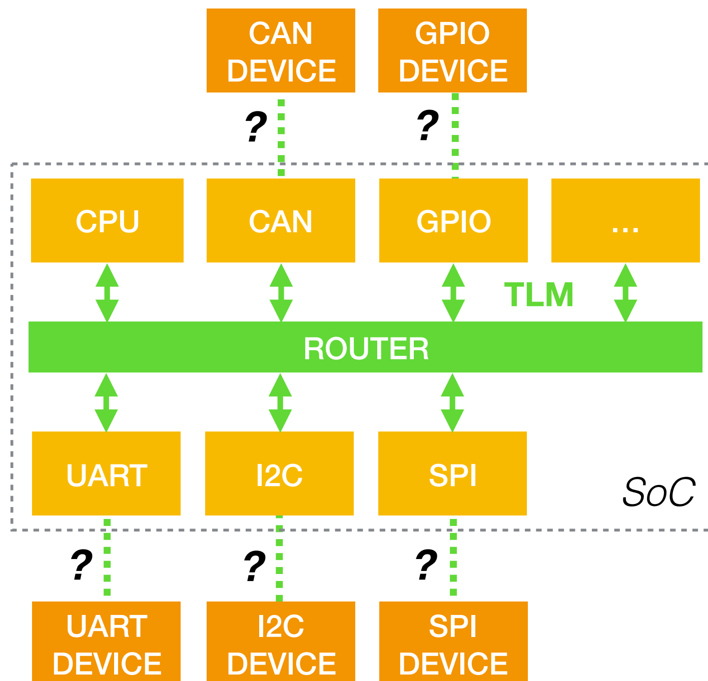


Figure 3.7 – Example of non memory mapped communications

The abstraction of the communications with TLM-2.0 standard is discussed in the paper [126] and presents a new methodology to analyze system level performance. The Authors defined accesses as a mechanism to abstract communications and allowing seamless refinement. Although TLM allowed them to speed up their simulation removing unneeded details, they conclude saying communication channels need to be formalized. Finally, this paper did not study non memory mapped protocols and did not treat in detail the temporal aspects. The temporal decoupling in TLM communications can add a potential latency. This can happens when the data is coming from a SystemC channel for example which is then feed into the TLM mechanism. This is explored in [79] which deals with FIFO-based communications. While temporal decoupling has been applied to

memory mapped communications, FIFO-based communications raise new issues. Using `sc_fifo` in TLM models implies more synchronization with the SystemC kernel and defeats the purpose of the temporal decoupling. However, removing synchronization in FIFO does not ensure functional accuracy and breaks the timing. Instead, they proposed a smart FIFO for TLM usages and benefiting from TLM features for un-timed models using local reader and writer TLM time. The proposed solution looks interesting for communication protocols having arbitration like CAN in order to improve the accuracy of incoming data.

According to the papers [170][192][125][201][98], virtual platforms complexity increase drastically. In virtual platforms, even if TLM-2.0 standard was designed for memory mapped protocols and used for bus modeling, communication with UART, SPI and other non memory mapped protocols should not be ignored. They can be used to validate the communication inside the platform but also the communication outside of the platform with external devices. However, none of these publications modeled non memory mapped protocols like UART or SPI protocols whereas platforms in papers [125][201][98] include UART IP and papers [192][98] include SPI IP. The application scope of virtual platform is so limited.

The authors of the paper [188] also employs TLM-2.0 standard for bus communications and does not consider non memory mapped protocols. Likewise, [138] suggests that TLM is mainly intended for exploring various bus-based communication architectures. In addition to providing blocking and unblocking interfaces for transactions, TLM also provides a DMI feature. This mechanism is intended to be used for memory accesses. It aims to decrease the number of TLM transactions as presented in the Chapter 1. While this interface is not essential, it enables to speed up data set transfer as in [48], used for the modeling of a flash memory. Finally, as mentioned by the authors in [80], TLM-2.0 standard did not target their use case : interrupts. They used plain SystemC ports with a specific SystemC interface for the interrupts. That is to say the ability to use different level of abstractions in the same virtual platform is also used [124].

The generic protocol [5] offers three recommended alternatives for transactions which are not directly compatible with TLM-2.0 standard. The first option consists of ignorable extensions of the TLM-2.0 standard base protocol for fields that are not present in the generic payload. The second option consists of the definition of a new protocol inheriting from the generic payload. Finally, the last and least inter-operable solution enables the designer to define a new protocol using a new transaction type. It is not recommended by the standard for interoperability. A solution for non memory mapped protocol was proposed in [145]. It was applied to SPI communication links. Unfortunately, papers they do not detail the solution they have chosen, they do a comparison of SPI transactions based on a CABA model and a TLM model. At the CABA level, the serializing and the sending of bits one at a time is made at the SPI clock frequency thanks to a SystemC process as showed in Figure 3.8.

For the TLM model, the TLM SPI data transmission is performed at the byte level. Byte transfer is delayed according to the SPI clock speed. Even if the socket connection is not detailed, a specific socket is mentioned for the Master Output Slave Input (MOSI) part of the SPI protocol. While the TLM-2.0 standard aims to abstract transactions, the separation of each signal which defines the SPI interface with a TLM socket is maybe not the optimal choice in terms of simulation performance. It increases the number of sockets to bind and so human errors. While this choice could be useful if each signal is redirected to different modules, this is rarely the case. For SPI, all signals are connected to each master / slaves. Finally, they conclude that the usage of a TLM model for SPI transaction drastically reduced the number of synchronizations with the SystemC kernel compare to the CABA model. Thus simulation time was faster. The SPI protocol is detailed in the Section3.4.

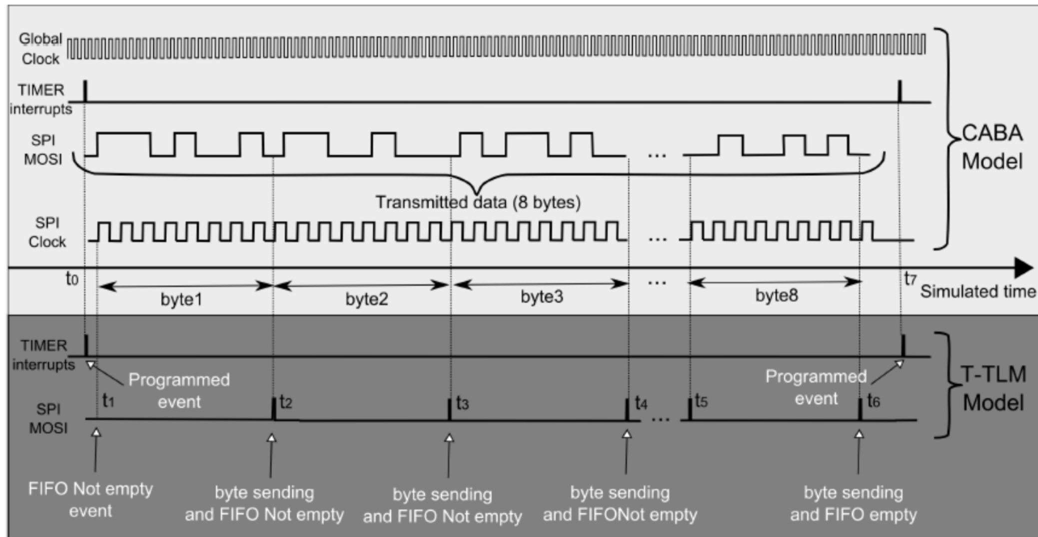


Figure 3.8 – SPI CABA level vs TLM level [145]

Solutions to model others protocols using the TLM-2.0 standard were also proposed. The authors of [161][160] modelled CAN. Ethernet protocol is presented in the paper [205], USB-3.0 in [174], and I2C in [147]. For the ethernet protocol, authors have implemented a specific socket, payload and phases. A bidirectional socket is required but TLM-2.0 standard does not provide this in the current standard. The publication [22] also described the implementation of a bidirectional version of the TLM socket with two channels. In that case, the third option offered by the standard [5] is used, which explains the related interoperability issues. In [151], the authors have mentioned the pin multiplexing issue: the same pin on a chip can be used by multiple protocols as showed in Figure 3.9. The author finally looked for elaborating a solution for serial protocols with TLM. They propose a generic solution for serial protocols providing a new payload with fields for the routing and arbitration. This includes new interfaces.

When accuracy is required for communication latency, it can be necessary to develop models at low abstraction levels (e.g. cycle accurate models). On the other hand, TLM-2.0 standard can be used with the time decoupling feature decreasing the latency accuracy. A first approach for the SPI interface has been discussed above according to [145]. However, no concrete solution is given to improve the accuracy. Interrupts are also part of these issues. If an interrupt is triggered to a non-TLM module that is in sync with the SystemC kernel, the interrupt can be delayed and break the functional part. When TLM-2.0 quantums are used, but signals need to be treated within a quantum, the standard suggests reducing the quantum appropriately. Meantime, the authors of [29] present a different solution based on a rollback mechanism. This solution is not compatible with SystemC/TLM. A SystemC/TLM compatible version is also given in [123]. The rollback mechanism looks like a good solution to fix the accuracy issue but can also impact the simulation speed. The solution detailed in [72] purposes a generic mechanism to adapt dynamically the quantum to avoid a latency with a black box model as input. Results are encouraging and can be a candidate for non memory mapped protocols issues in the communications. SystemC-Analog/Mixed-Signal (AMS) [4] can also be used for protocol modelling like I2C [9].

Protocol checking is also a critical issue as the protocol configuration can change during run-time. A solution based based on Petri networks is given in [21]. It is based on a previous paper [20] that does extraction of TLM configuration to a Petri network. Petri networks enables the TLM-2.0 standard protocol consistency used by modules to be checked. The check has been done with

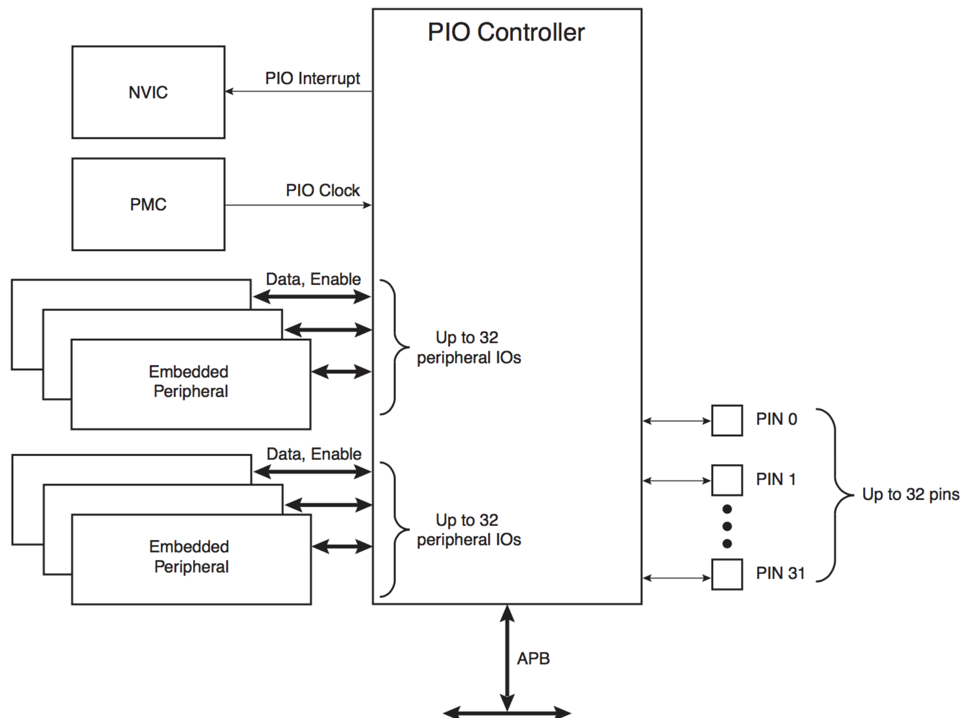


Figure 3.9 – Parallel input/output controller on SAM3X driving PIN input/output from multiple peripherals

the default TLM-2.0 standard protocol, but, it can be extended to customized one and so can be used to detect deadlocks within a design. Another solution is presented in [27]. They defined testbenchs for TLM protocols in order to check accuracy at various levels. They aim to validate the protocol functional accuracy. [26] and [25] also did model checking of TLM protocols, there the check is done by TLM synthesis to RTL and then with the RTL model checker.

There are many communication protocols inside and outside SoCs. SystemC channels can be applied to model them all, and the literature shows this solution being used to model buses as well as interrupts in virtual platforms. However, the abstraction level may not fit the requirements, and the solution may suffer from poor simulation performance. On the other hand, TLM can be used to model communications with better results in term of simulation performances. However, it primarily intended to support memory mapped protocols and poorly supports non memory mapped protocols. The main objective of this chapter is to propose an improved version of TLM to better support memory mapped protocols and non memory mapped protocols in virtual platforms.

3.4 Evaluation of protocols

3.4.1 Introduction

Virtual platforms are composed of different elements that communicate in a non homogeneous way. Communication protocols can be split into different categories. Memory mapped protocol include all buses protocols like AXI, AHB, APB... On the other hand, non memory mapped protocols include UART, SPI, I2C, CAN, Ethernet... Except I2C can be memory mapped, as can SPI and both have a notion of 'address'. The family of non memory mapped protocol protocols can be

also split in different sub parts. Some protocols like UART are "One to One" communication links whereas SPI or I2C are "One to Many" and CAN or Ethernet are "Many to Many". These families of protocols are analyzed below. The analyses shows how the OSI model can help to establish the transaction, and associated phases. The intention is to exploit these protocols to examine different aspects of how interface kits should be built, and to validate the use of the OSI model.

3.4.1.1 “One to One” protocols

In this section, one to one protocols are considered. These involve one sender and one receiver during the transaction as described in Figure 3.10. The most common and low level communication protocol is surely the signal. Although in general, the signal is considered as simple, often that masks a level of detail. For instance an interrupt, a kind of signal, can have different levels, depending of the interrupt vector peripheral and their physical wiring. It is not always the case that because a protocol is implemented on a single wire, the protocol is in any way ’trivial’.

Figure 3.10 – One to one protocol scheme

In virtual platforms, signal links are often required to model interrupt connections. It is the simplest one to one communication link. By considering only signals that can be active (asserted), or not active, the ’packet’ is the assertion of the signal. In case of an Interrupt ReQuest (IRQ), the number of the IRQ should be available for the receiver (mostly an interrupt controller). The number associated with the communication link does not change during the run-time of the simulation. This means that the information can be redundant with a TLM extension. Furthermore, many if not all of these “simple” signals often carry extra information such as vector address. In the hardware architecture, this may be encoded by their physical placement, their wiring or indeed by multiple wires.

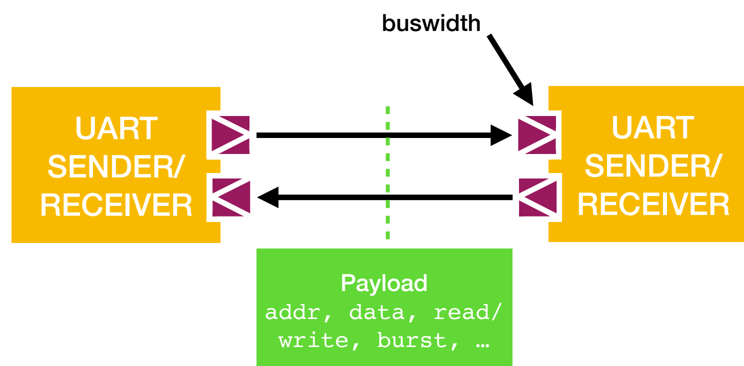


Figure 3.11 – UART model with TLM-2.0

UARTs protocol can be seen as a family of multiple serial protocols as well as Recommended Standard 232 (RS-232) or Recommended Standard 485 (RS-485). It is based on a transmission and reception line. The Universal Synchronous/Asynchronous Receiver Transmitter (USART) adds synchronicity, the communication between two UART devices is bidirectional. The protocol can be simplex, half or full duplex. The useful data length is typically between 5 and 9 bits. Some UART peripherals in SoCs support many protocols like SPI or RS-485 [173]. Some solutions like

[119] enable a bridge between UART and SPI. A start bit and one or multiple stop bits delimit the frame as showed in Figure 3.12. There can also be a parity bit to check the received data. In case of synchronous mode, there is no start, parity or stop bits. These fields cannot be handled naively in the generic payload.

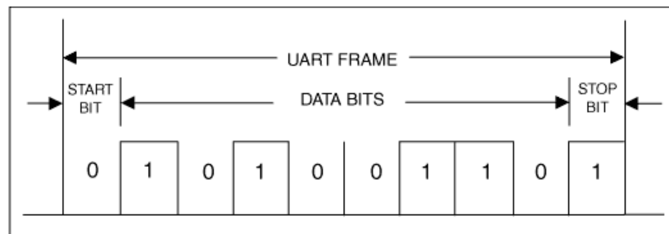


Figure 3.12 – UART frame, 8 bits, 1 stop bit

The Figure 3.11 shows how an UART protocol can be modelled with the TLM-2.0 standard. Transmission and reception is done using two specific sockets and the generic payload is extended with UART fields like the parity bit. However, as the UART protocol is not a memory mapped protocol, the UART frame does not fit well with the TLM-2.0 standard generic payload. Moreover, fields like `address`, `byte_enable` or `DMI` do not make sense for an UART communication. Finally, for one to one protocols, communications need a payload transfer between master and slave devices. UART communications between the sender and the receiver are bidirectional. Moreover, no DMI feature is associated with UART data transfer. The TLM-2.0 standard is not compliant or optimized with these fields.. The section 3.5 details some solutions to solve this issue.

3.4.1.2 “One to Many” protocols

“One to many” protocols extend the previous described features. They involve one sender and many receivers, which requires addressing features as described in Figure 3.13. For instance, the RS-485 standard is a one to many protocol. It is similar to RS-232 except it uses a different physical layer. It also contains a master/slave relationship adding routing. Hence, some aspects of RS-485 standard are in layer three as well as OSI model layer two. A kit supporting RS-485 would have to consider ’routing’ between multiple masters/slaves on the same ’wire’. RS-485 transactions contain both address and data information, while a one to one implementation would only need to contain data. If used with Modbus protocol [120], each peripheral has an address. In that way, the routing is done inside the frame through an address field.

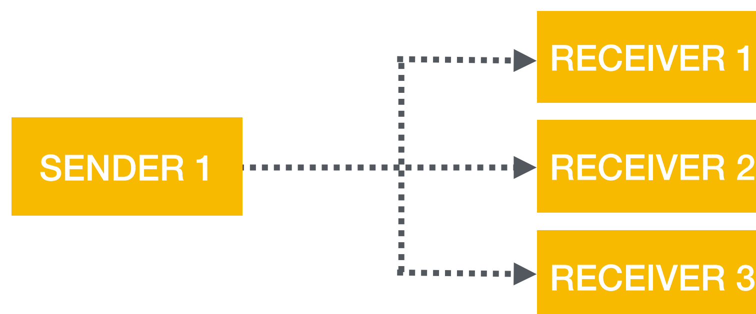


Figure 3.13 – One to many protocol scheme

Another protocol that has some of the same feature requirements is the SPI protocol. It is a synchronous protocol used for communication between multiple devices like memories, sensors,... It is a master to slave protocol. A Chip Select (CS) mechanism enables routing transaction to the right final slave(s). Each node is either a master or a slave. The communication is full duplex. In this way, once set up, SPI is normally a point-to-point protocol; though there are modes in which transactions can address multiple slave devices at the same time.

Viewing SPI through the prism of the OSI layer model, routing information should be handled in layer three. However, it is by no means clear that the routing mechanism actually forms part of the protocol. Indeed, the SPI frame is not standard and is specific for each device. If we will consider a simple SPI connection in which a master enables a single slave, within a TLM context, this can be adequately modelled as a point-to-point connection. The socket binding enforces the topology of the system and there is no routing as such. In the case of a controller, the controller itself has a bus interface. “Routing” happens within the controller. The controller itself sets up multiple SPI interfaces to the various slave nodes. In this case, the controller is acting as a sort of router and bridge.

Commonly, data are first buffered in a register with a limited size and the transaction is sent to the SPI connections. This size can be a non multiple of a "byte". However, the current TLM-2.0 standard only support the transaction length as a number of bytes. Indeed, the generic payload includes a ’size’ field which is the number of bytes implicated in the transaction. SPI data can be sent on four different clock modes. It is necessary to ensure that this mode is the same on each side. In addition, like UART protocol, the SPI protocol can be implemented in software using a GPIO controller.

3.4.1.3 “Many to Many” protocols

In this section, many to many protocols are highlighted. These protocols involve many senders and many receivers as described in Figure 3.13. For instance, I2C protocol is a many to many protocol. It is half duplex and involves the communication between multiple nodes. A I2C node can be either a master or a slave or a multi-master. A multi-master can talk at the same time to multiple slaves. As multiple nodes can potentially send a frame at the same time, an arbitration mechanism is necessary. Arbitration is managed by each master device that has to wait for the availability of the bus. More details on the protocol are available in the Appendix A.2.2.1. The communication is based on two connection lines: Serial Clock Line (SCL) and Serial Data Line (SDA).

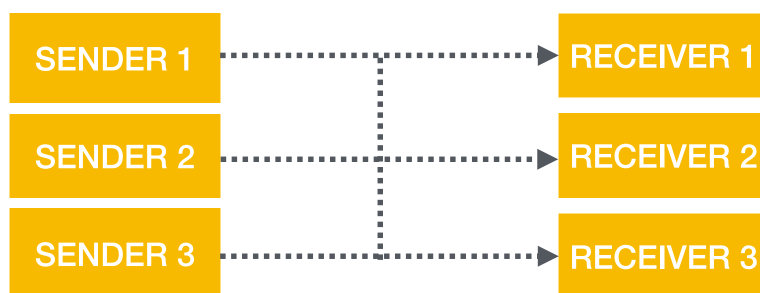


Figure 3.14 – Many to many protocol scheme

Each node on the I2C bus is identified by one address or multiple addresses stored on a range from 7 to 10 bits. A communication between nodes consists of multiple frames as presented

in Figure 3.15. Contrary to the SPI protocol, I2C frames are standard. They contains address, control content and data information in packets. Transferred data are stored on 8 bits. At OSI layer 2, there are frames for address, control and data, with start and stop phases. Collisions cause the completion times of some frames to be prolonged. To detect collisions, an I2C implementation has to find overlapping frames by using their start times and to check that their data elements are not identical. According to the I2C standard, the first master that attempts to write a 0 while another master writes a 1 looses the bus arbitration. I2C also has the concept of a frame acknowledge, pause and restart. These are all OSI layer 2 features, and should be modelled using phases in TLM. These phases can occur during the frame transmission, and may abort the frame or the packet.

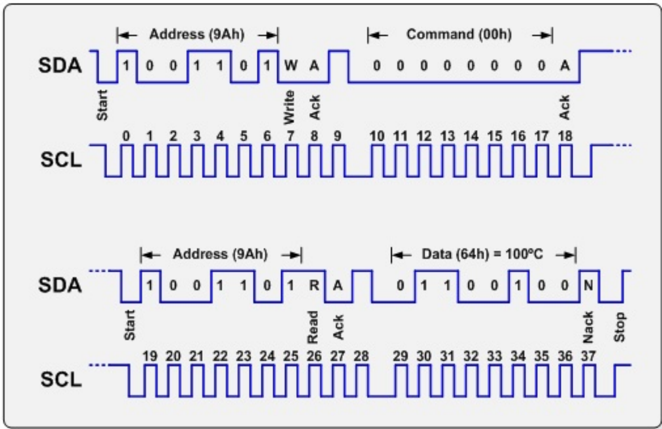


Figure 3.15 – A I2C frame structure

The CAN bus is another half duplex many to many protocol, also called a broadcast protocol, like a common memory mapped bus. Each node can be a master or a slave and there can be multiple masters. Node exchange information using frames or extended frames as showed in Figure 3.16. A frame is composed of a message identifier to differentiate potential recipient(s); the data and related data fields; a error detection mechanism based on the Cyclic Redundancy Check (CRC); and frame delimiters.

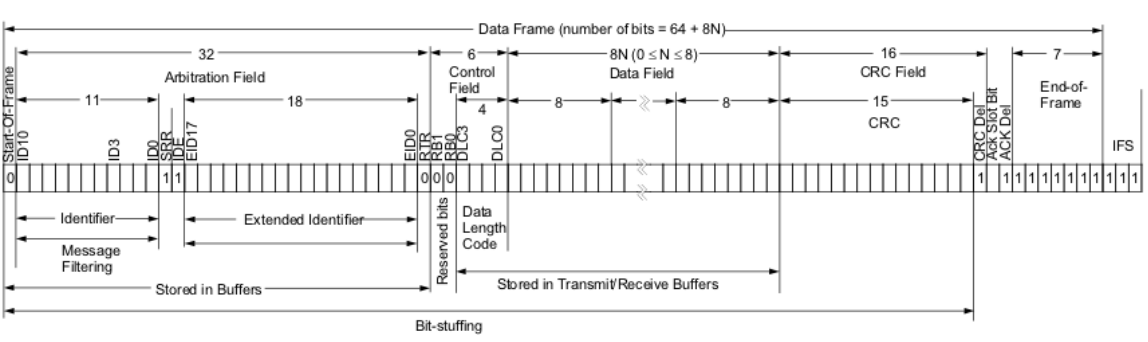


Figure 3.16 – A CAN extended frame structure

As multiple nodes can be masters, collisions can happen if multiple messages are sent at the same time. In that case, the message identifier is used as a priority field. The highest message identifier is distributed first A.2.2.2. The identifier field is used by a routing mechanism which sits as OSI layer three. In terms of modelling, routing can of course happen as it does in hardware via a broadcast. However, it is computationally much more efficient to use a “router” to model the bus medium itself. In this way, it only passes transactions to the correct node.

As discussed in the analysis of the protocols and depending of its nature, multiple ports of protocols can be interconnected together. Ethernet can be interconnected with a switch or a router. CAN ports are interconnected through CAN lines and transceivers. Interconnection is modeled with a TLM router. Routers enables TLM socket interconnections. They route packets according to the rules defined for each protocol. A TLM router is more than a simple packet forwarder. It can handle logic. For example, in order to speed up the delivery of packets, the router can decide to directly forward a transaction to the right receiver instead of broadcasting to all receivers if the protocol is a broadcast protocol. In that case, the router has to be ability to decode the transaction in order to know the right receiver. The decoding is specific to each protocol. For this reason, a router should be defined for each protocol.

Data sits in the transaction contents as they are an OSI layer three feature. But frame element like CRC error have no place in the transaction. The CAN data frame is the general frame used to send data to another node. It seems perfectly reasonable to have a frame identifier in the transaction and a union of the components that each frame carries. As the size of a CAN frame is not fixed, the data can be from 0 to 8 bytes. The TLM-2.0 standard data mechanism can be used to handle the data itself. Only one frame is considered per transaction. So at OSI level 2, there is nothing more than the single frame which can be modelled at AT with two phases.

3.4.2 Modeling requirement summary

Table 3.1 – non memory mapped protocols in regard of TLM

Protocol	Signal	RS-232	RS-485	SPI	I2C	CAN
Family	One to One		One to Many		Many to Many	
Payload	Ø	data_ptr, data_length	data_ptr, data_length, address	data_ptr, data_length, address	cmd, data_ptr, data_length, address	frame_type, identifier, data_ptr, data_length, address
TLM Phases	start, stop	start, stop	start, stop	start, stop	start, stop, ack, pause, restart	start, stop

The Table 3.1 sums up some protocols. In this section, a number of protocols have been examined with reference to OSI layer model. The transaction and associated phases have been identified. As the OSI model is an international standard applicable to communication protocols, it decreases ambiguity between designers about how to define the transaction. It fits with the existing Accellera TLM-2.0 standard generic protocol designed for memory mapped buses. Moreover, it seems to work for a wide cross section of other interfaces. Therefore, it seems a good approach for determining what should be identified as a transaction. However important a transaction is, there are unfortunately other issues with the TLM-2.0 standard which also enables for ambiguity when building a new interface. The Section 3.5 proposes an evolution of the standard to facilitate the building of new protocols for TLM.

3.4.3 Interconnection

A TLM-2.0 standard router forwards a transaction to the right receiver based on the bus address of the transaction. This implies that it needs to know what is the address range of each connected receiver in order to forward to the right one. This part is currently not standardized. Each implementation can use its own mechanism that decreases interoperability. As briefly exposed in Section 3.6, CCI parameters can be used to enhance the protocol negotiation interoperability. Indeed, they could be used in the TLM sockets to expose their position on the memory map. The same logic can be applied to non memory mapped protocols. SPI devices could expose their chip select number. Similarly, ethernet devices could expose their MAC address.

3.4.4 Conclusion

This section has detailed the non memory mapped protocol commonly used in SoCs. Analysis of one to one protocols, that included the simplest protocols, showed that the current TLM-2.0 standard is not natively supported. This assertion has been confirmed with one to many and many to many protocols. The fields included by default in the generic payload are not adapted for all non memory mapped protocol. Some are useless for a certain category of protocols (e.g. the DMI) but useful for others, others are inappropriate (e.g. the bus width) but required for others and a few are incompatible (e.g. the length of the data) to support properly a communication. There isn't bidirectional socket support, but bus width is useful for protocols. In resume, this analysis has help inform the need to build an improved version of TLM-2.0 standard in order to support non memory mapped protocol protocols.

3.5 Proposed improvements of TLM

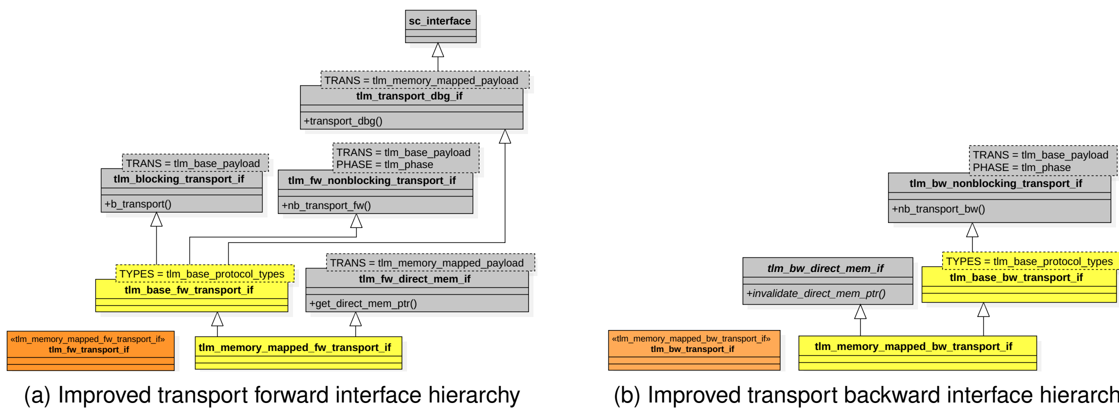
3.5.1 Introduction

In order to add support for new protocols which are not directly compatible with bus "like" protocols, it is necessary to define a completely new payload. Moreover, the generic payload class includes a mechanism to register extensions. This mechanism is hardly linked to the generic payload class and cannot be easily re-used. The same issue happens with phases for non blocking transports. The TLM phase mechanism can be extended. The transaction and the phase are not the only ones that are required. TLM-2.0 standard defines forward and backward interfaces that have been elaborated to be compatible for memory mapped protocols.

Related works also presented some requirements for sockets update. The TLM-2.0 standard does not provide native support for bidirectional sockets, though the standard allows the inheritance of both initiator and target sockets in order to build a bidirectional socket, which is less convenient. As the implementation of the bidirectional socket is not provided by the standard and so left to the designer, interoperability is decreased.

3.5.2 TLM Transport

Each TLM transport class is based on two interfaces : one for forward and another one for backward communication. The forward interface inherits from four interfaces all containing exclusively pure virtual methods. These classes contain methods for blocking/non blocking transport, for DMI and debug. This means that the DMI and debug methods must be implemented even if they are not applicable to a protocol. For the RS-232 and RS-485 protocols, the DMI mechanism makes no sense. To solve this issue, a re-factor of the current TLM standard is proposed while maintaining backward compatibility. This is illustrated in Figure 3.17. Thus, non memory mapped TLM interfaces can be defined inheriting from a more basic class without DMI. An example usage for the UART protocol is presented on Listing 3.1.



(a) Improved transport forward interface hierarchy (b) Improved transport backward interface hierarchy

Figure 3.17 – Transport interfaces for memory mapped and non memory mapped protocols. Grey = existing. Yellow = added. Orange = modified.

The DMI interface is only necessary when it is suitable for the protocol concerned. For example, DMI makes no sense for signals, UART based protocols and CAN. On the other hand, the DMI interfaces do make sense for memory like nodes with an I2C. This protocol supports a sort of burst read/write. For the SPI protocol, as the frame content is not standard, DMI is hardly applicable as shown in Chapter 5.

The debug interface is always valuable and makes sense. It is not specific to a particular protocol or a specific property. Hence, it is recommend all interfaces inherit the debug functionality.

Listing 3.1 – UART forward and backward interfaces

```
1  class tlm_uart_fw_transport_if :
2          public tlm_base_fw_transport_if<tlm_uart_protocol_types> {};
3
4  class tlm_uart_bw_transport_if :
5          public tlm_base_bw_transport_if<tlm_uart_protocol_types> {};
6
7  class tlm_uart_transport_if :
8          public tlm_uart_fw_transport_if ,
9          public tlm_uart_bw_transport_if {};
```

3.5.2.1 Socket and binding

Those are new sockets and binding challenges to support non memory mapped protocols as in Figure 3.18. The current TLM-2.0 standard core interfaces pass transactions from initiators to targets in only one direction. An implementation of a TLM bidirectional socket is available in [75]. The implementation instances a pair of initiator and target sockets in a single top level socket. This implementation is considered as a good candidate for supporting backward compatibility and is aligned with the LRM. From a user perspective, it simply looks like a bidirectional TLM socket.

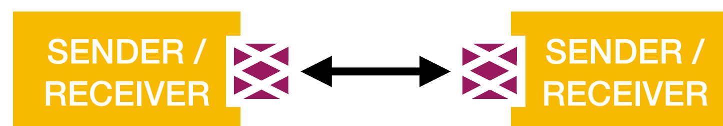


Figure 3.18 – Bidirectional socket scheme

The base socket classes of the TLM-2.0 standard (initiator and target) have a notion of bus width as a template parameter (`BUSWIDTH`) with a default value equal to 32. These classes also implement a method to retrieve bus width. This is a significant issue. Protocols like signals, UART, CAN... have no notion of bus width. Different `BUSWIDTH`s can lead to binding issues if the value is not the same on both side even if the value itself is not relevant. The proposal is to refactor this providing a more generic socket class and removing related bus width fields and methods called `tlm_base_generic_initiator_b` as illustrated in Figure 3.19.

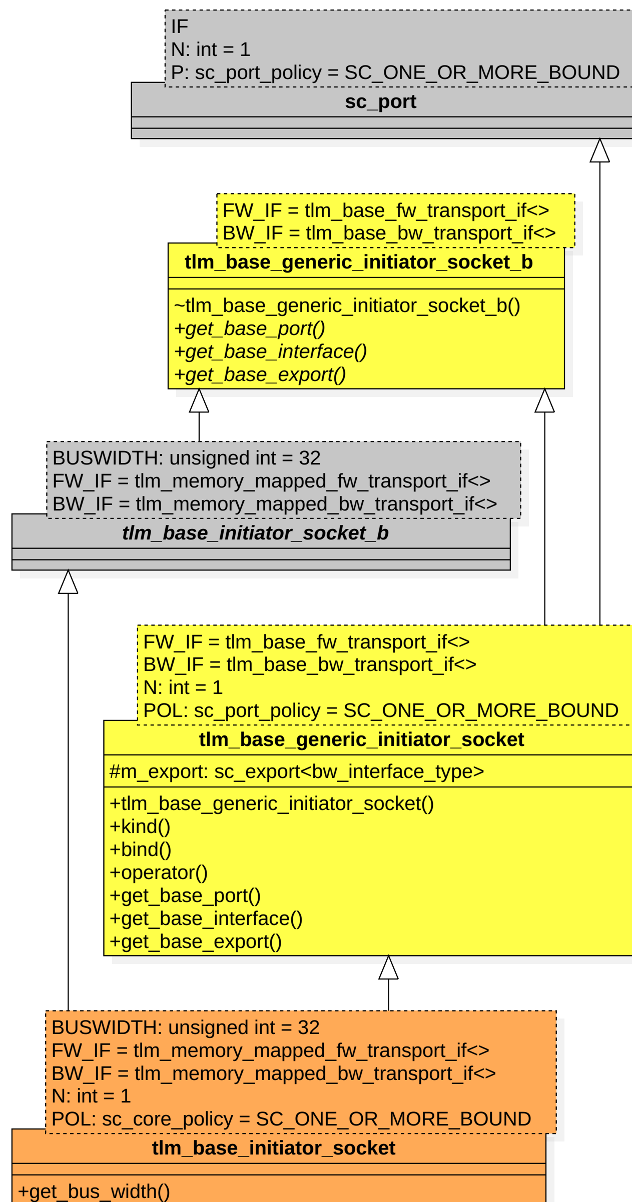


Figure 3.19 – Generic TLM initiator socket with backward compatibility

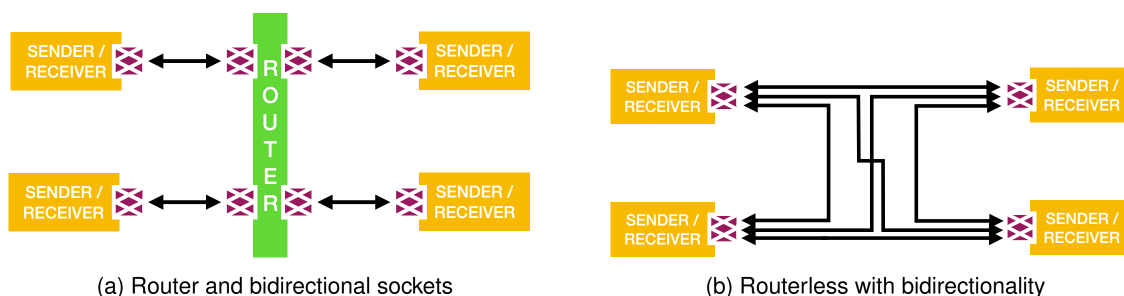
Thus, convenience sockets for non memory mapped protocols can be implemented inheriting from `tlm_base_generic_initiator_socket` without the redefinition of the port related methods. The TLM-2.0 standard class has been left for backward compatibility with the new proposed base classes. An implementation of the UART socket, a bidirectional protocol, is detailed with only few lines on Listing 3.2.

Listing 3.2 – UART bidirectional socket

```
1  class tlm_uart_socket :
2          public tlm_bidirectional_socket<tlm_uart_protocol_types,
3                   tlm_uart_fw_transport_if, tlm_uart_bw_transport_if>
4  {
5  public:
6      typedef tlm_bidirectional_socket<tlm_uart_protocol_types,
```

```
7               tlm_uart_fw_transport_if, tlm_uart_bw_transport_if>
                  base_type;
8
9         tlm_uart_socket():
10                base_type()
11        {}
12
13        explicit tlm_uart_socket(const char* name):
14                base_type(name)
15        {}
16
17        /* ... */
18
19        ~tlm_uart_socket() {}
20  };
```

The LRM for the TLM-2.0 standard encourages the usage of a router for data transfer between modules if multiple modules share the same data transfer layer like a bus. A router can be used at the AT and LT abstraction levels. At the AT level it is used to calculate collisions and delays on the transport medium. In the case of "One to Many" and "Many to Many" protocols, something in the protocols has to indicate the final destination of the data transfer. This field can be an address. It can also be a signal that can be used to build the transaction with an address like with the SPI protocol. "Addresses" are relatively generic. Hence it is natural to use a router as mentioned in Section 3.4.1.3 and presented in Figure 3.20, eliminating the need for each target to check the relevance of each transaction. For completeness, "routerless" simulation of broadcast protocols is useful if the arbitration is not required. In this case a bidirectional and multiple socket is required.



(a) Router and bidirectional sockets

(b) Routerless with bidirectionality

Figure 3.20 – Bidirectionnal sockets and "One to Many" / "Many to Many" protocols

3.5.2.2 Payload

The extension mechanism and other payload fields are in the same unique generic payload class. It is not possible to define a new payload without a duplication of some features of the original generic payload. This is not convenient as the generic payload gives us the start of a pallet of fields that can be re-used in other protocols, again increasing interoperability. A payload also defines the TLM responses status and commands. The current implementation defines a default enumeration for both. However, the enumeration does not cover the needs of each protocol. As noted above, it is likely to be protocol specific.

A re-factor of the payload in TLM is proposed. The idea is to separate payload fields and the extensions mechanism. The extension mechanism should form the basis of all payloads. The extension mechanism is exclusively defined in a new class called `tlm_base_payload` as detailed

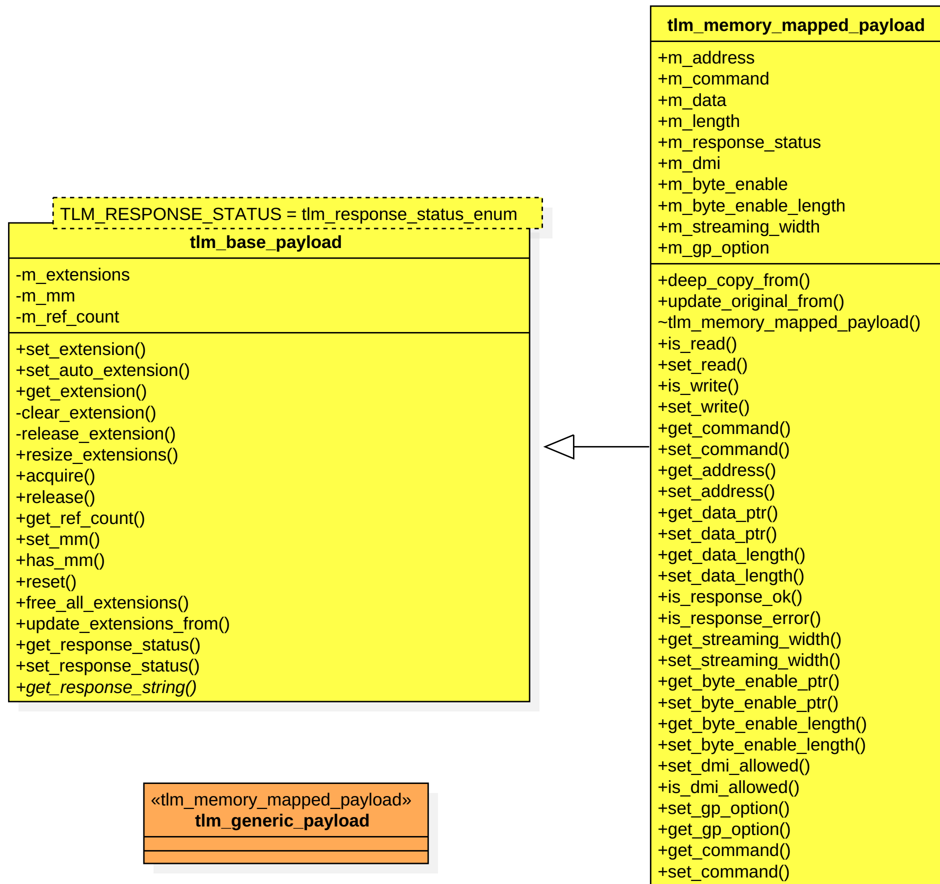


Figure 3.21 – Base payload description. Yellow = added. Orange = modified.

in Figure 3.21. It enables specific payloads to inherit from this class. Then, for an I2C payload, a `tlm_i2c_payload` class is defined inheriting from `tlm_base_payload`. It only reuses the fields of the base protocol as in Figure 3.22. The mechanism by which fields themselves can be standardized and re-used, probably using a macro approach like the extension mechanism is left for future work.

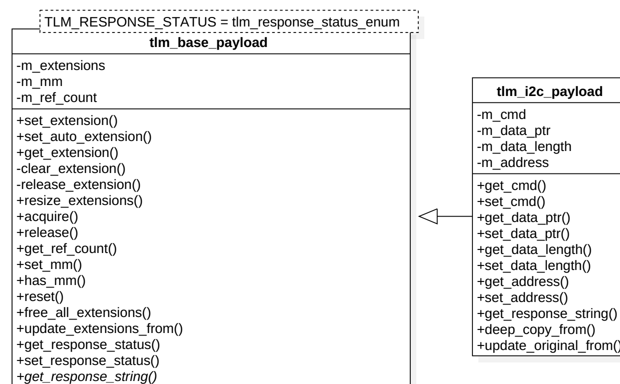


Figure 3.22 – TLM I2C payload inheriting from TLM Base Payload

Some issues with the terminology associated with the TLM-2.0 standard have been noted as quoted previously. Hence, a new memory map payload with a name different than “generic” is

proposed : `tlm_memory_mapped_payload`. For backward compatibility, a `typedef` between the new definition and the old payload name has been provided.

TLM-2.0 standard defines the “generic payload”. Considerable care was taken to allow many simple buses protocols to be modelled at least at the LT level using this “generic payload”. It does not explicitly cover many bus features. But, it enables an extension mechanism to enable those to be modelled. It is recommended in all protocol classes.

In the case of UART, parity is a layer two feature, and should be modelled as part of the transaction. Parity errors can be generated by transmission collision. These would be handled by requesting re-transmission on a parity error. This may be handled in software or hardware depending on the UART protocol and/or system configuration. A suitable TLM-2.0 standard implementation of an UART can be found in [75]. It provides an initiator and a target bidirectional serial socket and also a payload for UART communication that includes RS-232 protocol. The parity bit has been left in the payload. Though the parity bit is correctly part of the payload, the implementation has been revisited. An implementation of the UART payload, using the base payload, is given in Listing 3.3.

Listing 3.3 – UART payload

```
1  class tlm_uart_payload : public tlm_base_payload
2  {
3  public:
4      /// Default constructor
5      tlm_uart_payload()
6              : tlm_base_payload()
7              , /* ... */ { }
8
9      explicit tlm_uart_payload(tlm_base_mm_interface* mm)
10             : tlm_base_payload()
11             , /* ... */ { }
12
13 private:
14     /// Copy constructor
15     tlm_uart_payload(const tlm_uart_payload& x)
16             : tlm_base_payload(x)
17             , /* ... */ { }
18
19     /// Assignment operator
20     tlm_uart_payload& operator= (const tlm_uart_payload& x)
21     {
22         tlm_base_payload::operator=( x );
23         /* ... */
24     }
25
26 public:
27     // non-virtual deep-copying of the object
28     void deep_copy_from(const tlm_uart_payload & other)
29     { /* ... */ }
30
31     /// Destructor
32     virtual ~tlm_uart_payload() {}
33
34     /* ... */
35
36 private:
```

```
37        tlm_uart_command                m_command;
38        unsigned char*                  m_data;
39        unsigned int                    m_length;
40        tlm_uart_response_status        m_response_status;
41        bool*                           m_parity_bit;
42        unsigned char                   m_stop_bits; // [1,3]
43        unsigned short                  m_enable_bits;
44        unsigned int                    m_baudrate;
45    };
```

3.5.2.3 Phases

TLM default class for phases is named `tlm_phase`. Methods of this class are based on the default enumeration `tlm_phase_enum` which defines four phases. It is currently possible to add new phases on top of the default one through a macro. Default phases are already really basic. They can be applied directly to non memory mapped protocols. In the case of UART, frame fields are OSI layer two properties. The should be modelled as timing events (phases) by marking the start and end of a frame. Again it is suggested that, as far as possible, phase names are kept within a pallet.

3.5.3 Conclusion

Finally, TLM-2.0 standard focuses on bus communications. However, it does not facilitate the definition of the customization level as it is left to designer choices. Moreover, the analysis of the current architecture of TLM-2.0 standard showed that the extension of the TLM-2.0 standard is not natural and can lead to code duplicate. For instance, the modeling of the UART protocol is summarized in Figure 3.23.

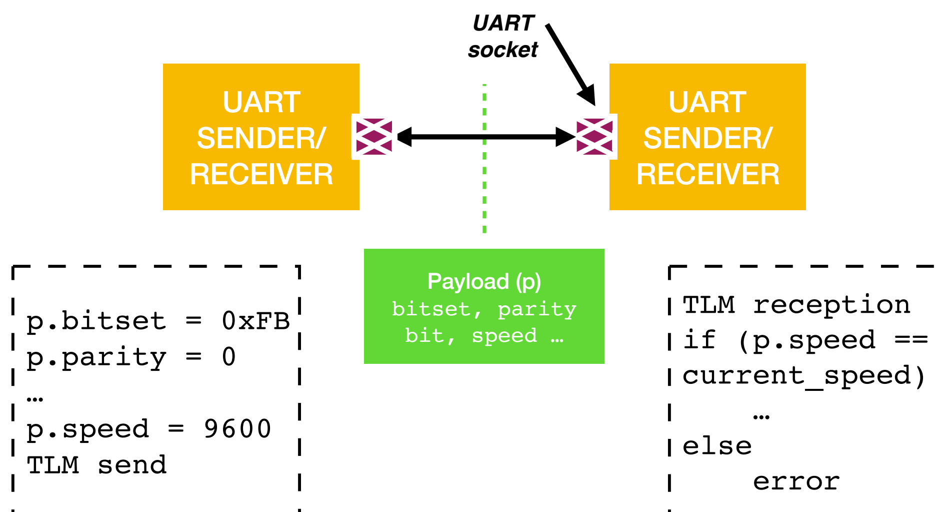


Figure 3.23 – UART protocol with the improved version of TLM

This section has detailed a proposal to clean and refactor TLM-2.0 standard without breaking the backward compatibility. It enhances code re-used and decreases duplicated code while not affecting performance. However, the enhanced version of TLM-2.0 standard does not solve all

issues. While the architecture solves the static part of a protocol - binding, payload, phase -, it does not solve the dynamic part like the clock frequency, the baud-rate, the clock edge mode... The use and the integrated of CCI with TLM, presented in the previous chapter, is given in the next section.

3.6 Protocol configuration check with CCI standard

3.6.1 Introduction

Transactions are not only defined by the exchange of data but also according to external parameters. For example, it can be the clock frequency, clock edge, baud-rate... as in Figure 3.24. These parameters are the meta-data of the protocol. Indeed, communication protocols can, themselves, be parametrized, just as any other IP. One use of the extension mechanism is to pass so called 'ignorable' and 'non-ignorable' extensions as a mechanism to negotiate protocols and allow some degree of parametrization. For instance, a target that supports a security zone may offer an ignorable 'zone' extension. A master that also supports that extension will be able to rely on it's implementation. In the case of a more generic initiator, the target will have to revert back to 'generic' non-secure-aware access. Some protocol extensions require no extra data to be transmitted, and extensions to signal the presence of these features would carry no meaningful data, except for their presence itself. This form of protocol negotiation is typically static, and using the extension mechanism can bloat the transaction, increasing the memory footprint and wast resource.

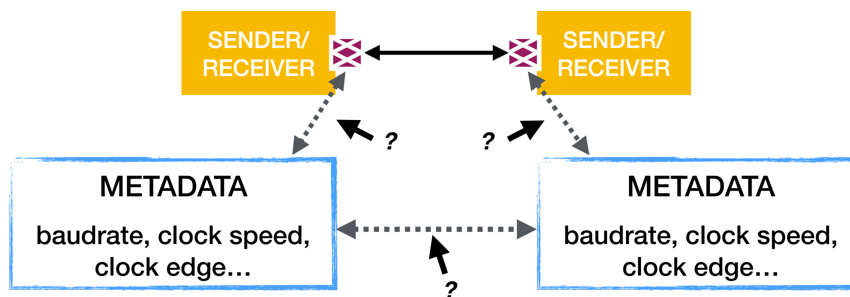


Figure 3.24 – Example of protocol meta-data exchange

In order to support these parameters, a first solution could be to include them in the payload of the protocol. However, it implies a check in each transaction to ensure the configuration of the protocol between the sender and the receiver. If the number of parameters is high, it can reduce the performances. Besides, these parameters are not changing too much for each transactions and mainly configure one time during the initialization. CCI is standard for parameterising IP, the next section will explain how it can be applied to communication protocols.

3.6.2 CCI standard applicability

The initiator or the target socket class has to be extended in order to add new attributes as CCI parameters. The Listing 3.4 shows an implementation example of two different CCI parameters in a TLM SPI socket.

Listing 3.4 – Example of CCI parameters in a TLM socket

```
class tlm_spi_socket : public tlm_bidirectional_socket<
  tlm_spi_protocol_types,
                    tlm_spi_fw_transport_if, tlm_spi_bw_transport_if>
{
public:
    typedef tlm_bidirectional_socket<tlm_spi_protocol_types,
                    tlm_spi_fw_transport_if, tlm_spi_bw_transport_if>
                      base_type;

    tlm_spi_socket():
        base_type(),
        m_clock_edge_mode(std::string(std::string(name()) + ".
          clock_edge_mode"), 0),
        m_clock_frequency(std::string(std::string(name()) + ".
          clock_frequency"), 3),
        /* ... */
    { /* ... */ }

private:
    cci::cci_param<bool> m_clock_edge_mode;
    cci::cci_param<unsigned int> m_clock_frequency;
    /* ... */
};
```

Two new private attributes are added to the class. They are the meta-data of the protocol. In order to track and also check the configuration of meta-data on each side of a communication, the other side socket will register a callback in order to get notified of any meta-data change. This enables dynamic configuration correctness checks. Contrary to adding of new (extension) fields in the TLM payload, the CCI parameter, reduces the number of checks thanks to callback checking. Configuration check of the protocol can be cached and is only updated on configuration change, notified by the callback mechanism.

3.6.3 Protocol configuration check

If the meta-data of a protocol is embedded in sockets as showed in Figure 3.25, a protocol negotiation is necessary. Basically, the negotiation can happen on initialization, or during the simulation if the meta-data change. Meta-data can be configured through registers. In that case, parameters have to be updated for each register update of the meta-data. Finally, the proposed solution for protocol negotiation consists in the use of CCI parameters to track and check the meta-data compatibility between each side of the transaction.

Contrary to the TLM-2.0 standard extension mechanism, CCI meta-data enables tracking the configuration of the sockets using the CCI API. In the case of an IRQ, the IRQ number is available as a CCI parameter number and can be set during the initialization. In the case of UART, the baud-rate, stop bit, and the parity bit configuration can also be exposed as a CCI parameter used as meta-data. Finally, CCI parameters can also be used to get the address of a module on the memory map

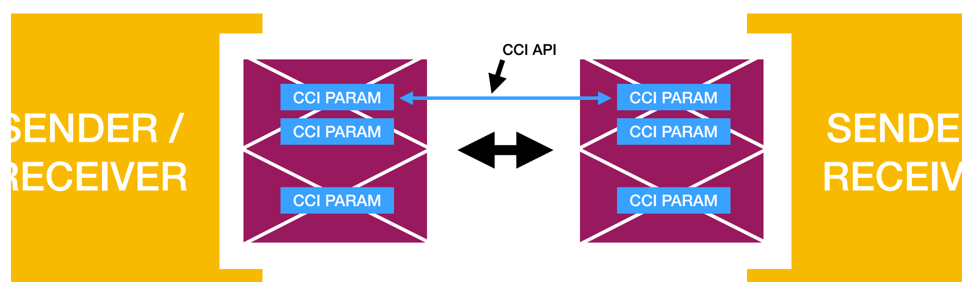


Figure 3.25 – Protocol meta-data embedded in sockets

3.6.4 CCI meta-data interoperability

If CCI parameters are present in sockets, the interoperability has to be considered. If both side of a transaction using sockets want to exchange parameters, they need a mechanism to do so. As quoted in Chapter 2, the name is the universal way to retrieve parameters from the broker. The name as well as the position in the hierarchy are required. If sockets want to get parameters from other sockets, then they need to know the hierarchical name of the parameter to use. Unfortunately, the current standard does not offer a mechanism to retrieve the name of the bound socket, neither the one sending the transactions. If the connection between sockets is a one-one connection, an access to the `name()` of the bound socket through the forward and backward interfaces looks possible. If a router is placed between the initiator and the final target, it is more complicated. The solution consists in adding more logic to the router.

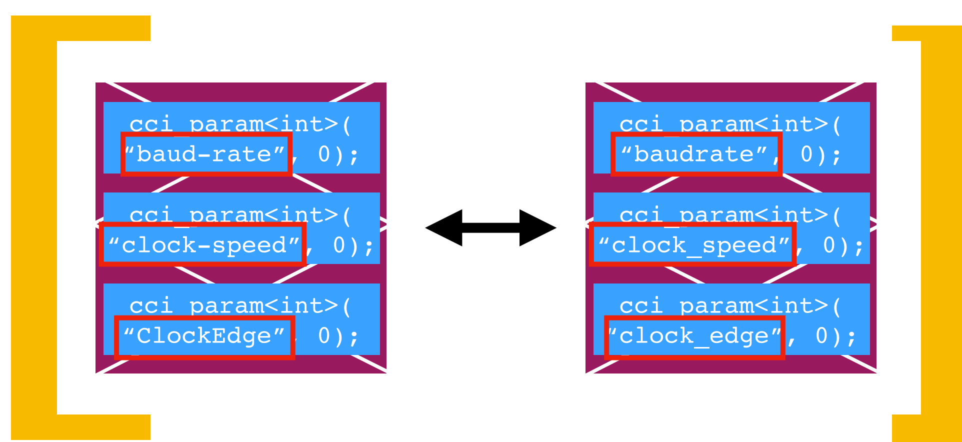


Figure 3.26 – Meta-data interoperability

The name of meta-data parameters for a protocol can be different, even if they model the same thing as showed in Figure 3.26. Without standardization of names, this mechanism will not be inter-operable. Even if the OSI model can help to extract the meta-data of a protocol, it does not standardize their name which is left to the designer. In general, it is recommended to use the same pair of sockets compatible with a protocol in order to ensure same parameters and so the interoperability.

3.6.5 CCI meta-data limitations

As CCI parameter names use the nearest SystemC module above the object in the hierarchy to add prefix names. Hence parameters in a socket are not prefixed with the socket name. This implies that names are mixed in the module namespace. A solution has been found and is illustrated in Listing 3.4. It prefixes manually the socket name. This solution defeats the easy instantiation of CCI parameters. Fortunately, it has to be done once by the protocol designer. Another option may be to improve the SystemC attribute mechanism, to use CCI parameters.

In order to improve the ease of CCI parameters with TLM sockets and other SystemC based object, an attachment mechanism similar to SystemC attributes should be added. It would simplify the code of designers avoiding unnecessary inheritance. We have opened discussions with the SystemC Language Working Group (LWG) and solutions are today under discussions. SystemC attribute is part of the standard for a long term and cannot be replaced in one go breaking the API.

An inheritance of a SystemC attribute type from a CCI parameter class is an envisaged solution. It would simplify the code of designers avoiding unnecessary inheritance. We have opened discussions with the SystemC LWG and solutions are today under discussions. SystemC attribute is part of the standard for a long term and cannot be replaced in one go breaking the API. An inheritance of a SystemC attribute type from a CCI parameter class is an envisaged solution.

3.6.6 Conclusion

In conclusion, CCI parameters attached to a TLM socket could offer new opportunities for transactions. It splits cleanly the static and the dynamic part of the transactions. However, the current implementation is not without issue and its integration in a TLM socket is currently not straightforward. An improved approach for the interoperability should be discussed in the SystemC LWG. The modelling of the UART protocol and the CCI standard is illustrated in Figure 3.27.

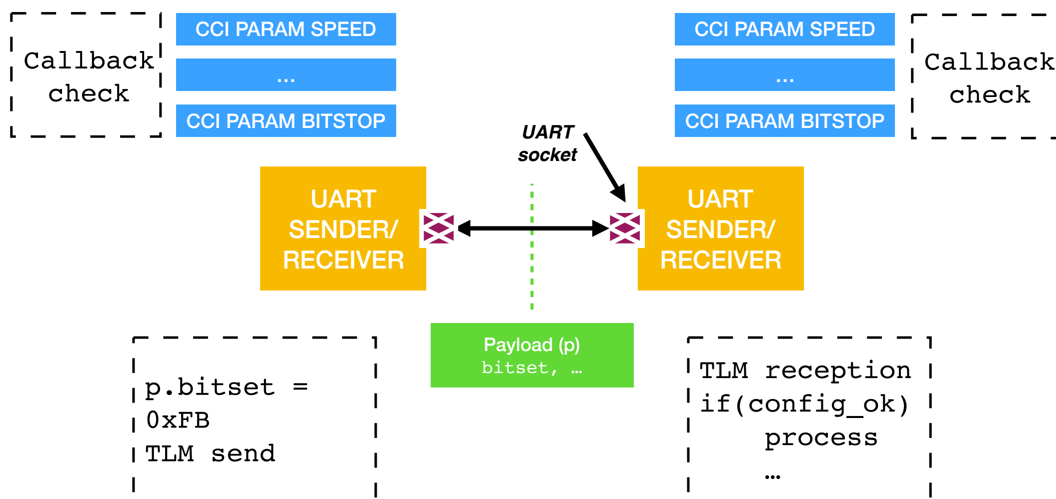


Figure 3.27 – UART protocol example with the improved version of TLM and CCI

3.7 Future works

3.7.1 Software emulated protocol

Non memory mapped protocols available in SoCs are commonly implemented inside hardware IP. However, non memory mapped protocols can also be software emulated. The software drives pins of a GPIO controller as showed in Figure 3.28. In this example, pins are then connected to a UART device. This solution is commonly used to increase the flexibility of a chip.

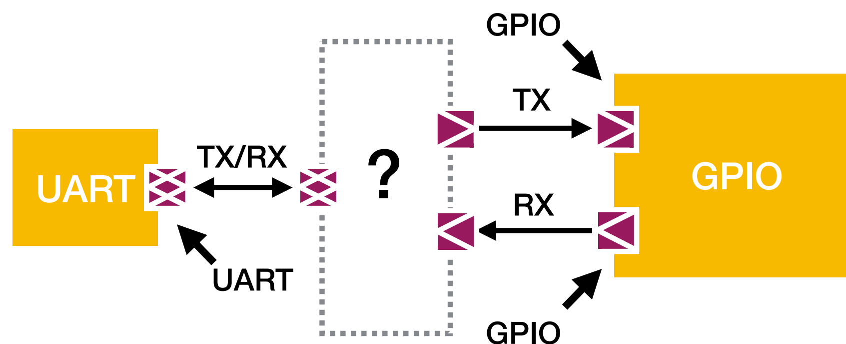


Figure 3.28 – TLM and software protocol

Software emulated protocol cause communication abstraction level issues. On the one hand, GPIOs are driven by the software at a low level while the UART device is using a TLM UART socket at another abstraction level. Even if the emulated protocol cannot be directly handled, some mechanisms are required to support it. A solution is examined in Appendix A.2.5 as well as its limitations.

3.7.2 Pin functions

As mentioned briefly in the previous sections, the same pin-out of a chip can be used for multiple protocols (cf Figure 3.9). The configuration of the peripheral driving the pins is mainly done with the software. This issue has not been entirely considered in this work but some candidate solutions have been studied.

- If the protocol used by the pin is know and fixed during the execution, a specific TLM socket implementing the protocol can be used. This solution is not perfect as the model is fixed according to a specific execution case.
- If the protocol used is unknown during the design or can change during the run time, the addition of all possible protocol sockets can be used. This implies some potential binding policy issues about unbound sockets if some are not used. Indeed, if the port policy of the socket does not enable to let unbound the socket, the simulation never starts. Furthermore, if sockets can be left unbound, it can potentially lead to undefined behaviours and is error prone.

Finally, the GPIO controller can also be used to drive a part of a protocol. In a case of the SPI, a GPIO can be used to drive the CS. This issue cannot be managed easily by the designer as it is driven by the software. Finally, the multiplexing issue has not been resolved due to time constraints but deserves to spend more time on it according to its scope.

3.8 Conclusion

TLM-2.0 standard focuses on memory mapped protocols and does not manage non memory mapped protocols. The transaction notion is not clearly defined and its understanding is left to the designer. This leads to confusion and interoperability issues. In this chapter, the modeling of communications in virtual platforms has been studied. Related works show that there are currently no proper and interoperable solution to model non memory mapped protocols with the TLM-2.0 standard. An analysis of TLM-2.0 standard has showed that the architecture is not build to be easily extendable.

My contributions concern a more robust and reusable architecture for TLM. Results have been featured in [60, 61]. A protocol negotiation solution has been proposed using the CCI standard. Finally, the question of the software protocol has been discussed. While SystemC and TLM aim to model hardware communication, the software emulation protocol communication cannot be ignored. A blue print has also been elaborated and is available in the Section A.2.4 of the Appendix.

A clear solution for modeling almost all communications in virtual platforms is proposed. It enables the easy construction of new protocols for TLM, improves their interoperability and their maintainability. A clear distinction between the protocol configuration and the exchanged data is also proposed with CCI. This solution opens up new opportunities like the simulation of multiple platforms connected by non memory mapped protocols.

Chapter 4

Parallelism in SystemC/TLM

***

4.1 Introduction

The evolution of application complexity in embedded systems requires multicore solutions, both homogeneous or heterogeneous in nature. Increasingly more cores are integrated into a single chip as in the SoC [10] illustrated in Figure 4.1. It is a NXP SoC used in many application domains. This SoC includes four ARM [12] A53s, two ARM A72s, and two ARM R5s. The adoption of multicore systems brings a wide range of potential benefits. Low power cores can be used for light tasks to reduce the energy consumption. Powerful cores can be used for more challenging tasks. Depending on the application, various cores can communicate between themselves using one or multiple buses.

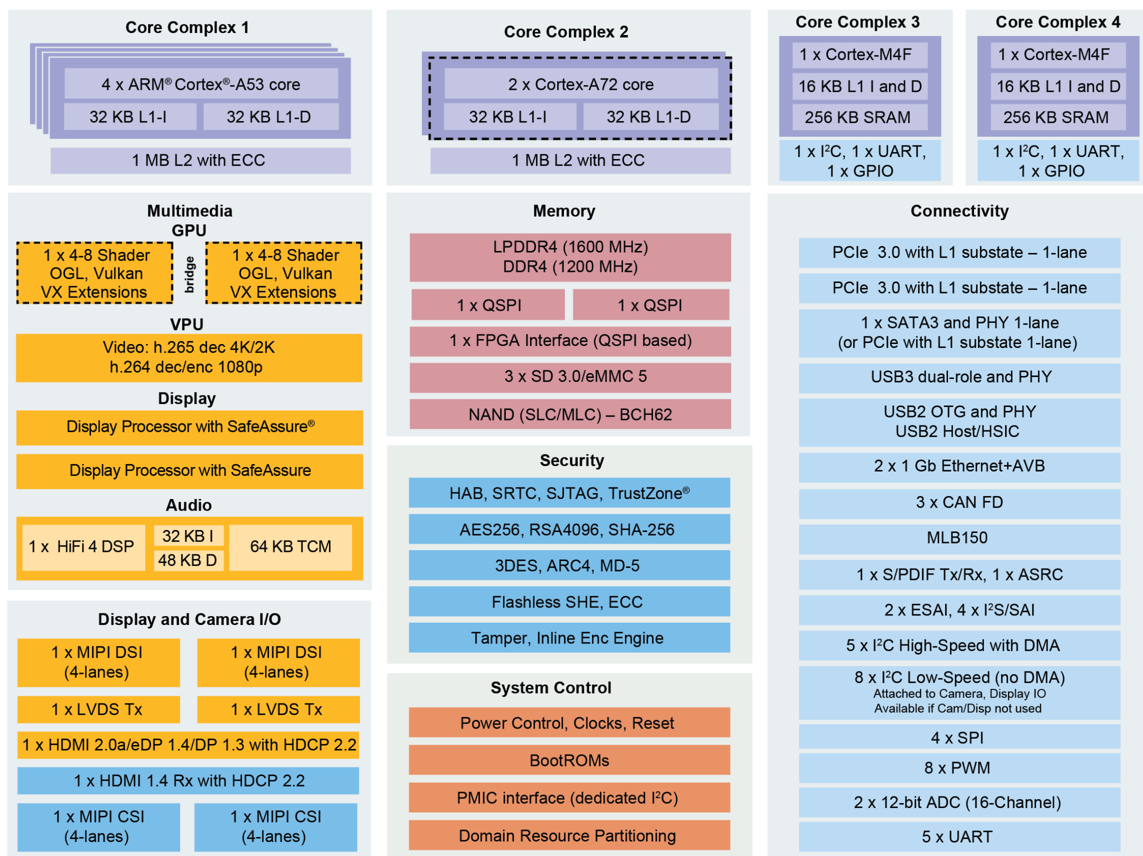


Figure 4.1 – NXP i.MX 8 heterogeneous SoC that contains ARM A53, A72 and M4F cores [131]

The number of cores in a SoCs must be reflected in the corresponding virtual platform. This

generates new issues in terms of design, evaluation and debugging of virtual platforms. Indeed, as explained in Chapter 1, the SystemC simulator runs processes in a sequential manner and in a single thread. As a result, the simulation performance of a virtual platform decreases as new models (e.g. processes) are added to the simulation kernel. The multiplication of CPUs models in virtual platforms impacts the simulation execution speed. A way to reduce simulation time of complex virtual platforms including many processor models is thus required.

To improve the virtual platform simulation execution speed, executing time consuming models in parallel to the SystemC kernel seems to be a valuable solution. Models can be run in different host threads, but within the same SystemC process or models could also be executed in different SystemC kernels. However, multiple conditions are required to ensure the data and time remains consistent.

First, the Section 4.2 introduces the requirements to run models in parallel and the related works. The aim is to better balance the load to speed up the simulation. Next, Section 4.3 presents two asynchronous mechanisms. They aim is to facilitate parallel simulations of SystemC models. Then, synchronization solutions based on asynchronous mechanisms are studied in Section 4.4. Synchronizations based on TLM quantums and its impact are discussed. Different synchronizations solutions are evaluated in Section 4.5. Finally, a conclusion of the parallel TLM solutions is given in Section 4.6.

4.2 Related works

4.2.1 Requirements

The IEEE-1666 SystemC [5] standard enables systems, software and hardware designers to model complex SoC platforms [33, 52, 11, 172] thanks to a large set of language primitives. Many of these language primitives enable the synchronization of the data being modeling. Consequently, it is possible to model a platform such as one presented in Figure 4.2.

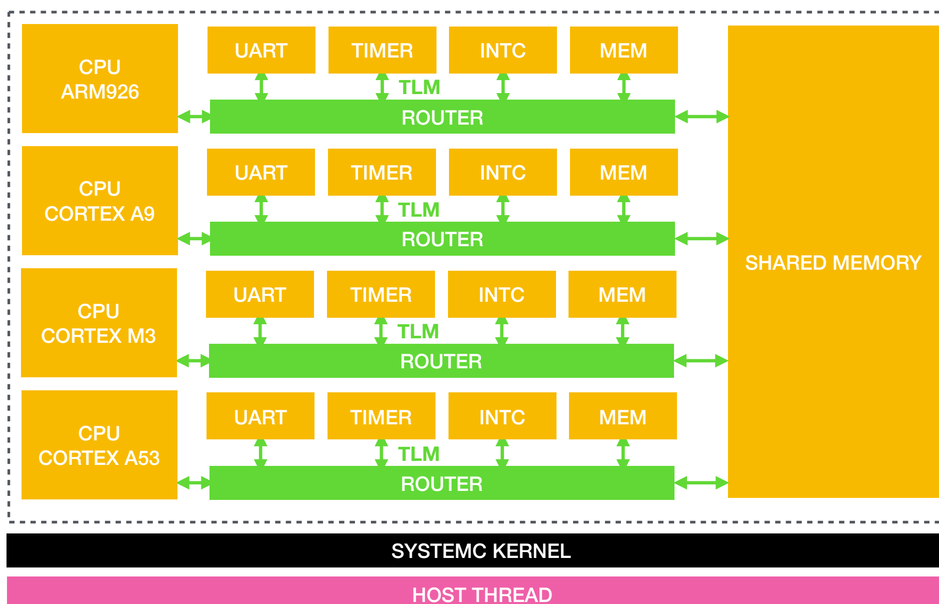


Figure 4.2 – Multi-core heterogeneous platform

This platform is composed of four heterogeneous processors connected to a shared memory through different buses. The simulation of such a model with the current version of SystemC would result in a mono-threaded slow simulation. There has been interest in distributing simulations across host threads, or multiple hosts, to further increase simulation execution speed as showed in Figure 4.3. The entire virtual platform is distributed over multiple SystemC kernels and different threads. However, in all cases, synchronization has to be done between different parts of the simulation in order to ensure time and data coherency.

In Figure 4.2, transactions through buses are modeled using TLM-2.0 transactions. The models abstraction level is LT. Then, to reduce the interaction and synchronization with the SystemC kernel, hence improving simulation execution speed, TLM-2.0 contains the notion of time decoupling and “quantum”. Thanks to the quantum, TLM-2.0 models can run a certain amount of time (the quantum) ahead of SystemC kernel simulation time before a synchronization has to occur. In other words, this means that each CPU, in Figure 4.2, initiator models, can potentially run with its own local time. This time can be ahead of SystemC kernel simulation time by a quantum.

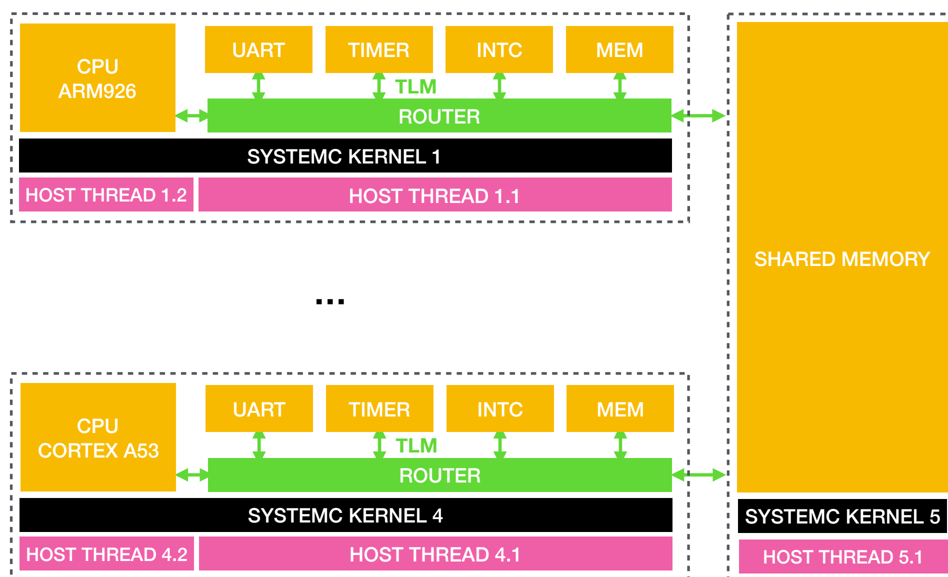


Figure 4.3 – Multi-core heterogeneous platform based on different SystemC kernels and threads

The benefits of TLM is to accelerate the simulations of SystemC virtual platforms. To facilitate the modeling of CPU, Instruction Set Simulators (ISS) can also be used instead. As a TLM initiator model, an ISSs can have its own local time. In which case, each ISS should stay synchronized with SystemC’s simulation time to be coherent. For instance, QEMU [143], an ISS which is both an emulator and virtualizer, implements its own base of time. However, neither SystemC kernel nor TLM-2.0 support multiple simulator synchronization natively. SystemC does not support natively the parallelism between multiple SystemC instances, a SystemC instance with multiple ISSs or a mix of both.

To parallelize SystemC simulations, different strategies have been proposed. A distinction between attempts to provide parallelism in or between SystemC simulations can be drawn. The synchronization mechanism itself, that is employed in order to enable that parallelism can also considered separately. In most related works, the synchronization mechanism is not fully detailed. The focus is on the parallelism. The aim of this chapter is to provide a generic and interoperable synchronization mechanism. Many researchers concentrate their efforts on the SystemC scheduler itself. They focus on synchronization between SystemC modules fully synchronized with

the SystemC kernel, often with SystemC channels. Other investigations examin parallelization between “sub-systems”.

Focusing on inter simulator parallelism, the aim is to facilitate the construction of different schemes on the top of a standard synchronization mechanism. For instance, the split between SystemC channels is reported by [190], while the split between TLM transactions is reported in [157, 139]. In our solution, there is no requirement that both ends of the mechanism are running SystemC kernel. The mechanism can be used between SystemC kernel and any other simulator with an independent notion of time. Furthermore, the mechanism must be suitable for standardization, and is intended to support future research efforts to define efficient inter-SystemC parallelism mechanisms. One application and one of the motivations are to support complex multi-core platforms such as that represented in Figure 4.2.

4.2.2 Parallelism inside a SystemC kernel

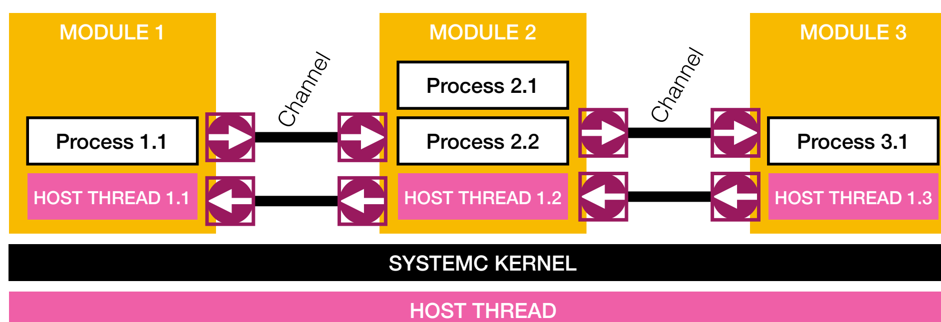


Figure 4.4 – Parallelism inside a SystemC kernel

The first parallelism strategy consists of taking the SystemC processes that can be fired in parallel in a single simulation environment into account. Thus, the challenge is to identify the ones that can be executed in parallel without violating the integrity. The authors of [136, 184] present a solution that includes multiple SystemC processes within a single SystemC simulation during the same delta cycle. The communication links between modules are used as a synchronization barrier. For instance, in example presented in Figure 4.4, the processes 1, 2 and 3 can be executed at the same time if there is no time constraint. However, due to the SystemC semantics for write channel access, a queue synchronization mechanism is necessary to avoid data inconsistency. In the same way, in [185], a custom SystemC kernel implementation is proposed. This solution implies the use of non-standard primitives in order to parallelize models. These solutions provide interesting simulation time reductions. However, the speed-up is better for models which rely on intensively execution SystemC processes with few data transfers. The atomicity and data consistency issues are analyzed in this article. Finally, the authors propose a solution where the designer annotates all potentially shared resources in the code coupled with a consistency resources monitor. This means that the designer has to take care of the parallelization inside the models.

In summary, parallelization during delta cycles is efficient when simulated modules are intensive and do not require synchronization. All modules run on the same SystemC kernel instance. Moreover, the data consistency can require an effort from the designer with specific annotations in the models. However, implementation is mainly limited to a single host platform.

4.2.3 Multiple SystemC kernels without quantum

Instead of taking advantage of parallelism within the delta cycle, another way to better benefit from the power of the host machine's power is considered. Typically, a system can be divided into multiple subsystems that can be executed in different SystemC kernels. Contrary to the previous parallelization solutions, the system modules are not necessarily hosted and executed in the same SystemC environment. This split is typically achieved by dividing models along the lines of their interconnect as described in Figure 4.5. For instance, this is done across SystemC channel(s). In this example, the split is done between the SystemC Kernel 1 and the SystemC Kernel 2. The authors in [206] propose to run different SystemC environments in a distributed manner on multiple host cores reducing the total simulation time. This solution implies a manual split of the subsystems. A bad split in the complex system can imply high latencies between nodes due to frequent synchronizations. In this solution, the synchronization is done by an external tool. This implies a modification in the SystemC scheduler. In [165], a synchronous solution based on a parallelization of channels that work on multiple hosts is proposed. This contribution focuses on MPSoC simulations on SMP host computers. The SystemC language has to be modified in order to protect SystemC process execution. The data exchange mechanism assumes a protection mechanism in the models themselves.

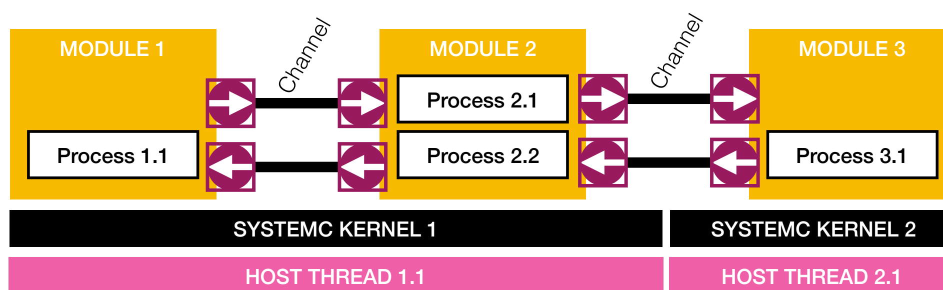


Figure 4.5 – Parallelism illustration between multiple SystemC kernels

A similar solution devoted to parallel MPSoC simulation is detailed in [153]. Processor models are dispatched on a cloud architecture. The synchronization between processor cores is done asynchronously. The data is exchanged through a message-passing cluster mechanism. However, it does not ensure the partial order of simultaneous events. The parallelization between SystemC channels is efficient when processors can be easily isolated into smaller subsystems. Finally, System-Level Design Language (SLDL) solutions other than SystemC often have a different approach to parallel execution. For example, SpecC [51], includes an automatic synchronization and protection mechanism as explained in [50].

4.2.4 Multiple SystemC kernels with quantum

In order to speed up simulation between multiple simulators, a time decoupled simulation can be applied. Indeed, decoupling time reduces data exchange rate through the SystemC kernel. Consequently, this would improve parallelization efficiency. This can be seen in Figure 4.6 between SystemC Kernel 1 and the ISS. However, with time decoupling, time coherency has to be considered with care. It happens between TLM modules as well as the ISSs.

To take advantage of the quantum mechanism in TLM-2.0, the authors of [139] propose to distribute blocks of models on different hosts. They introduce a new synchronization mechanism to bound

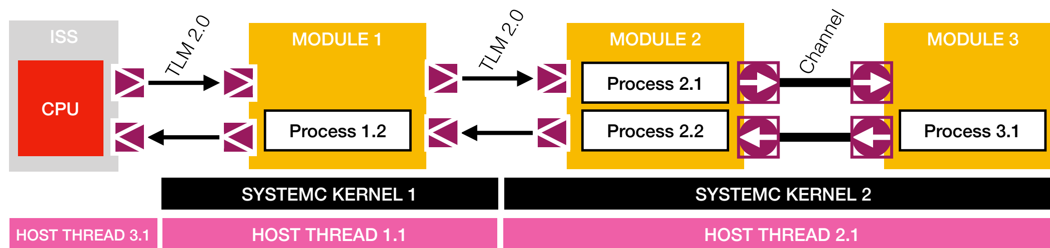


Figure 4.6 – Multiple SystemC kernels with a quantum

the temporal error that are produced by asynchronous communication. Explicit synchronizations at regular intervals are necessary to reduce the error. Each SystemC instance includes a synchronization module. Using a specific SystemC process for synchronization, they ensure that SystemC kernel execution does not exit prematurely. For the data, a part of the presented system guarantees consistency constraining the simulated system to have a write exclusive memory access policy. This discards the TLM-2.0 DMI feature. usage across modules otherwise. If a DMI feature is required, the split should be done on the same host.

A concurrent model interface is detailed in [157]. In order to avoid the SystemC kernel prematurely exiting, the quantum based synchronization mechanism sleeps (wall-clock time) in order to wait for synchronization notifications from other nodes. Even if this solution reduces CPU spinning, it is not optimal nor deterministic. Nonetheless, articles [127] and [150] clearly mention the interest of the `async_request_update()` feature introduced in SystemC 2.3.x. This enables asynchronous notification of SystemC events from other threads. It is a good candidate as starting point for the generic synchronization mechanism.

Some other approaches try to remove the need of time synchronization. For instance, the authors of [116, 141] present a distributed version of TLM called TLM-DT, part of the SoCLib [106] project. The principle is to parallelize execution with multiple instances of the SystemC kernel. Each SystemC kernel runs in its own thread which can be executed on different hosts. The SystemC kernel times are synchronized via a message exchange mechanism. This solution enables a more distributed simulation environment and removes the central time synchronization. The data exchange mechanism is done in the interconnect thanks to a central buffer. Inside, some specific protections are integrated. However, some SystemC semantics are broken. It cannot be used with existing (standard compliant) models without modifications.

Another solution with a specific TLM mechanism is presented in [95, 94]. A TLM simulation kernel for parallel simulations on multi-core machines is detailed. However, the solution exclusively focuses on the TLM API to the detriment of the SystemC kernel. Each module performs periodic synchronizations with the other modules. Their solution is based on a global simulation time synchronization algorithm. Unfortunately, the interactions with the SystemC kernel are not clearly specified.

Finally, a parallel SystemC simulation solution based on time-decoupled segments is given in [190]. The solution exploits TLM to inter-connect ISSs's with the SystemC kernel. However, each segment is required to employ it is own SystemC kernel on top of the ISSs, adding an overhead to the global simulation. Segments that have not reached their local time limit are considered as ready to simulate. Others are considered to wait for their peers to catch up. The mechanism that is necessary to pause the execution of segments is not detailed in the article. The data exchange does not trigger specific issue as the data exchange is done through channels. All the simulation components run in the same host process, hence in the same address space. Actually, the aim is

to propose a simple and efficient way to implement this kind of algorithm.

4.2.5 Asynchronicity

All these solutions provide parallelization techniques. However, they are not standards compliant. Moreover their applicability is model dependent. To overcome this, existing properties of SystemC have been studied to build a generic and standard synchronization mechanism. The properties of the SystemC scheduler imply that when no more events are available, the scheduler ends the simulation. This is the default behaviour for a stand alone DES. While SystemC is a discrete event simulator, its degree of interaction with other applications (including other simulations) has only be added in a recent revision of SystemC published in 2011. It is supported using the `async_request_update()`. This causes the scheduler to queue an update request for the current primitive channel in a thread-safe manner with respect to the host operating system. In other words, it allows any other threads to cause the scheduler to evaluate an event whose state can be updated from another thread.

This mechanism only solves half of the asynchronous problem in the SystemC simulator. While a simulator can be notified asynchronously, the reverse can also happen. A simulator can be require to wait for an asynchronous event. In this case the simulator cannot predict the future time at which it will receive the event as it can depend of an other independent thread. The asynchronicity issue happens when SystemC is waiting for one or more external event(s). If the scheduler has no more events to process, it will exit prematurely. In order to avoid this behaviour, the designer could loop around the SystemC simulation start `sc_start()` in order to restart SystemC if it has stopped. This solution has been implemented by the authors of [82]. Non standard methods available from the Accellera implementation are applied which breaks the portability of their code. The solution can also lead to bad behaviour as in `end_of_simulation()` simulation callbacks, which will be erroneously called during simulation. This could lead to resetting of models while the simulation in its entirety does not end. Another option would be to run a SystemC process that can call a host wait condition (e.g. a Portable Operating System Interface (POSIX) `pthread_cond_wait()`) when there are no more pending events, it can then detect pending activity with `sc_pending_activity()` and `sc_time_to_pending_activity()`. However, this can lead to a deadlock if a specific mechanism is used in two isolated parts of the simulation as showed in Figure 4.7.

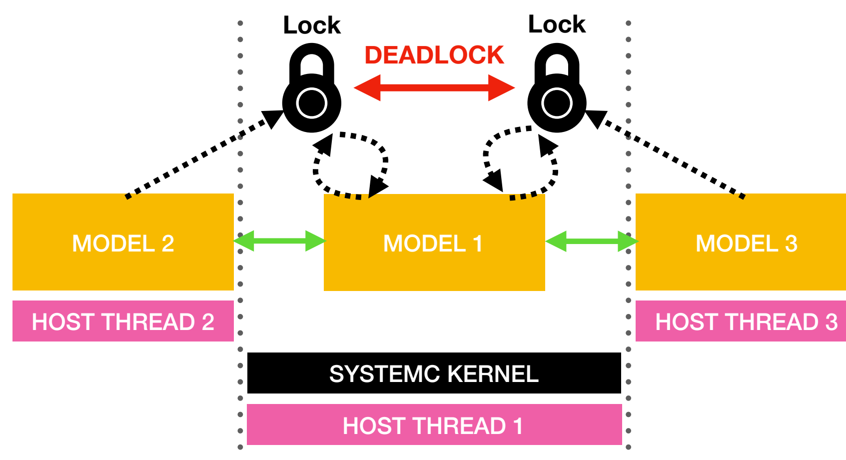


Figure 4.7 – Asynchronicity and potential deadlock

In order to add this kind of properties to a simulator, changes have to be applied with care. Allowing a simulator to interact with other host threads can potentially lead to deadlock. For instance, in Figure 4.7, two models are running in their own threads outside SystemC environment (e.g. ISS). Model 1 can catch SystemC just before the simulation finishes if it knows that it will post an asynchronous event in the future. A semaphore mechanism can be applied here. However, if Model 2 uses a similar mechanism with its own semaphore, a deadlock can happen.

Finally, the asynchronicity property aims to enable synchronization modules to be built, as shown in Figure 4.2. They are mainly composed of a SystemC process built in a standard manner. These synchronization modules that are used to synchronize various simulators with the SystemC kernel, are applied to synchronize the SystemC kernel time The synchronization process has to use a standard mechanism that guarantees no deadlock. The advantage with this kind of solution is that synchronization and data exchange are completely decoupled from the SystemC simulation.

4.2.6 Conclusion

An overview of the issues in SystemC and TLM parallelization is provided in [18]. They concludes that temporal decoupling should be exploited to reduce the amount of synchronization and hence speedup simulation. However their proposed mechanism is not designed to be generic. Moreover, it cannot extend to external simulators or ISSs.

SystemC modules can instantiate any number of `SC_THREADs`. However, the purpose of the SystemC library is to enable multiple SystemC threads to be scheduled on a single host thread. SystemC's serialization is co-operative, process must specifically elect to yield. This notion is important within the SystemC kernel and is relied upon by models. This is why parallelizing SystemC kernel at the SystemC process level is fraught with some difficulties. This level of parallelism is not examined further in the rest of this chapter.

The contribution of this chapter focuses on providing an API for asynchronous synchronization (data and time) between concurrent simulators as shown in Figure 4.2. This contribution does not attempt to parallelize SystemC models themselves, or parallelizing the SystemC kernel. The standard time synchronization issue remains whereas fine solutions have been presented to solve data exchange issues. However, parts like the DMI have not been solved if they are used in different address spaces. The next section introduces two solutions to solve asynchronous wait.

4.3 Asynchronous parallelization

4.3.1 Asynchronous event based solution

4.3.1.1 Introduction

This section presents a first contribution based on a generic mechanism to achieve synchronization between multiple threads, a number of SystemC kernels and/or others simulation environments (such as an ISSs like QEMU) as shown in Figure 4.6. It aims to add an event to the semantic with asynchronous properties, also called an asynchronous event. The mechanism enables some synchronizations between simulators with their own notion of time.

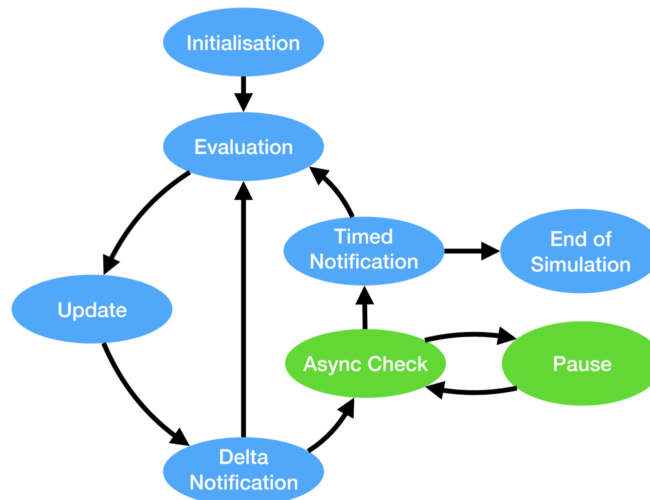


Figure 4.8 – SystemC scheduler with asynchronous mechanisms

SystemC 2.3.0 introduced a mechanism to notify a SystemC event in a thread safe manner called `async_request_update()`. It enables, for example, an external thread to collect data from another simulator. To do this, the external thread notifies its associated primitive channel which calls internally `async_request_update()`. This notification is not directly processed by the SystemC kernel itself, rather a channel update is marked as pending. During the next delta cycle in the scheduler, the SystemC kernel checks the "external" status of the pending updates (which are managed inside a queue). The SystemC kernel then propagates in a thread safe manner the pending primitive channel update(s). However, the SystemC kernel does not provide the reverse. It does not include a mechanism to wait for such an asynchronous notification event while SystemC is running out of events.

The asynchronous update feature leads to a modification of the behaviour of the SystemC scheduler. Some additional states in the scheduler are added. They are represented in green in Figure 4.8. The new state called "Async Check" allows the SystemC kernel to not advance time when an asynchronous event should appear. This state is detailed in Section 4.3.1.3.

The presented solution, based on a kernel modification, extends the SystemC features without a modification of existing semantics. Thus, it includes backward compatibility of models. This extends current semantics that are available in the SystemC standard by adding two new functions to permit synchronization with other simulators. The new mechanisms do not break the existing semantics. Indeed, no restriction is imposed on current models.

4.3.1.2 Asynchronicity and asynchronous event

An asynchronous event wait feature is the basis of what is required to implement a synchronization mechanism such as those detailed in Section 4.2. Two additional main features are required:

- Waiting for an asynchronous time (which is an event) must not advance simulation time,
- Waiting for an asynchronous event must not stop the simulation if the scheduler run out of (classical) events.

The later of these can be found in all of the attempts at parallelism listed in Section 4.2. The former is more subtle, but essentially derives from the same semantic. When waiting for a time that will

be triggered by some external event, the kernel should not advance time. Rather it should wait until other local or external events advance the time as illustrated in the Figure 4.9. The discovery of these two semantics is the key to the proposal in this chapter for the first solution. It is also fundamental to support parallelism in SystemC simulation.

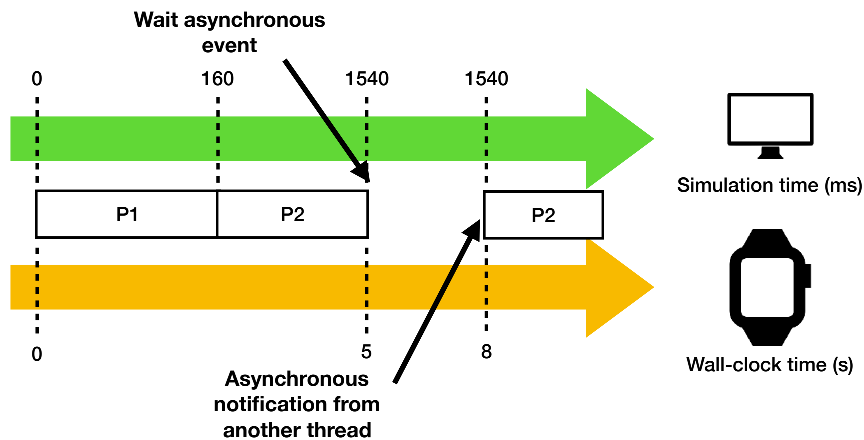


Figure 4.9 – How the asynchronous event waiting mechanism works

The initial attempt to solve this problem was to introduce a new `wait()` function called `async_wait()`. This function was compatible with regular `sc_event`. However, the usage of `async_wait()` with multiple events taints them all as asynchronous. The solution did not permit the usage of different kinds of events (synchronous and asynchronous) in the same call to `wait`. Rather, asynchronicity has been added as a property of an event. Asynchronous events are then treated specially by the kernel, as explained below. As time based events are considered in the same way as classical events within the SystemC scheduler, it should also be possible to wait for time in an asynchronous manner. Hence, a new class called `sc_async_time` based on `sc_time` that contains the asynchronicity property has been introduced. The semantics for waiting on events or times are as follows:

- `wait(sc_event)`: *Wait for an event. If the event has not occurred, and there are no runnable processes then the simulation may exit, unless there is a pending wait with an asynchronous event.*
- `wait(sc_async_event e)`: *Wait indefinitely resuming when the event has happened. In this case no more SystemC processes are runnable, the SystemC kernel pauses execution until a process becomes runnable due to an asynchronous event.*
- `wait(sc_time)`: *Wait for a simulation time. If there are no other runnable processes between the current simulation time and the pending time, SystemC kernel advances time and marks this process as runnable.*
- `wait(sc_async_time t)`: *Wait for the simulation time to advance by* `t`*, without actively advancing the simulation time. The SystemC kernel does not mark these (or any subsequent) processes as runnable unless there are pending events with simulation times greater than or equal to the specified time. Note that it may cause the simulation to finish if there are no other runnable processes.*

The SystemC kernel should not advance the simulation time for simulation time for events that will be triggered by `sc_async_time`. The kernel should wait until other events in the system cause time to advance. This means that the kernel does not move time forward for these sorts of events as it would do normally. Although a new family of events has been added. It does not change the existing semantic rules. `wait()` for `sc_event` and `wait()` for `sc_time` are the current semantics for `wait()` in SystemC syntax. Moreover, just as with synchronous event, all current functions like

`wait()` support the additional semantic rules for asynchronous events.

`wait(sc_async_time t, sc_event e)` and `wait(sc_async_time t, sc_async_event e)` semantics can be naturally defined from the previous definitions and current standard of the `wait()` semantic. Combinations with `sc_event_and_list` and `sc_event_or_list` should follow the same logic. Lists (both *AND* and *OR* logic functions) may contain a combination of asynchronous events and normal events. The `async()` function introduces the concept of "waiting" for both a time and an event. This is the combination of the semantics of the "wait" functions for both a time and an event. While this function is not strictly necessary, the only other alternative is to 'spin waiting' which is incredibly inefficient and waste-full. This new semantic is defined as `async()`. It should be read as "asynchronous synchronization". `async()` takes both an asynchronous time and an asynchronous event, combining the semantics. It operates like `wait()`, suspending the calling SystemC process. It enables the process to resume when both the event occurs, and the time is as specified. The `async()` function arraigns that time does not move forward from the time specified. So, it guarantees both the event and time happen together. This result is different from the classical `wait()` function which does not ensure both the event and time happen together.

- `async(sc_async_time t, sc_async_event)`: *Wait indefinitely, resuming when the event happens at the simulation time which has advanced by* `t`. *This function is only defined for asynchronous events, and asynchronous time. While a process is not runnable, due to this function, and there are no other runnable processes between the current simulation time and the requested time, SystemC kernel does not advance time or makes processes runnable. This happens when the kernel has a time greater than that specified. In the case that there are no events pending between the current simulation time and the time specified, the SystemC kernel pauses the execution until a process becomes runnable due to an asynchronous event.*

To implement the proposed approach, a new class for asynchronous events, a new class for asynchronous time and a new type of 'wait' like function called `async()` have been introduced. This last function guarantees that both the time and the event happen at the same simulation time. The function is summarized in Figure 4.10. Other classes have been updated but the implementation is specific to the SystemC POC.

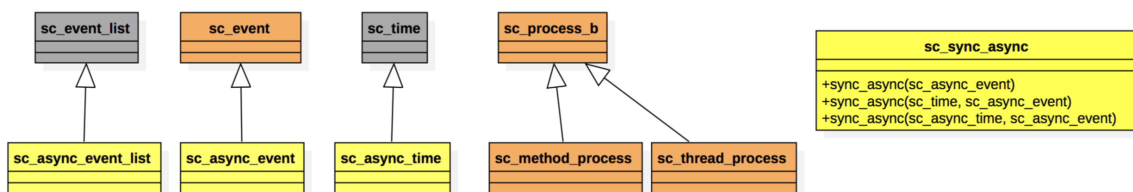


Figure 4.10 – Updated and new classes in the SystemC kernel. Grey = existing. Yellow = added. Orange = modified.

4.3.1.3 Modification of the SystemC kernel

In order to exploit the proposed features, a modification of the SystemC kernel scheduler is required as shown in Figure 4.8, and has been implemented within the Accellera SystemC POC simulator. The modification adds an asynchronous check state into the scheduler and evaluates pending asynchronous events or times. This check happens at the end of the delta cycle, before the potential increase of simulation time. In some cases, the asynchronous mechanisms can pause the SystemC kernel at this point in the delta cycle, waiting for an external event. The

asynchronous check state alows the SystemC kernel to pause if there are no more runnable processes, so long as there is at least one pending `wait()` for an asynchronous event.

The SystemC kernel is paused by using a POSIX condition. It is resumed by using a POSIX signal in order to avoid the CPU spinning. The Accellera POC simulator already includes a class for host mutexes so the proposed solution adds an implementation for host conditions and signals in a similar way. In the SystemC POC, the simulation context class (`sc_simcontext`) has been updated to pause the simulation in a more convenient way. Thus `block()` and `unblock()` methods are provided.

When a new event is notified with the asynchronous notification mechanism, the condition is unblocked. Then, all processes are evaluated again during a delta cycle. This functionality has been added to the method `async_request_update()`, part of the class `sc_prim_channel_registry`. The delta sweep mechanism has also been altered. A pending process is not marked runnable if it is waiting for asynchronous time. Note that an asynchronous event (such as that delivered to an `async()`) is considered as a normal timed event. It causes the corresponding process to be marked runnable through the semantics of `async()`.

The existing semantics of `wait()` does not have the strict *AND* between a time and an event. In contrast, when triggering from an asynchronous event or an asynchronous time in an `async()`, both arguments are required to set a process as runnable. Therefore, the trigger method in the process has been edited to handle this specific case. The new `wait()` and `async()` semantics have been added in the `sc_module` class. So, they are also available in processes. Then, notifiable events at a specific time are checked. If there are timed asynchronous events (`sc_async_time`) but there is no normal event posted at current or later simulation time and no asynchronous events, the SystemC kernel ends. In other words, an asynchronous timed event is not marked as runnable unless there are subsequent normal timed events at the time of the asynchronous timed event. The rest of the semantics are then followed. As normal, the semantics potentially cause the SystemC kernel to end if it has no runnable processes. However, if there is at least one pending asynchronous event, the simulation pauses in accordance with the semantics of waiting for an asynchronous event. Finally, the `async()` semantic is the result of both `sc_async_time` and `sc_async_event` behaviors in the scheduler in addition to a different runnable condition in the process.

4.3.1.4 Conclusion

In summary, a modification of the SystemC scheduler is proposed. Asynchronous synchronization features have been detailed. This enables parallelization of models. New semantics enables waiting for an “asynchronous” time. This avoids simulation time advancing. Another semantic to wait for an asynchronous event without the simulation stopping has been introduced. The kernel modifications are in line with the existing SystemC standard preserving all existing semantics. This contribution is available as open-source [57]. It has been proposed to the SystemC LWG for a standardization. However, due to issues found after further investigations, the solution has not been retained by the LWG. While this solution aimed to simplify the build of synchronization solutions using the `sync_async` mechanism, it is too strict a condition in some cases which could lead to deadlock.

4.3.2 Asynchronous channel solution

4.3.2.1 Introduction

A second approach aims to answer to the same asynchronous synchronization problem in order to wait asynchronously for events from external threads without deadlocks. The first proposition was build on the `wait` mechanism to be consistent with the SystemC syntax. However, this approach, that provides backward compatibility, can confuse the designers. Consequently, a second approach is proposed.

To enable parallel synchronization between multiple 'external' models, few features are actually required. It includes time synchronization. Due to the asynchronous properties of external events, the simulator has to remain running, even if there are currently no more eligible processes to run. If the SystemC simulator is too far ahead, it should stop and wait until others asynchronous simulators catch up. If the SystemC simulator is too far behind, SystemC should try to catch up with the other TLM models that run in external threads. This could be done injecting events to move time forward if no more events are present.

As explained before, the wait of asynchronous event can introduce some deadlock issues if the designer does not take care. Indeed, the mechanism can stop falling off the end of time. If a few models, running in external threads, use different mechanisms to lock the SystemC simulator a deadlock can occur as explained previously. In the first proposed approach, a global lock has been proposed, within the SystemC kernel. This solution looks like the best candidate to avoid deadlocks, it guarantees that two or more locks cannot be in competition for the same objective. In the second approach presented below, the mechanism to manage, access and trigger this lock has been modified.

4.3.2.2 Callback approach

A second approach has been investigated based on the SystemC phase callback mechanism introduced in 2.3.0. This mechanism is still not standardized by IEEE. This feature is currently experimental. It has to be explicitly enabled during the building of the library. The phase callback mechanism enables the execution of functions on specific internal simulator events. This includes the end of initialization phase, the end of the update phase and also before a time step. These can be used to integrate custom introspection techniques in a non-invasive manner.

Callbacks enable hooks in the SystemC kernel that can be used to avoid it exiting, without having the synchronization mechanism in the model itself. While the before time step callback can be used to avoid SystemC exiting, it does not work in all cases. Indeed, it doesn't centralize the lock and doesn't avoid the 'busy waiting' mechanism. Moreover, there is no way to catch the simulation before it exits. A first solution consists of adding a new callback before the end of the delta phase. This means callbacks would have been fired if there are no further events in the delta. However, this solution has triggered performance issues. The end of delta phase can happen at multiple times during a simulation as showed in Figure 4.8. Moreover, the callback can be called while nothing is happen in it. Indeed, if there are no more pending activity, it would be required to post new events or use a semaphore. In any case, this mechanism also implies that the semaphore, and hence the lock, would be defined by the user, which, as has been said before, can lead to deadlocks. Finally, this experimental solution increased the justification to introduce a global lock in the SystemC kernel. Indeed, it is currently a major issue in order to ensure the interoperability.

4.3.2.3 Asynchronicity and channel

While in the first approach, the asynchronicity management is in the `wait` semantic associated with specific events, the second approach implements the asynchronicity directly in primary channels. In the Accellera POC implementation, primary channels are all registered in a central place, named "registry". This registry is evaluated during the delta cycle to perform the updates. It happens in the Update step as illustrated in Figure 4.11.

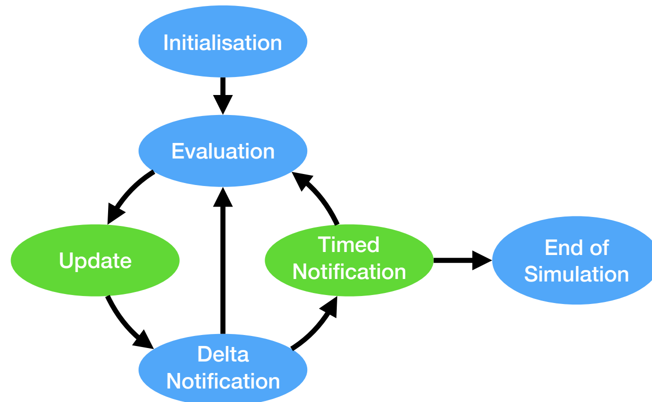


Figure 4.11 – Principle of the SystemC scheduler channel update

During the Update state, if an asynchronous updates have been requested, the channel is evaluated in a thread safe manner. This mechanism has been extended to support the asynchronous wait to avoid SystemC simulation termination. In other words, the reciprocal of `async_request_update` function has been introduced in a standard way manner.

Two new methods have been added to the primitive channel class. The objective is to mark a channel as waiting for an event notification from another thread. This is a mechanism to not let SystemC exit if a channel is waiting an external notification. The semantic of the functions is detailed in the next section. The global solution is illustrated in Figure 4.12. In the example, four TLM models run in their own threads while the SystemC simulation kernel has its own thread. SystemC models use asynchronous events that are built on top of primary channels. The code is given in Listing 4.2. The key is that each model attaches the channel to an external source of events through semantics defined in the next section. In this solution, the lock is stored in the SystemC kernel and so prevents deadlocks.

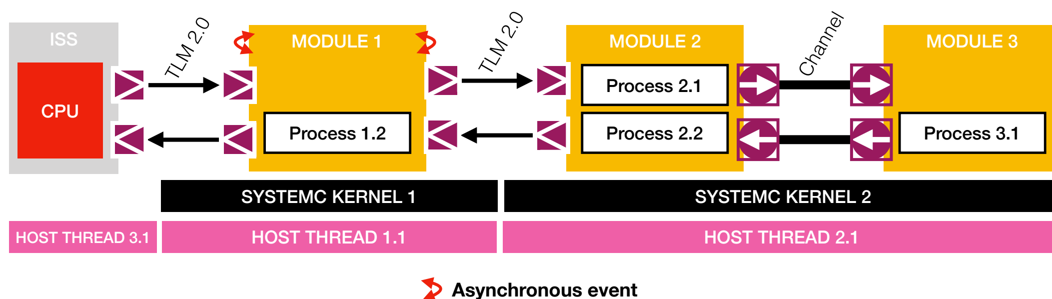


Figure 4.12 – SystemC kernel with other TLM models in different threads

To deal with external notifications which are pending, a mechanism has been added to the Timed notification state in Figure 4.11. This kernel modification in the scheduler checks if there are channels marked with pending external notifications before ending the simulation. In that

case, SystemC blocks on the global lock until an external event notifies with the already existing `async_request_update`. This function has been updated in order to release the lock to allow SystemC to proceed with the channel update. As long as there is at least one channel marked as waiting external update, SystemC does not stop the simulation.

4.3.2.4 Formal function definitions

This solution is based on two new semantics:

- `bool async_attach_suspending()`: *A primitive channel can elect to attach to an external source of events (and therefore request the presence of the semaphore). If the SystemC kernel runs out of events, rather than exiting as it would do normally, the semaphore will lock it and wait for external updates.*
- `bool async_detach_suspending()`: *A primitive channel can elect to detach from an external source of events (and therefore remove the request for the presence of the semaphore). If no primitive channels are then attached to external events, the semaphore plays no role in simulation and the simulation kernel will exit as usual.*

The global lock, using a semaphore, is completely abstracted for the designer as it is inside the SystemC kernel. Both functions are used to indicate that a channel is asynchronous and can call `async_request_update`. The kernel should arrange to suspend rather than exit while this channel is attached, thanks to the global lock. It is detailed in the next section.

4.3.2.5 Implementation in the SystemC kernel

The previous section has described the key parts of the second approach. It introduced two new semantics and a global semaphore. As the two semantics are related to the primitive channel class, they have been introduced in it as new methods. Both methods are internally tightly coupled to the channel registry, which is specific to the Accellera SystemC POC, and has been updated. A vector that contains all the channels attached to an external source of events has been added. Attach a channel implies that the channel is added to the vector. Detaching a channel means to removing the channel from the vector. This vector enables the scheduler to easily check that there is at least one attached channel to an external source of events. It can then update or not its behaviour before exit.

A host semaphore class has been implemented similarly as host condition and signals in the previous solution. This semaphore is not part of the simulation context but is part of the channel registry. This registry is also unique and attached to a simulation context. The semaphore is checked (and potentially waited for) if no further events are available before the end of the simulation. This happens during the Timed notification state. The code is available in Listing 4.1. The implementation has the advantage that there is no additional cost for the 'check', as it only happens at the end and not during the simulation. The global kernel semaphore is released when the `async_request_update()` function is called. This is the only added cost during the simulation execution. It is incurred whether there is are suspendable channels attached or not.

Listing 4.1 – SystemC kernel suspending

```
1 if ( !next_time(t) || (t > until_t) ) {
2     if ( (t > until_t) || m_prim_channel_registry->async_suspend() ) {
```

```
3            // requested simulation time completed or no external updates
4            goto exit_time;
5        }
6        // received external updates, continue simulation
7        break;
8    }
```

4.3.2.6 Conclusion

This second approach is based on SystemC primitive channels. It enables to notify that a channel is going to receive external event. This mechanism avoid SystemC to die prematurely to wait external events. In contrast to the first approach, the simulation cost is only at the end of the simulation and during the attach/detach of channels, which can be resumed to a push to a vector. The second approach has been implemented by Mark Burton in agreement with Philipp A. Hartmann in order to build directly an upstreamable and standardizable solution according to the SystemC LWG constraints. This second approach is also simpler than the first one. Simulation execution time results will be given in the experimental results section.

However, there are some limitations. This second approach does not provide all the features contained in the first approach such as the opportunity to lock the SystemC time during a simulation. However, it is possible to reproduce this behaviour using a loop. Indeed, a 'spinning' loop is faster than a lock/unlock mechanism.

4.4 Synchronization and quantum impact on parallelization

4.4.1 Ordering and timing of the simulation

One of the most important features of the SystemC library is how time is modelled. SystemC contains its own notion of time that is normally called the simulation time. The SystemC kernel manages its own local time and provides a global time stamp. The SystemC kernel advances time to the next outstanding SystemC event. Hence, SystemC time is non-linear and bares no relationship to real time. Furthermore, activity can occur during zero simulation time.

TLM-2.0 introduced temporal decoupling and the notion of quantum. Time decoupling is the term given to models that run within their own time reference “decoupled” from the SystemC kernel simulation time. Typically they run ahead of the kernel simulation time. SystemC is fundamentally constructed to provide a “co-operative serialization” of parallel threads. Each thread in SystemC is expected to periodically yield to other threads in the system. This enables all threads to advance. In the case of TLM-2.0, it is expected that initiators are SystemC threads. A quantum simply indicates how often they should guarantee to relinquish control to other threads.

Initiators in a TLM context are often complex models such as CPUs. To be efficient simulations are often require to consider relatively large blocks of functionality, which may take considerable time to execute. It is advantageous from a simulation point of view to execute these blocks atomically. The quantum limits the size of these blocks. Hence in general, the larger the quantum is the larger the block of functionality that may be considered. The simulation performance may be more efficient. However there is a limit to the size of block that is optimal. This effect was examined in

[58].

At the end of a quantum a model is expected to be re-synchronized with the simulation time reference. For instance the SystemC wait function can be called and so other events can occur. A convenience function is provided in the TLM Quantum Keeper which appropriately calls wait to achieve synchronization. The TLM Quantum Keeper is present in the TLM-2.0 kit to help to manage local quantum time. TLM-2.0 recommends the usage of the TLM global quantum, but it is possible to use different local quantums in different sections of systems.

Additionally, TLM-2.0 introduced the notion of annotated time on transactions. It corresponds to a current temporal decoupled time within the quantum. Temporal decoupling and the use of a quantum is only supported at the LT level of abstraction defined in TLM-2.0. Overall, there are different notions of time: wall clock time from the person running the simulation, the SystemC simulation time, the quantum time and the annotated time.

Generally, local quantum time of initiators increases through local compute operations or through TLM transactions using the annotated time. Typically, an initiator can send a transaction with a delay α. The target reads this delay, prepares an answer, possibly increases the delay by including the time of the answer and then sends the results and the time back to the initiator. The initiator sets its local quantum time from the value received from the target. Hence, the expectation in the TLM-2.0 library for typical target models is that they may not need to interact directly with a quantum keeper.

In the article [58], the efficiency of using quantums is studied. By correctly adjusting the quantum length, simulation speed is optimized. However it is apparent that there is no simple formula to calculate the optimal quantum value. This is a significant disadvantage of the approach. Indeed when the effect on simulation speed against quantum length is plotted, there can be several optimal values. Meanwhile the commonly accepted practice within industry seems to be to set the quantum to the minimum interrupt period expected in the system. Indeed our earlier work shows a minimum simulation time at exactly the point where the quantum is set to the period of the clock used to drive the Linux kernel. It is the case because it is the smallest clock in the system. For the experiment, 10ms as the kernel clock is set to a frequency of 100Hz. This is a reasonable approach. As expected, as what ever is driving those interrupts needs to be triggered from the SystemC kernel. However this approach has its own limitations. System clocks are typically responsible of regular and frequent interrupts. They are not only programmable but different operating systems have more or less tolerance of different clock 'skew'. They require different frequencies. Hence, the quantum needs to be adjusted by the end user. As systems become more complex, finding a common denominator for all the clocks in a system becomes more complex, especially if some of those clocks are not known. Consequently, the result could be a very short quantum.

A different approach has been proposed in which ordering, rather than time is considered. In this approach, rather than periodically yielding each quantum, specific points are found in the design where yielding is imperative. This is typically where information is passed between initiators. In this case, models are "synchronized" with each other as required and not necessarily with the SystemC simulation time. Effectively, an optimal serialization of the different simulation threads is achieved [18]. This has a number of advantages:

- It assists for solving problems in a design architecture caused by the expectation of synchronicity that is not enforced in the hardware architecture,
- It constraints designers to consider how synchronization happens in their architecture.

This technique does not require any adjustment of the quantum by the end user. However, it has a number of disadvantages. Models of this sort may not work with sub-systems that expect quantums to be used. In an attempt to mitigate these issues, some changes to the way in which the quantum keeper is implemented and used are presented. Models that are capable to generate interrupts from external sources (e.g. clocks) are a critical driver for quantum lengths. Research focused on them, though the solution should work for a much wider range of events in the system.

4.4.2 Endless quantum keeper

It has been shown how the critical driver for quantum lengths seems to be timer interrupt generators and other sources of interrupts. These are typically driven from events based on the SystemC simulation time. However, really their time relates to the local quantum time of the device that they will be interrupting. It would be more relevant if they were based on the quantum time.

Doing so would also remove the need to base interrupts on events that were essentially local to the local quantum domain (typically a SystemC thread). It would allow system designers to concentrate on the inter-thread issues, either adopting an ordering, or a quantum based approach. Enforcing a synchronization would then allow an ordering approach across these interfaces. In addition, this may be a way forward in terms of standardization. In other words, one possibility is that the standard stipulates that any communication between a quantum domain is accompanied by a synchronization between the domain and the SystemC kernel. There is currently no technical means of ensuring this, which does not also effect internal-quantum domain communication.

4.4.2.1 Notification system

Currently, a timer can use SystemC events to get a periodic callback based on simulation time. If the timer period is smaller than a quantum, the timer is called less frequently than it aught to as seen from the perspective of the initiator that will receive the timer interrupts". There is a detrimental effect on simulation performance. The proposal is to modify the TLM quantum keeper class to provide an event queue based on the local quantum time. The improved quantum keeper class would add methods to register a callback at a specific time. As quantum time progresses under the control of the initiator, it executes any pending callbacks at specified time. Typically, a timer can use this notification system to run a clock decoupled from SystemC time, without synchronization between the local quantum time and the simulation time. As the timer is now based on the initiator local quantum time, this solution also improves the timing accuracy of the model. Note that, while an example of improved Quantum Keeper is presented, the key is the API and the interoperability.

4.4.2.2 Quantum keeper improvement

Two major changes are required to implement a notification mechanism based on the local quantum time. First the quantum keeper itself needs some modifications; the API of the quantum keeper needs extending. Second, the quantum keeper must be findable by all models within the quantum domain. It islikely encapsulated by an element of the SystemC model hierarchy. In order to solve this issue, the quantum keeper should become a first class CCI object (`cci_param`) supportable through the CCI standard. In that case, it could be found through the CCI parameter

searching system if you know their name. The code of the proposed Quantum Keeper is given in the Appendix A.2.3. An experimental result is given in the Chapter 5.

4.4.2.3 Conclusion

This section has gone on to look at how time is modelled and synchronization between time domains is achieved. It founds that there are currently ambiguities in the standard and different approaches. A new Quantum Keeper that provides an event queue assists in simplifying the differences between these approaches has been proposed.

The proposed version of the quantum keeper is currently a POC. If a target is able to access to a quantum keeper, it is possible to add different kinds of notification methods. A future improvement is to add a registration method with a period argument. This is a matter for further works within the standardization bodies, to find the right balance between mirroring the existing SystemC event notification system and to provide extra utility to users.

4.4.3 Quantum based synchronization solutions

4.4.3.1 Introduction

The first part of the chapter introduces a mechanism to enable missing asynchronicity feature. The low layer of the whole synchronization solution is presented. The attachment to an external source of events inside channels enables models to run in the SystemC thread to wait TLM models running in other threads. However, the time coherency through synchronization as introduced at the beginning of this chapter has not been solved. It is what this section aims to solve.

In order to do time synchronization, two major synchronization algorithms have been analyzed. Both are based on the quantum. They are to be used with TLM models. The first one is based on a static quantum with a strict quantum check. The second solution is based on a windowed quantum which is more lax than the first.

4.4.3.2 Static quantum

The first sync algorithm is the more natural one. At the start of the simulation, the models that are executed in parallel and run until they end their quantum. The quantum is the same for all. When the first model reaches the end of a quantum, it notifies the others and waits from them to finish their quantum too. When all models have done with their quantums, they continue with a new quantum. The end of quantum is a synchronization barrier as showed in Figure 4.13.

This solution ensures that there is never a difference between two models greater than a quantum. It is summarized in the Equation 4.1.

$$|t_{modelAhead} - t_{modelBehind}| \leqslant quantum \tag{4.1}$$

However, this synchronization solution is not optimal. If a model is faster than others, it can wait for others to catch up, which is not optimal as some threads could be idle.

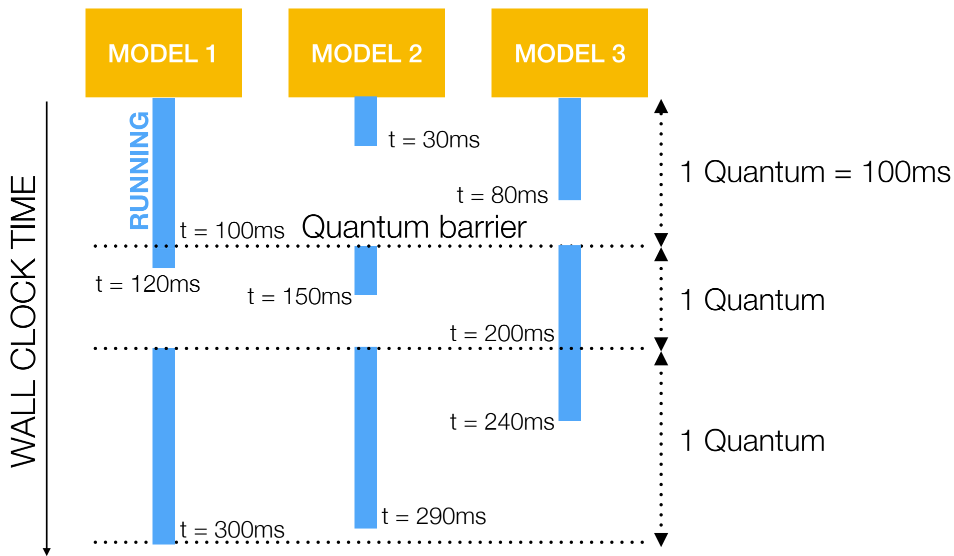


Figure 4.13 – Execution example with a static quantum

4.4.3.3 Windowed quantum

The second sync algorithm aims to fix an issue of the first one. The windowed quantum solution tries to avoid quantum barriers. At the start of the simulation, the models that are executed in parallel start and run. When they update their local time or when there are interactions, they notify the window synchronizer as showed in Figure 4.14. Model 1 notifies the synchronizer at $t = 50ms$. Then, model 2 notifies the synchronizer at $t = 100ms$. It stops because it is not allowed to move on. Finally, model 3 notify the synchronizer at $t = 75ms$. The synchronizer notifies all models that all models are now running with a minimum local time equals to $t = 75ms$. It allows model 2 to continue even if all models do not reach the end of quantum. The window of the quantum effectively moves.

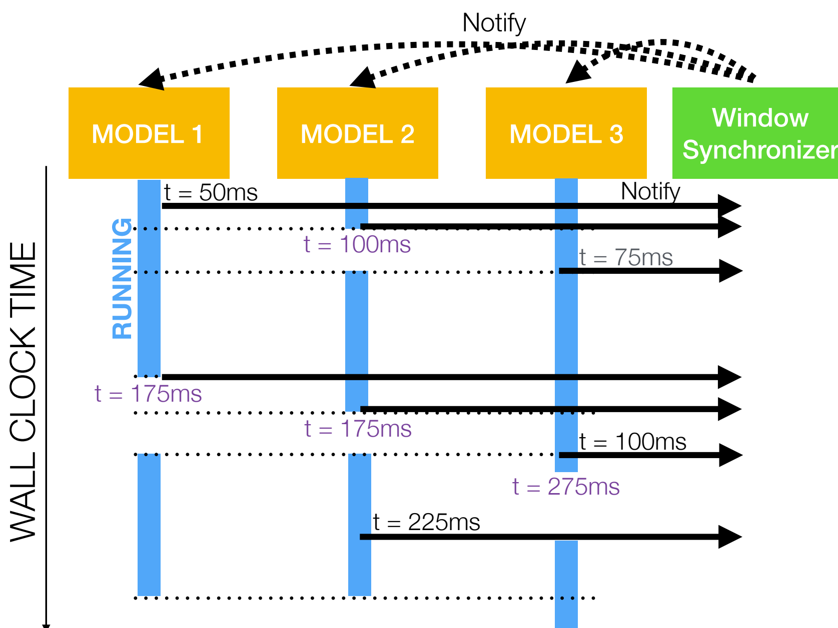


Figure 4.14 – Execution example with a windowed quantum

This solution ensures the same equation than the first solution. However, there are less quantum barriers. Models can move on before all other models reach the end of quantum.

4.4.3.4 Conclusion

This section showed two quantum based synchronization solutions. They can be used to synchronize TLM models with the running SystemC kernel. The first solution aimed to solve the time synchronization with a natural quantum barrier as available in SystemC itself. This solution can be sub optimal. If a TLM thread is much more slower than the others, it can delay all others. On the other hand, the second solution tries to solve this issue moving the start barrier of the quantum as much as possible.

4.5 Experimental results

4.5.1 Introduction

To demonstrate the validity of the presented mechanisms, solutions have been measured. These tests constitute the most basic use cases. Two use cases have been evaluated. One of them is quantum based and the other one is an un-timed synchronization. All the experiments have been done on a host platform built on a Intel i7-6700HQ CPU composed of 4 cores and 8 threads. An external timer, wall-clock time, was used to measure execution time, averaging over ten runs.

An asynchronous thread safe event has been constructed based on previously presented solutions. It is described in the Listing 4.2. It constitutes the core of the synchronization. It can be used as a normal SystemC event. Basically, when the event and its attached primitive channel are created, they register the channel to not end the simulation when the kernel runs out of events. It also implements thread-safety notification using the `async_request_update()` update feature. The `update` method is evaluated only during the update phase of the SystemC scheduler. This updated always happens in the SystemC thread, whichever thread which does the notification.

Listing 4.2 – Synchronization module

```
1  class thread_safe_async_event : public sc_core::sc_prim_channel {
2  public:
3      thread_safe_async_event(const char* name = sc_core::
         sc_gen_unique_name("async_event"))
4              : sc_core::sc_prim_channel(name)
5              , m_event((std::string(this->basename())+"_event").c_str())
                 {
6          async_attach_suspending();
7      }
8
9      void notify(sc_core::sc_time delay = sc_core::SC_ZERO_TIME) {
10         m_delay = delay;
11         async_request_update();
12     }
13
14     operator const sc_event&() const { return m_event; }
15
```

```
16  protected:
17      void update(void) {
18          m_event.notify(m_delay);
19      }
20
21  private:
22      sc_core::sc_time m_delay;
23      sc_core::sc_event m_event;
24  };
```

4.5.2 Two SystemC kernels without time synchronization

The first experiment corresponds to a basic producer and consumer use case. Two SystemC modules executing in two distinct threads are simulated as illustrated in Figure 4.15. The modules are the same in each kernel. The add module is in charge of increment the received value and send it back. It also consumes simulation time. The add export module is in charge of the execution of the value bridge between both kernels. POSIX sockets have been used for the communication between both add export modules. Basically, this example aims to compare the synchronization speed of different asynchronous mechanisms that are now available.

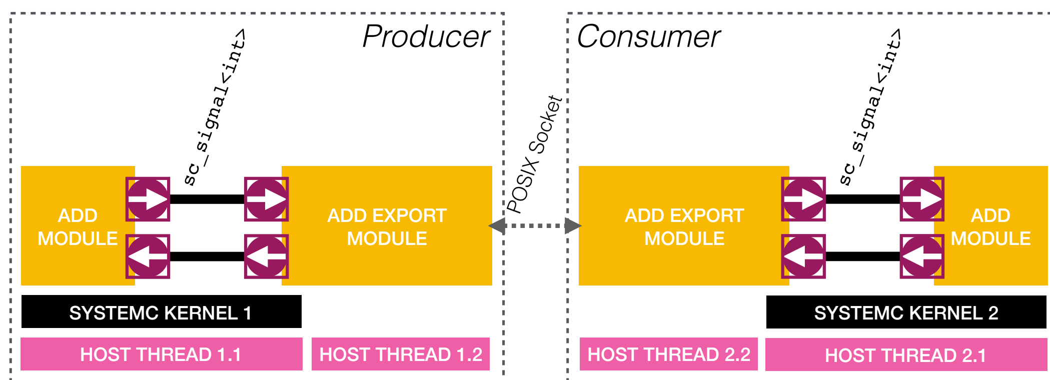


Figure 4.15 – SystemC kernels that run as producer and consumer

In this experiment, each SystemC kernel runs in its own host process. The time synchronization is not considered in this experiment, so asynchronous event usage is enough. The source code of both modules are provided in Appendix A.3.1. The simulation duration have been limited to 5000 seconds. Results are given in the Table 4.1. The first approach is the slowest one, which is explained by the addition of new steps in the scheduler that are executed in each delta. The version using SystemC callbacks is about three times faster than the first approach. However, as quoted before, it does not guarantee a deadlock-free simulation. The second approach not only includes a simpler semantic but is also the fastest approach. The addition to the scheduler is only executed when the SystemC simulation is going to end, which happens less frequently than the first approach.

4.5.3 Two SystemC kernels with a quantum based synchronization

The second experiment is close to the first. However, the time synchronization between SystemC kernels is considered. Consequently, a synchronization mechanism of the time has been added in

Table 4.1 – Comparison of asynchronous mechanisms (without synchronization)

Asynchronous mechanism	Runtime (ms)
SystemC callbacks	235
First approach	711
Second approach	198

the Add Export module. It is based on quantum. Both kernels exchange periodic asynchronous events to notify the end of quantum. The source code of this update is given in the Listing 4.3. It has been implemented from the second approach. When a SystemC kernel instance reaches the quantum time, it checks that the other simulator has also reached the quantum boundary. If it is not the case, it waits for it to do so before continuing. Basically, the synchronization code is simple but efficient. It is interoperable and deadlock-free. However, the SystemC kernel "spins" here. which is one drawback compare to the first approach.

Listing 4.3 – Synchronization SystemC thread

```
1  void sync() {
2      while(1) {
3          ipc->sendEOQ(); // Send end of quantum
4          sc_core::sc_module::wait(tlm::tlm_global_quantum::instance().
             get());
5          while(!ipc->quantumNotification) {
6              sc_core::sc_module::wait(sc_core::SC_ZERO_TIME); // Do not
                 move
7          }
8          ipc->quantumNotificationOK(); // Update notification
9      }
10 }
```

The simulation duration have been limited to 5000 seconds. Results are given in the Table 4.2. The SystemC callbacks is the slowest one, which is explained by the executions of the callback mechanism between two synchronizations while no action is required. The version using he first approach is about two times faster than the SystemC callbacks approach. The second approach is also the fastest approach. It is noted that the synchronization mechanism increased simulation time compare to the version without time synchronization.

Table 4.2 – Comparison of asynchronous mechanisms (with synchronization)

Asynchronous mechanism	Runtime (ms)
SystemC callbacks	40382
First approach	21374
Second approach	17840

4.5.4 Summary

In this section, different approaches have been implemented to enable the synchronization of different SystemC kernels with or without the time synchronization. While the first approach offers a richer semantic that is available in SystemC processes, its performances are not the best. On the other hand, the second approach offer a simpler semantic. It enables to achieve better performances. However, the time synchronization with the second approach based on quantum implies spinning. A more concrete evaluation of the second approach is given in Chapter 5. The mechanism is applied to a complete virtual platform that contains multiple cores running in parallel.

4.6 Conclusion

Different approaches to enable the execution of different SystemC processes in different threads are detailed in this chapter. To do this, asynchronicity has been considered. SystemC supports the asynchronous notification of events but has no mechanism to wait (indefinitely) for asynchronous events. After a first approach presented in [59] and [66], deadlock issues were been found, new approach was considered. This approach has been upstreamed to the SystemC LWG and merged in the mainline. The original contribution has been released publicly in SystemC 2.3.2, available publicly. With only few lines of code, it is possible to build an asynchronous event. This event can then be applied to build a synchronization mechanism for data and the time. The last three chapters contain contributions for the configuration of SystemC models, the communication between TLM models and to speed up the simulation. The next chapter details a concrete application of all these contributions.

Chapter 5

Application

***

5.1 Introduction

Previous chapters have presented contributions to address new challenges for SoC modeling. They have been validated with different applications or standardization process. Nonetheless, to give a better explanation of these contributions, they have been applied to a concrete use case. The use case that has been selected is a system that controls the lights of a city as illustrated in Figure 5.1. Each node has two features: first, to control the light and second to report the state of the sensors. Each light has to be controlled independently. The monitoring of these nodes is decentralized. A single gateway acquires and processes the data of all nodes.

In order to model the system that is required for nodes and for the gateway, a virtual platform has been designed. The objective of this virtual platform is to illustrate what has been proposed in this manuscript. It consists of a gateway and several sensors. The sizing and the response time of the nodes and gateway functions are evaluated.

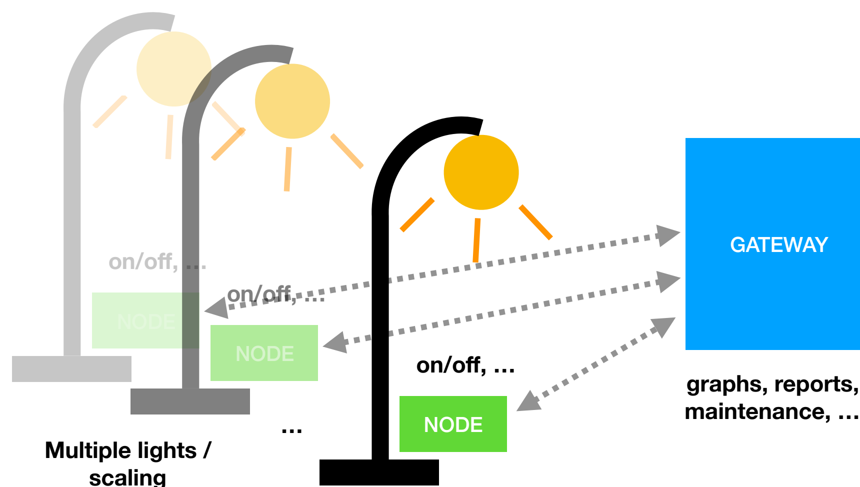


Figure 5.1 – IOT platform use case

The objective of this platform is to control the light in the city and to maximize a quality of service. To do this and for questions of flexibility, each node is developed around a system processor and hardware blocks. It is necessary to qualify the architecture of the processor as well as to study the impact of different internal parameters. Then, the protocol used for the communication between the different parts can change during the design time. Hence at an early step, the final architecture can be unknown. Multiple radio protocols are available. A fast simulation is required with a high number of nodes to benchmark within a reasonable time.

According to the system specification, each system element contains at least one processor core. Thus, there is a need for simulated CPU models in the virtual platform. There have been many attempts to build libraries of CPU models. Some of them are not build natively with SystemC as its core but instead interacts with it only for input/output transactions through wrappers. These solutions can be based on QEMU [143], an application that emulates different CPU architectures. However, a proper integration with SystemC is missing.

Section 5.2 introduces the requirements for the simulation of a complete IOT platform. Next, section 5.3 presents QEMU in a Box (QBox), a solution to embed QEMU in a SystemC model as a CPU. Section 5.4 details the modelling of the entire platform with QBox and different contributions presented in previous chapters. Then, performances of the platform are analyzed in Section 5.5. Finally, a conclusion is given in Section 5.6.

5.2 Requirements

In the context of a light control mechanism in the city, the system should be designed to fit exactly the requirements. It should not be oversized to limit costs. It should not be undersized to avoid performance issues. One option is to purchase many different development boards and then to perform an evaluation of each of them. This implies that for each board the hardware and software must been designed. This can be a time consuming and an expensive step. In the proposed use case, the modeling of the platform should provide the same features as that approach. It enables to dimension the system and to test hardware configurations or hardware design choices. The virtual platform aims to confirm the choices in the architectures. It also allows the software teams to develop the codes that are not yet chosen. Consequently, the level of abstraction for the modeling of the platform is LT.

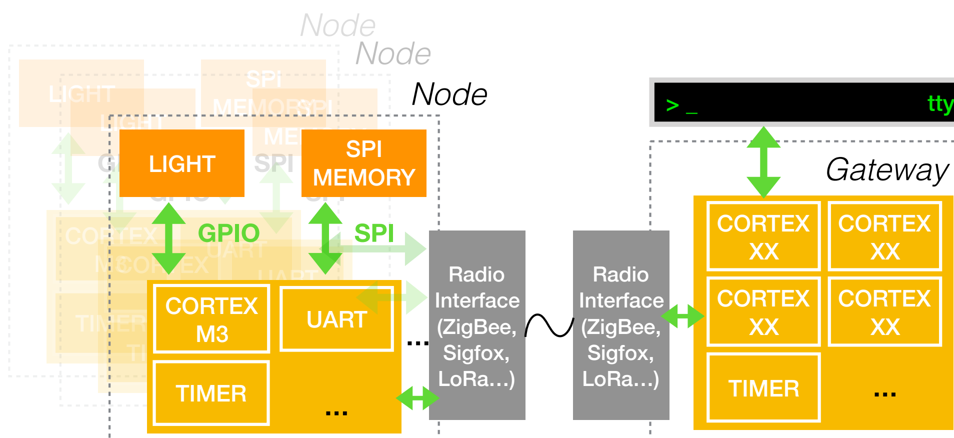


Figure 5.2 – IOT platform architecture

A simplified version of the architecture is given in Figure 5.2. Each node, placed near a light, is in charge of the light control. It should be able to report information like the on/off status, the light sensor, etc. Thus, each node is composed of a processor and peripherals. We assume that the data from each node is transferred directly to the gateway. If the connection is lost, then the information is stored in a temporary memory. This implies the presence of a storage memory in the nodes.

A gateway is in charge of the report of all nodes and the storage of the information in a file. The

gateway, a more powerful node, should be able to run different algorithms to compute future optimization and better detect defects in the electrical networks. Its computing performance has to be greater than a node.

5.3 QBox: a SystemC CPU model based on QEMU

5.3.1 Introduction

CPU elements are involved in the system that should be simulated. Consequently, SystemC CPU models are required. A way to simulate CPU core in a SystemC simulation is QBox. QBox is an integration of QEMU in a SystemC model. QEMU [143] is an open source system emulator that can be used to emulate various CPU architectures through dynamic binary translation. Contrary to the QEMU-SC solution [76], QBox considers QEMU as a standard SystemC module within a larger SystemC simulation context. SystemC simulation kernel remains the “master” of the simulation, while QEMU has to fulfill the SystemC API requirements. This solution is an open source QEMU implementation wrapped in a set of SystemC TLM-2.0 interfaces. QBox enables the powerful Just In Time (JIT) based CPU simulations to be totally exploited within a TLM-2.0 context. QBox is provided as shared libraries that contain QEMU based CPUs as illustrated in Figure 5.3. Each QBox library contains a specific CPU core and exports a set of TLM-2.0 like 'C' interfaces. A QBox library is instanced in a SystemC simulation context through the SystemC wrapper called TLM2C. TLM2C library provides the C++ TLM-2.0 standard interfaces. It exports TLM-2.0 'C' like interface to the standard TLM-2.0 C++ interface.

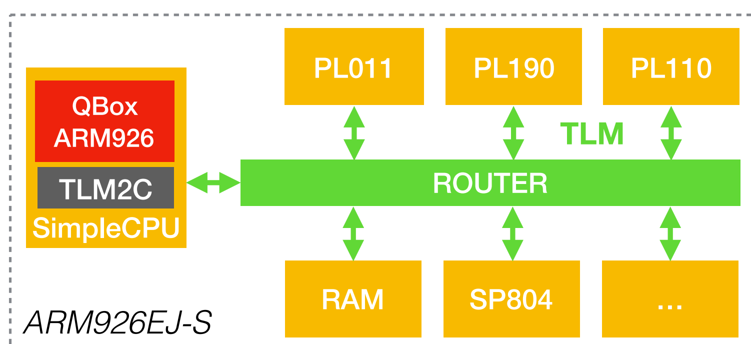


Figure 5.3 – ARM926EJ-S virtual platform architecture with QBox

5.3.2 Time and synchronization in QBox

Inside of QEMU, time is managed in different manners. It can base its source of time on the host clock that is wall-clock time, or the real time clock that is the monotonic clock available on Linux or the VM clock that is the time reference for the guest system. This last clock is paused each time the simulated system stop. The `icount` QEMU is preferred. This approach is based on an instruction counter of the virtual processor to increment the time value.

With `icount` option, time measure is based on instruction counter of the emulated CPU. Unfortunately, the instruction counter is not perfect for the time measurement. When the emulated CPU is idle, for instance the CPU is waiting for an IRQ, no instruction is executed. In this case, the

instruction counter does not move and so neither does the simulated time. To solve this problem, the real time clock is temporary used as another source of time while there is no instruction. This can have a non negligible impact on the simulation time accuracy.

QBox notion of time is typically based on the guest clock. On the other hand, SystemC is purely event driven and its clock moves as fast as events can be processed. Due to different time domains, it is necessary to ensure time synchronization as illustrated in Figure 5.4. To further complicate matters, QBox and SystemC run in separate threads to improve efficiency and simulation speed. QBox takes advantage of the quantum mechanism that is built into the TLM-2.0 standard. It enables a model to be at most one quantum ahead of SystemC's current time. QBox's time increases in parallel to SystemC simulation time in a different host thread. Quantum level synchronization is maintained between threads as presented in the section 4.4.3.2 of the chapter 4.

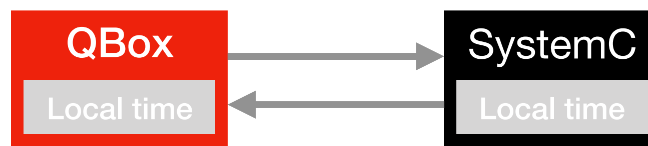


Figure 5.4 – SystemC and QBox local times

Time synchronization is one of the bottlenecks of a virtual platform. Indeed, due to the static length of a quantum, it is necessary to add checkpoint quantums even if there is no event pending in SystemC. In some systems, to handle tightly coupled Input/Output (IO), the quantum has to be reduced. But, this has a negative impact on performance because it increases simulation time. It can also be wasteful if IO is not always used. To ensure this, when QBox or SystemC executions are at a quantum boundary, they use a SystemC wait to synchronize. This is used to wait for the other parties to finish their own quantum. Currently, QBox only supports static quantums. However, in order to avoid redundant and unnecessary checkpoints, it is possible that a dynamic quantum mechanism may be beneficial. This is left as future work.

5.3.3 Multithread

Chapter 4 proposed a solution to enable the parallelism inside SystemC. But, another axis of parallelization is possible inside QEMU and hence QBox. QEMU is used to emulate mono-core and multi-core architectures. To speed up the simulation of these architectures, it is interesting to take advantage of multi-threading. The Multi-Thread Tiny Code Generator (MTTCG) project aims to allocate one host thread to each simulated core to significantly improve performance. This project has been developed by Frederic Konrad [58]. It enables the performance of the host machine within the heart of QEMU itself. MTTCG focused on the ARM architecture. But it can also be extended to all targets within QEMU. More details on the project are given in [58]. It is particularly useful for platforms using SMP. It avoids the redevelopment of the SMP mechanism inside SystemC as it is already present in QEMU.

5.3.4 Impact of multithread for QBox SMP

In order to provide a measure of the speed up with MTTCG, Dhrystone benchmark [13] has been used. Dhrystone is an open source synthetic computing benchmark program used to evaluate the

integer performance of processors. Dhrystone 2.1 has been built with the Buildroot toolchain. The program has then been added to root file system to be run in the guest OS (Linux). 10^7 Dhrystone computations are executed. As the time reported by a virtual machine may not be accurate, an external timer is used to measure execution time, averaging over ten runs. The Device Tree Binary (DTB) has been modified to limit the number of CPU's assigned. The simulation is useful to design a product that does not exist. However, the simulation duration can be slower than real hardware. In our use case, a similar hardware of the gateway was available. It is a board called Versatile Express A9. This board embeds four Cortex-A9 cores. Thus, Dhrystone benchmark has been executed on a version of QEMU without MTTCG, QBox with MTTCG and on Versatile Express A9.

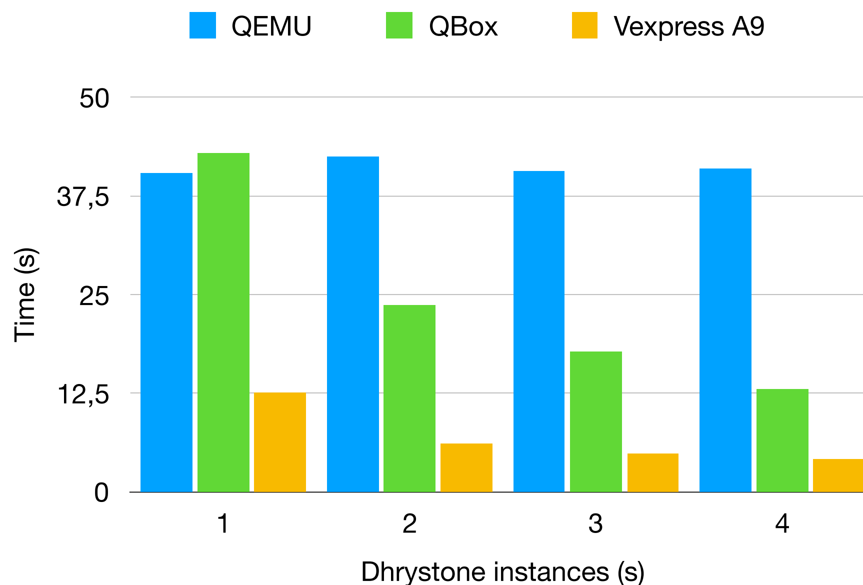


Figure 5.5 – Dhrystone runtime on four cores

As illustrated in Figure 5.5, the previous version of QEMU does not take advantage of increasing the number of CPUs. Indeed, the time to compute the four Dhrystone is approximately the same for all configurations. However both QBox and the real hardware are both capable of dividing the load over multiple CPUs. For a single CPU, QBox is slightly slower than standard QEMU. This is the overhead of ensuing thread safety in QEMU, it is estimated as about 14%. For the case of a four cores CPU, QBox is over three times faster than the existing QEMU. Overall, QBox has now been demonstrated with an impressive near linear speed improvement. Finally, QEMU emulator is around three to four times slower than real board compare to the host machine. Real board cores run at 1.3GHz so the host machine executes one virtual CPU instruction in around ten host instructions. Finally, MTTCG enables fast SMP simulations with an almost linear scale.

5.3.5 Conclusion

This section provided a review of a solution to interface QEMU and SystemC: QBox. QBox can be used as a CPU model inside a SystemC simulation. It supports homogeneous and heterogeneous simulations. To speed up the SMP simulation, a multithread solution has been presented for QEMU called MTTCG. It is now part of QEMU mainline. This solution is then used for the simulation of the system. Thanks to the performance in SMP, it enables to run the CPU cores of the same platform inside a same QBox instance. Thus, it avoids to extract cores one by one from QEMU. This also avoids to implement again the synchronization mechanism of the cores.

5.4 The virtual platform

5.4.1 Architecture

The architecture of the simulation system that should be simulated can be decomposed in two main parts: the node and the gateway. The global architecture is represented in Figure 5.6. Even if there are multiple nodes in the platform, the architecture is always the same. Of course, some parts can be customized through CCI as presented in the next section. In this example, some arbitrary choices have been made.

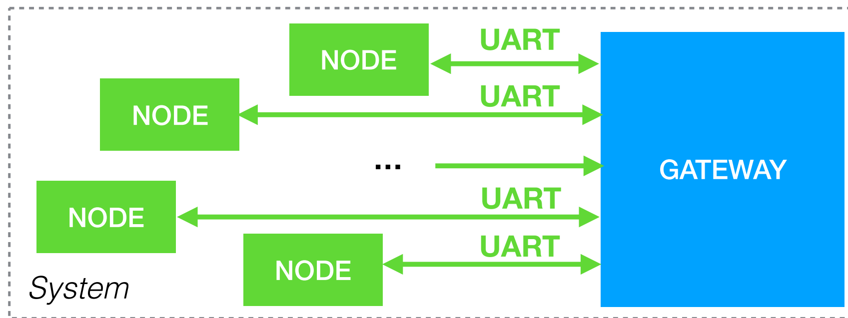


Figure 5.6 – Architecture of the system

The node is based on the SAM3X8E SoC [173]. This architecture contains a single ARM Cortex-M3 core. The interrupt controller is a Nested Vectored Interrupt Controller (NVIC) and is directly integrated in the core. The CPU has been modeled with QBox. As the NVIC is integrated in the core, it is a part of the QBox Cortex M3 CPU model. This means that other SystemC models directly connect their interrupt output to the interrupt input socket of QBox. The SAM3X8E is composed of many features like UART, SPI, Timer, Watchdog, ... However, due to time constraints, not all the parts of the SoC have been modeled. Instead, only those used by the software have been modeled as illustrated in Figure 5.7. It includes the CPU, Random Access Memory (RAM), Read-Only Memory (ROM), UART, SPI, Timer, Power Management and GPIO. All these models have been modeled at the TLM LT level. Basically, registers have been modeled first and then their behaviour according to the data sheet. Models have then been connected through a router.

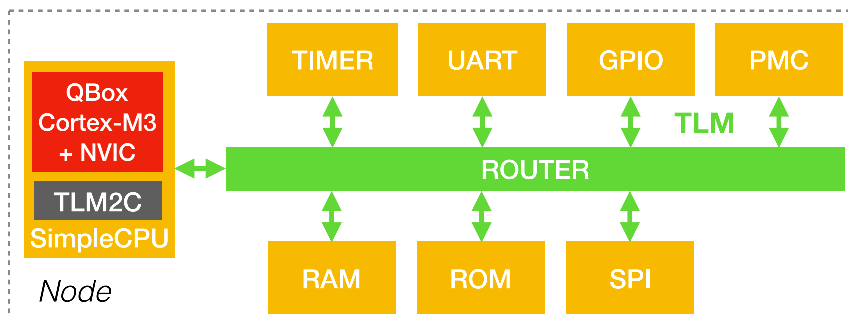


Figure 5.7 – Architecture of the node

The gateway is a Versatile Express A9 board. It is composed of four Cortex A9s. It is a SMP architecture. All cores have been modelled inside a single QBox instance. The platform includes a Generic Interrupt Controller (GIC) to manage interrupts. For the same reason as the node example above, the GIC has also been left in QBox. Equally, as for the node, only the parts used

by the software are modeled as illustrated in Figure 5.8. This includes the CPU, RAM, ROM, UART, GIC, Timer. All these models have been built at the TLM LT level. A TCPSerial model has been connected to the UART model. This enables interaction with the system running on the gateway. Basically, it is connected to the host console (Linux OS). Then, it binds a TCP server on the machine. A “telnet” application can be used to interact with it. All models have been connected through another router. Finally, The TCPSerial model and the platform are instantiated and connected through a TLM UART connection.

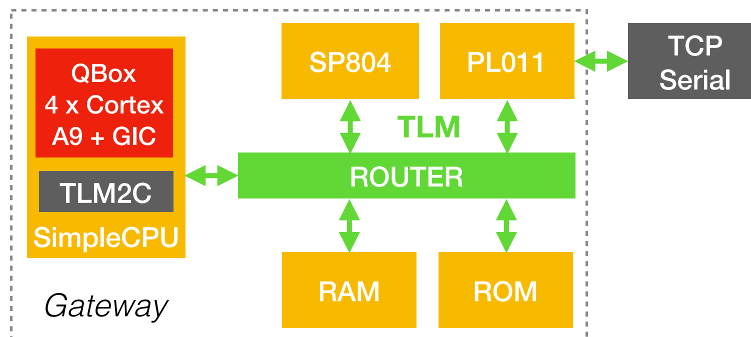


Figure 5.8 – Architecture of the gateway

5.4.2 Configuration

The global overview of the configuration architecture is detailed in Figure 5.9. Basically, different parameters have been implemented in different models. This includes the interrupt number, binary file name, base address, ... Parameters are sets with a default value in the models. However, most of them are initialized and so overwritten during the construction. To do this, a configuration file has been used to load the initial values. A single configuration file is necessary to propagate the values for all parameters. However, different files can be used for more accuracy in the configuration.

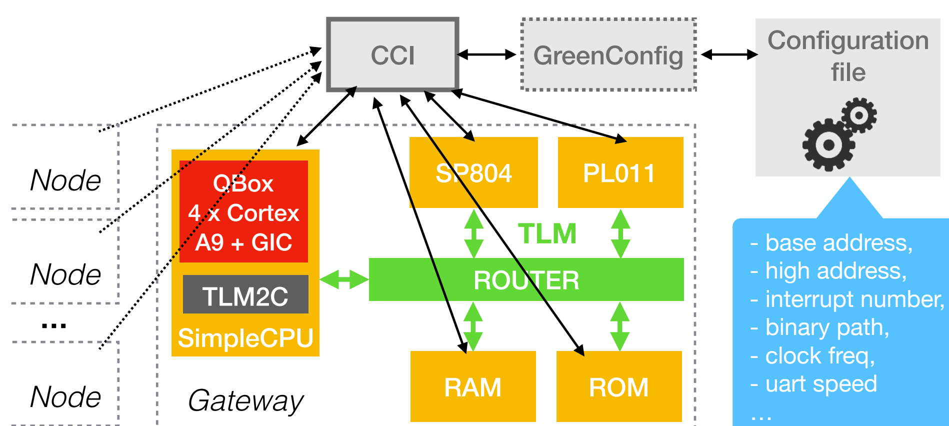


Figure 5.9 – Configuration of models in the gateway

The configuration file is a Lua [107] file and is partially given in Listing 5.1. It is composed of the CPU, the quantum, the QBox library to use, GNU Project Debugger (GDB) arguments, ... As we can see, the CPU core library can be easily modified directly in a file. For instance, a `icount` parameter to fix the time spent between two processor instructions is necessary to modify the frequency of the simulated CPU core. The memory map of the models is specified in the

configuration file as well as the interrupt number.

Listing 5.1 – Gateway configuration file (partial)

```
CPU = {
  library = "cortex-a9.so",
  kernel = "zImage",
  dtb = "vexpress-4a9.dtb",
  rootfs = "rootfs.ext2",
  kernel_cmd = "console=ttyAMA0",
  quantum = 1000000,
  icount = 1,
  gdb_port = 1234,
  extra_arguments = "-smp 4"
}
-- [...]
ram = {
  size = 262144,
  target_port = {
    base_addr = 0x60000000,
    high_addr = 0x70000000
  }
}
uart0 = {
  irq_number = 5,
  target_port = {
    base_addr = 0x10009000,
    high_addr = 0x10009FFF
  }
}
```

This file is read by the GreenConfig library [75]. Initially, this library is used alone. However, with the CCI standard, it has been integrated inside the standard as an external implementation. GreenConfig includes its own parameters called `gs_param`. Connecting the GreenConfig to CCI, it is possible to re-use natively existing code thanks to the CCI API to access these parameters. While a configuration file has been used, it could have been a database, a registry or any other solution to store parameters. Indeed, it is also possible to read the parameters from the command line as provided by [75]. Parameters are sets during the start of the virtual platform. It allows easy scripting for the exploration of different configurations. An example is given in Listing 5.2. This mechanism was not used in the experiments.

Listing 5.2 – Configuration of the parameters through command line arguments

```
./vp --param CPU.quantum=10000 --param uart0.irq_number=10
```

5.4.3 Parallelism

Without any improvements, the described simulation would be executed sequentially. However, according to the proposed use case, that contains many nodes, simulation speed can be very long. To speed up the simulation duration, the execution of the entire system has been split in different threads. This is illustrated in Figure 5.10. Basically, the four cores of the gateway are executed in their own thread inside QBox. They use SMP speedup that is offered by MTTCG.

Each core of the node, hence each QBox is executed in a specific thread. Models like SimpleCPU are represented in two threads because some parts of the model run in the thread of QBox and others inside SystemC. It should be noted that QBox itself is composed of many threads. However, to simplify the figure, only one thread has been drawn. The global split is done manually and the mechanism used to enable the parallelism is the one presented in chapter 4. It is based on an asynchronous event with the global lock in the SystemC kernel.

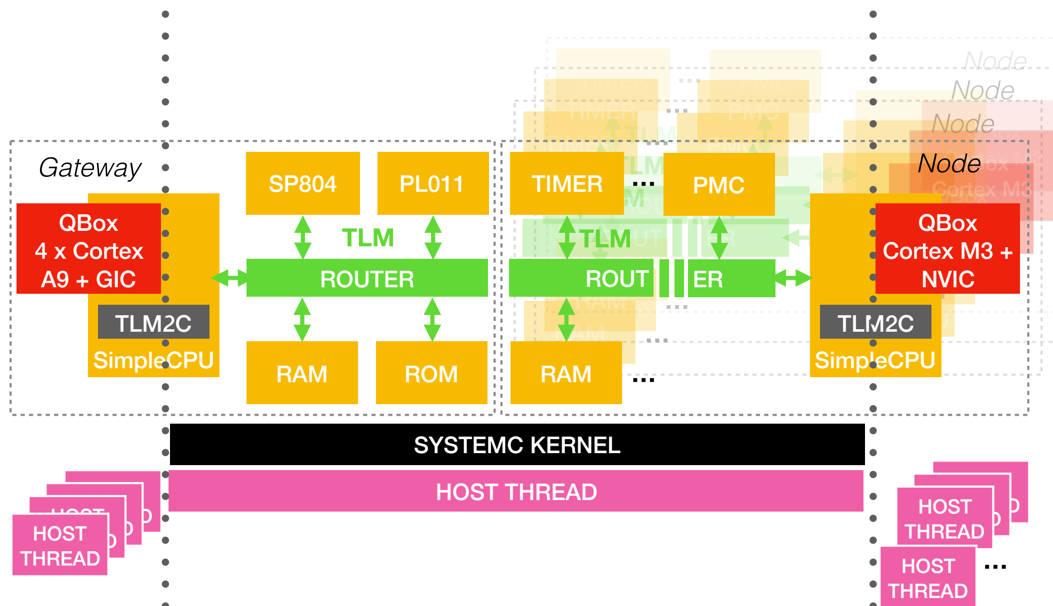


Figure 5.10 – System model based on different threads

Running models in different threads, an almost linear speed up is expected while physical host threads are available to run them. According to the host machine, four cores of the gateway, the SystemC kernel, and the three cores of three different nodes looks like the theoretical limit. However, the machine is able to run more nodes while maximizing the available host performance.

As explained before, QBox provides a TLM-2.0 like interface through TLM2C. Then, another bridge is required to connect it to the SystemC simulation. As illustrated in Figure 5.10, QBox is embedded in a SystemC module called SimpleCPU. This model does the synchronization with QBox for the time and the data. Basically, as QBox is provided as a dynamic library, SimpleCPU loads the right CPU library according to the configuration. Then, QBox is able to run in its own thread. The TLM2C interface is used to forward interrupt requests to the CPU for the gateway only. The TLM2C interface is also used to handle the IO requests from the QBox thread. Care is taken for thread safety. SimpleCPU is also the place where the quantum based synchronization happens which uses the asynchronous event. QBox is configured to trigger an "end of quantum" function in TLM2C and hence in the SimpleCPU module. Basically, it is the place where the asynchronous event is notified.

In this use case, not all models have been parallelized. Instead, the split has been done manually. The parallelism of models works well with time consuming processes. In our use case, it is only the case for QBox. Indeed, RAM and ROM `b_transport` are mainly read and write actions in an array, SP804 (Dual Timer) and Timer `b_transport` are mainly register read/write and rely heavily on SystemC events, likewise the UART modules.

5.4.4 Protocols

The communication between the nodes and the gateway is done through a radio protocol. This link can be done through another specific chip in charge of the radio part. In this case, the communication between the node and the radio link is commonly done through protocols like UART or SPI. However, to simplify the modeling of the platform, the radio protocol is not modeled. The protocol is not the heart of this PhD. Instead, a direct UART communication between the nodes and the gateway has been modeled. To model this protocol, a TLM UART communication has been used. The SPI implementation has also been used for the communication with a SPI memory 25AA256 from Microchip. This memory is used to store the sensor information if the link is lost. The usage in models as well as interconnection of sockets is easy. The useful data is directly available in the TLM payload.

5.5 Experimental results

5.5.1 Introduction

The gateway runs Linux 4.1 built with Buildroot. The DTB file has been updated to use only those devices that are available in the virtual platform. This prevents Linux from searching for devices that are not present. Devices include: the four Cortex A9, System Controller, GIC, SP804 and PL011 (UART). The DTB file is specified as a configuration parameter.

The node runs a bare metal application built with a GCC cross compiler. Peripherals are configured directly through registers. Benchmarks have been executed on a computer running an Intel Core i7 6700HQ and 16 Gb of memory. All the presented results have been measured with wall-clock time and averaged over ten executions. On the startup of the simulation, a time is left to connect a client to the Linux terminal for the gateway. This time is constant and has been removed.

5.5.2 Quantum

In this subsection, the impact of the quantum on the simulation duration is evaluated. This value has an impact on the simulation speed. Thus, this step can be useful to find the best quantum value to better take advantage of the host machine. Figure 5.11 shows the impact of the quantum duration on Linux boot on the gateway. The smallest boot time is obtained with a quantum around 10ms. For smaller quantum duration, QBox runs IO more frequently. It increases execution time and decreases simulation speed. However, as quantum duration increases, Linux sees fewer and fewer timer interrupts which are used to synchronize and schedule processes. As a result, processes which are spinning waiting for others, or waiting for a timer interrupt potentially spin for longer. It slows the overall simulation to the point at the far right hand side of the graph where Linux is no longer capable of running.

Figure 5.12 shows the impact of the quantum on the entire platform with a different number of nodes. It aims to show the impact of the quantum depending of the number of nodes. The update of the number of nodes and the quantum are easy, thanks to CCI. An edit of the file is enough between each run of the simulation. As the timer frequency is not the same for the gateway and the node, there is an impact compared to the previous curve. The curve is always 'bath' shaped. As the clock period of the Linux gateway is smaller than the one in the node, 10ms is still the

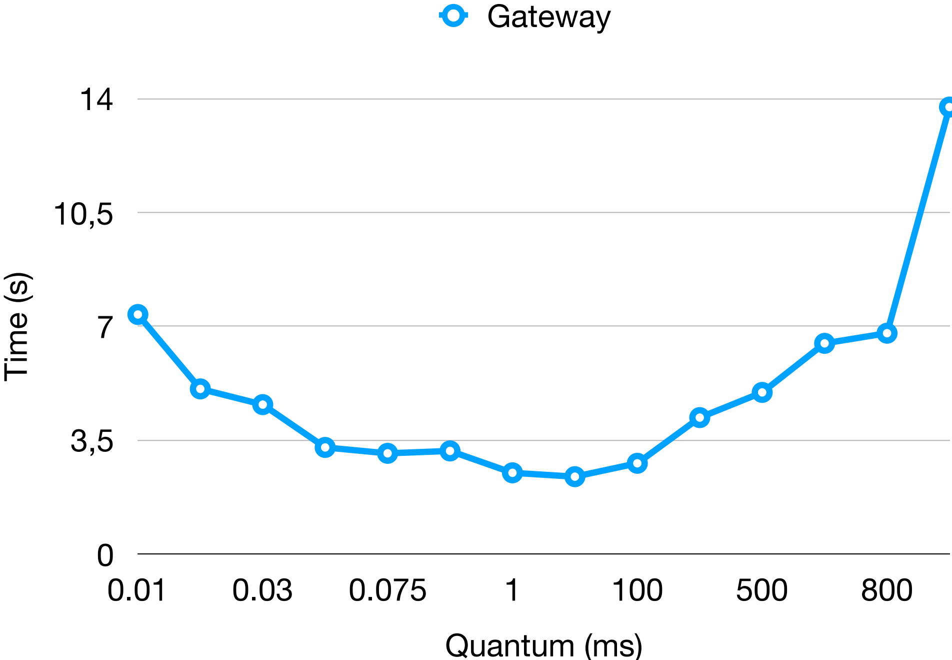


Figure 5.11 – Quantum impact on Linux boot (gateway only)

minimum. Thus, if the gateway and the node used a same minimum periodic clock in both system, it would improve the simulation speed. However, the clock should not be chosen for the simulation speed.

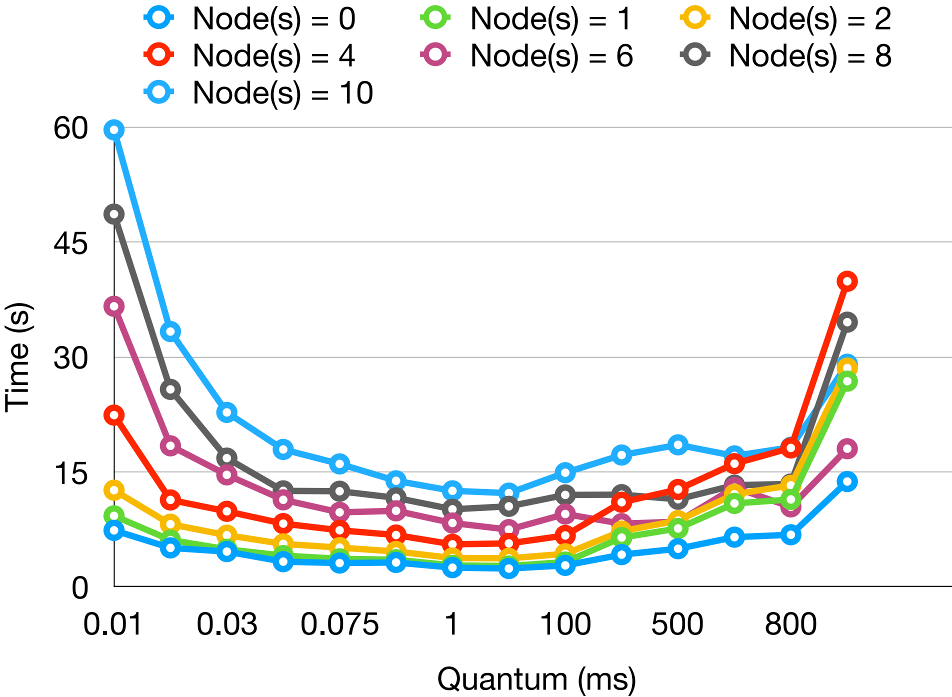


Figure 5.12 – Impact of the quantum and the the number of nodes on the boot time

It can be noted that the smaller the quantum, the more it has an impact on the simulation. Indeed, the boot time difference between the different curves decreases til the minimum and is almost the same at that point. Adding more nodes, the quantum impact is smaller. The number of nodes has

finally more impact on the simulation time.

The simulation duration has been measured after the start of the simulation (sc_start). However, the time to initialize different modules is non negligible. This time has been measured. It is around 400ms for the gateway and one node. This number increases with the number of nodes. It is around 15s for ten nodes and the gateway. Finally, as the selection of an appropriate quantum is hard, it takes experimentation, and is system dependent. Typically, a 'bath' shaped graph can be expected. Users may choose a lower quantum if higher timing fidelity is required.

5.5.3 Trace using CCI parameters

During the simulation, it is sometimes useful to track the change of signals, states, registers, etc. In our use-case, the UART activity will be tracked during the simulation. Fortunately, SystemC includes native mechanisms to record a time-ordered sequence of value changes during simulation. This trace solution uses the Value Change Dump (VCD) file format. Basically, the trace class provided by SystemC enables to set the time unit for the trace file. The time unit represents the sample period for all tracked signals. If this value is small, it can slow down the simulation as it directly uses the SystemC time from the kernel. In fact, the trace mechanism uses SystemC phase callbacks. It is called every time the time moves. Consequently, it doesn't work well with TLM-2.0 and the quantum. Instead, CCI parameters can be used.

Listing 5.3 – Registration of callbacks for trace

```
1  std::vector<cci::cci_param_untyped_handle> parameters =
2          cci::cci_broker_manager::get_broker(
3                  cci::cci_originator("sc_main")).get_param_handles();
4  std::vector<cci::cci_param_untyped_handle>::iterator parameterHandle;
5  for (parameterHandle = parameters.begin(); parameterHandle < parameters
     .end(); parameterHandle++) {
6      // ... Regex like filter on name
7      parameterHandle->register_post_write_callback(tracking_callback);
8      // ... Typed callback registration
9  }
10
11 [...]
12
13 void tracking_callback(const cci::cci_param_write_event<>& ev)
14 {
15     /* std::cout << ev.param_handle.get_name() << " updated from "
16                << ev.old_value << " to " << ev.new_value
17                << " by " << ev.originator.name() << std::endl; */
18     // Store to a VCD file
19     // ...
20 }
```

To track parameters from all nodes, the CCI API has been used. This code has been placed in the sc_main after the creation of modules. It is partially presented in Listing 5.3. Basically, it retrieves all parameter handles from the broker handle using sc_main as the originator, as we are not in a SystemC hierarchy. Then, it registers the post write callback. Thus, the callback will be triggered on each parameter change. Another solution could have been used here: the broker is called every time a parameter is created, and it also provides a callback for the parameter creation.

Thus, it is possible to register a callback and add the post write callback during the parameter registration. The callback is untyped as parameters are not the same type, and uses `cci_value`.

Table 5.1 – Comparison of state tracking between SystemC trace and CCI parameters

	Simulation time (s)
No trace	51.3
`sc_trace`	64.8
CCI parameter (untyped callback)	83.1
CCI parameter (typed callback)	67.3

A version using `sc_trace` and another using CCI parameters has been evaluated. Results are given in Table 5.1. The time unit of the trace mechanism has been set to 1 ps. The simulation time with untyped CCI parameter callbacks is around 1.25 times slower than SystemC trace mechanism. Even if `sc_trace` mechanism evaluates signals on each time update, it doesn't happen frequently in our use case. Indeed, most models use time decoupling. For typed CCI parameter callback, the time is about the same. In this case, `cci_value` is not used and saves time. More flexibility is provided by the CCI mechanism. Indeed, it is possible to dynamically disable and enable callback during the simulation. This option is not possible with `sc_trace`.

5.5.4 Impact of CCI on the simulation execution time

In the chapter presenting the CCI standard, the implementation was benchmarked in an intensive use of CCI environment. Here, the environment is composed of several tens of parameters. Some are used only for initialization such as binary, processor frequency, etc. Others are used more intensively during the simulation. This is the case for the module addresses on the memory map. In order to evaluate the impact of the CCI standard and more precisely the parameters on the simulation, we temporarily disable and replace some of them by variables. The simulation is composed of ten nodes with a quantum set to 10ms. There are 833 parameters. The simulation time and the load time was measured. The load time is the time between the start of the application and the call of the SystemC callback `start_of_simulation`. It is called by the kernel at the very start of simulation.

Globally, the impact of CCI on the simulation boot time is negligible. It was not possible to measure a concrete difference. The difference is about some milliseconds for the load of 833 parameters. The impact is less than 1% on the load time. Then, there are about 300000 reads of the parameters until the boot of Linux completes. This is again negligible compare to the wall-clock boot time, about 10 seconds in this use case. The impact is low and difficult to measure. It represents about several tens of milliseconds overhead. Compare to the performance analysis done in Chapter 2, CCI parameters are not involved continually. Even if CCI "objects" are heaver in memory and integrated into flexible mechanisms, it does not reduce performance significantly.

5.5.5 Exploration of the impact of the node CPU frequency

In this section, the impact of the CPU frequency on the data exchange between the node and the gateway is studied. The gateway expects a value every 500ms from all nodes. However, to

avoid really long simulation time with low frequencies, this value has been changed to 10ms in this experiment. Consequently, it is expected that if the frequency of the node processor is too slow, the packet for the gateway can miss the requirements. To study this impact, the platform has been stressed with different values of `icount`. Moreover, the test has been run for different numbers of nodes. The simulation time has also been measured. The quantum was set to 10ms. The number of packets sent over UART was limited to 100 for each node.

Table 5.2 – Impact of the node CPU frequency and the number of nodes. VP = Valid Packets, ET = Execution Time (s)

CPU frequency	0.5 Ghz		15 Mhz		244 kHz		7.62 kHz		3.81 kHz	
Node(s)	VP	ET	VP	ET	VP	ET	VP	ET	VP	ET
1	100	4.39	100	3.96	100	3.73	66	7.66	0	14.98
10	1000	20.55	1000	12.17	1000	10.35	660	16.11	0	23.5

Results are given in Table 5.2. As the frequency is reduced, less packets meet the time criteria imposed by the gateway. This is an expected result. With the increase of the number of nodes, the results are almost the same. As UART data is exchanged using TLM blocking transport, and the FIFO of the UART in the gateway is limited, some parts can be dropped. Indeed, the reduction of the frequency of the gateway increases the processing time of UART packets. This implies some packets are lost. The execution time is also not linear. The `icount` option of QBox and so the frequency of the node CPU impacts the simulation speed. The results show a bathtub curve with a minimum. One more time, the configuration of the CPU frequency is easy, thanks to CCI and GreenConfig. An edit of the file is enough between each run of the simulation.

5.5.6 Evaluation of the improved TLM standard

In this section, the improved version of the TLM standard is evaluated within our use case. A UART link is used between the nodes and the gateway. However, this protocol is not available natively with the standard. Instead, an implementation has been done with the improved one. However, for comparison purposes, UART protocol has been implemented with the existing version of the standard. Thus, it was possible to measure the impact on the simulation and the modeling. Indeed, the contribution not only influences the simulation time but also the modeling time. The number of lines of code, duplicate code, etc. That's why multiple criteria has been examined:

- the number of lines to implement the UART protocol,
- the degree of code duplicate,
- the time to exchange 1000 millions UART frames without updating the configuration,
- the time to exchange 1000 millions UART frames updating 1000 times the configuration.

Table 5.3 presents the obtained results. The number of lines to describe a UART protocol with TLM is divided by more than two with the improved version of TLM. Indeed, it removes code duplicate for the extension mechanism. It also implements only relevant interfaces. The execution time is about the same for all versions. Indeed, the function call TLM mechanism is the same. However, numbers are a bit different when frequent meta data updates of a protocol happen. In that case, with CCI and its callback mechanism, the check of meta data is simplified.

Table 5.3 – Evaluation of the improved TLM standard with UART

	Number of lines	Code duplicate	Execution time (s) for 1000 M frames	Execution time (s) for 1000 M frames with config update
TLM 2.0	~830	● ● ○	26.43	31.56
Improved TLM 2.0	~360	○ ○ ○	27.57	31.32
Improved TLM 2.0 with CCI	~410	○ ○ ○	28.02	29.68

5.5.7 Exploration of the parallelism in the simulation

Figure 5.13 illustrates the execution time of the simulation as a function of the number of nodes. The simulations have been executed from 1 to 10 nodes. The quantum is the same for all simulations and is 10 ms. The speed up is not really linear. In fact, in this experiment, IO are highly used. It involves many synchronizations between QBox and SystemC for thread safety. Moreover, as there is only one instance of SystemC, SystemC is the bottleneck. It can be noted that the CPU of the host machine is not optimized in this case. However it constitutes the worst case, where the nodes only make IO. A thread safety data passing between thread would help to improve performances.

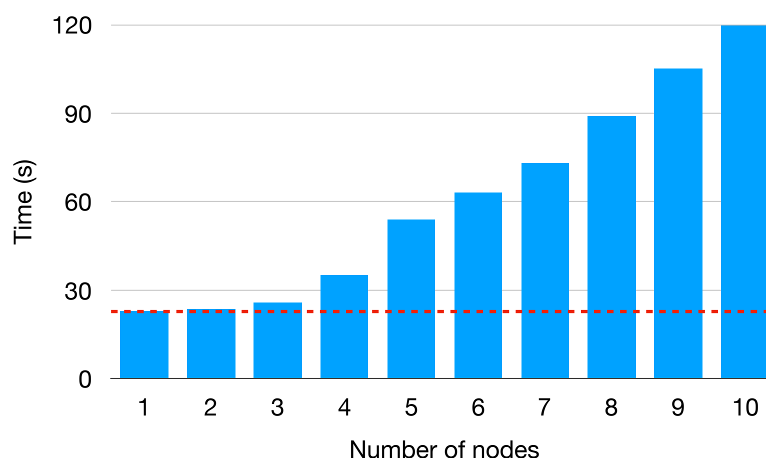


Figure 5.13 – Execution time of the simulation in function of the number of nodes - IO intensive

Another experiment has been done. In this case, IO are not used often but the CPU is used intensively. Results are given in Figure 5.14. In this case, the obtained result is the expected curve. The simulation time is almost constant until a number of nodes. Then, the simulation is slower. Indeed, more threads are executed in the simulator than the host can provide physically (and logically). Even if some threads are not always busy, the OS needs to schedule different threads.

Up til the limit of the host machine, the asynchronous synchronization mechanism provides an almost linear speed up. Then, the simulation is limited by the performance of the host machine. In that case, multiple hosts can be used to overcome this performance issue. In Chapter 4, it is demonstrated that it is possible to use multiple SystemC kernels. As the connection between the

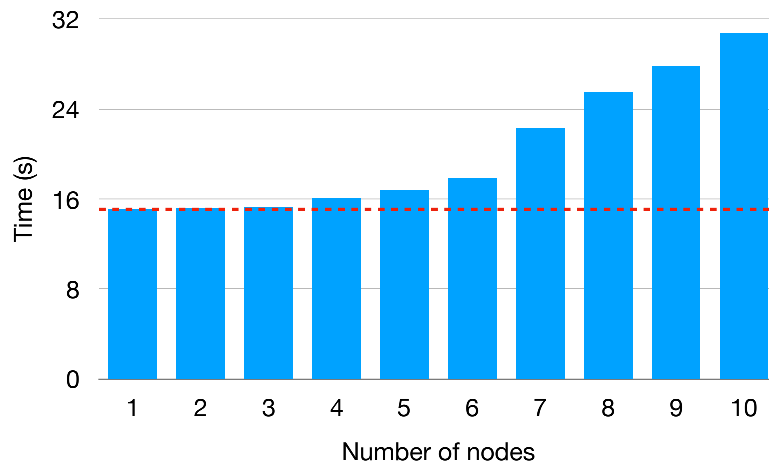


Figure 5.14 – Execution time of the simulation in function of the number of nodes - CPU intensive

two instances is done through POSIX sockets, a similar mechanism looks like a good candidate here. This experiment has not been done as the focus is on the efficiency of the mechanism itself. A good speedup is also expected.

5.5.8 Enhanced quantum keeper

The improved version of the TLM Quantum Keeper has been tested with the timer in the gateway containing only one core. Indeed, due to care that has been taken for thread safety, it is not finally trivial to make it work. In this implementation, the timer accesses the enhanced version of the TLM Quantum Keeper, called the Quantum Keeper Plus (QKP). The QKP is provided as a constructor parameter on instantiation. It is the one used by the SimpleCPU module. It is not perfect and has been discussed in the Chapter 4. In the future, it can be grabbed through the CCI mechanism exporting it as a parameter.

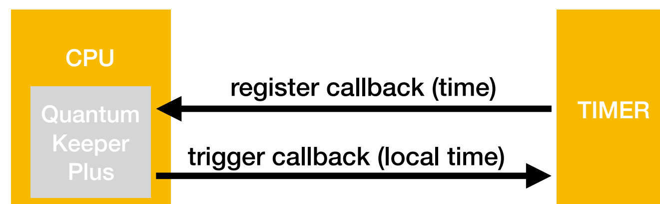


Figure 5.15 – Quantum Keeper Plus callback mechanism

To use the QKP, the timer module source code has been updated. On the initial version of the timer, a callback is called each time a write is done in the configuration register of the timer. Thus, it is possible to setup a SystemC event in the future, equals to the period of the timer. In this updated version, the timer no longer uses a SystemC event. Instead it uses a kind of TLM event. It registers a callback to the QKP, thanks to the API available in the Listing 5 of the Appendix A.2.3. The mechanism is illustrated in Figure 5.15. The notification is configured for a time equal to the timer period. Then, as the local time moves in the SimpleCPU module, the timer callback is finally triggered. Indeed, the time moves in the SimpleCPU module as a result of annotated time from transactions, simulation time moving locally, etc. As the time moves locally in the SimpleCPU, the callback is accurate. Indeed, if the local time is not updated between two quantum synchronizations, it can impact the timer accuracy. Note that if the period is greater than

the quantum value, then the notification mechanism has almost no effect. Finally, the registered callback in the timer is called. The timer generates its IRQ and re-register the callback for the next timer cycle.

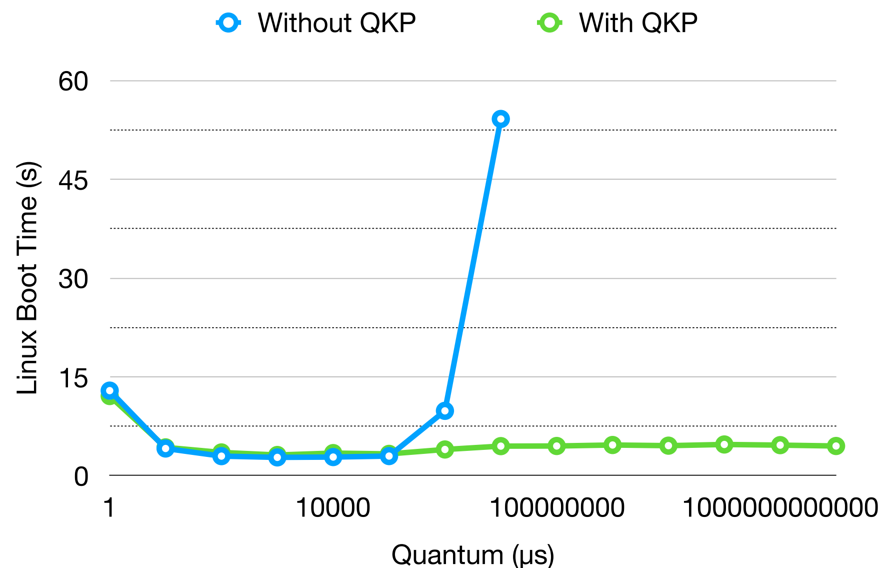


Figure 5.16 – Quantum impact on Linux boot with Quantum Keeper Plus

This mechanism has been measured with different values of the quantum. It has been compared to a system without the QKP. Results are given in Figure 5.16. Basically, the bathtub of the Figure 5.11 is also seen. However, in the case of the QKP, the end of the shape is different. Previously, after a certain value of the quantum in Figure 5.11, the boot time increased. Here, the boot time is almost the same. It means that the quantum has less impact on the boot time. Indeed, as the timer is no longer based on the SystemC time, the value of the quantum is not correlated to the timer period. Instead, the timer is directly related to the local time of the initiator, the CPU. With an almost infinite quantum, the number of synchronization with the SystemC is so hugely decreased.

In the case of multiple CPUs used in the same SoC, the timer can be driven by all of them. In this case the question arises as to which is the most legitimate initiator to notify the timer. With Linux, interrupt handling is randomly load balanced through all CPU. It is a software task. Therefore, it is difficult to fix a CPU. For the moment, this problem has not been solved.

Finally, the improved quantum keeper should be more easily findable by modules. A CCI object looks like a good candidate. It allows models to find the correct quantum keeper in the hierarchy. It also avoids passing a quantum keeper to all models whether they need it or not. Thus, it avoids an invasive change to large quantities of models. This approach enables new models to be written, taking advantage of the new quantum keeper. It also removes dependencies on the quantum duration. It also enables co-existence with current models.

5.6 Conclusion

An illustration of different contributions presented in the previous chapters is given. For this, the modeling of an IOT platform has been done. It is composed of many nodes and a gateway that are modeled inside virtual platforms. Each virtual platform is composed of at least one CPU. Thus, a solution to re-use the CPU models available in QEMU has been detailed. It facilitates the work

of the designer. QBox has been featured in [62, 63, 64, 58]. Virtual platforms are finally connected themselves through non memory mapped protocols.

The configuration standard has been applied for the configuration of the addresses of the different devices on the memory map, the size of the RAM and ROM, the interruption numbers, and many other parameters. Then, the enhanced version of TLM enables an easy interconnection between different virtual platforms. For this, the UART protocol has been modelled and then implemented. Next, to speed up the simulation, an asynchronous event mechanism has been applied. It enables the synchronization between CPU models that run in different threads. A SystemC model around QBox ensures the synchronization.

The gateway has been evaluated first alone. It shows the impact of the quantum as well as the one of the QKP. Thus, an optimal value of the quantum has been found and is directly correlated to the configuration of the timer. Then, the entire platform has been evaluated. In this case, the impact of the quantum with many nodes is less significant. Then, the exploration on the impact of the CPU frequency has been done, thanks to CCI parameters. Next, an almost linear speed up of the entire simulation is obtained thanks to the split of CPU cores in different threads. Thus, precious time of designers is saved by the reduction of the simulation duration. Finally, this chapter contains advances in SystemC/TLM virtual platforms through configuration, communication and parallelism contributions. For our point of view, they are the basic of the next generation of virtual platforms.

Conclusion

***

Summary

In conclusion, with the multiplication of features integrated in a single chip, it is necessary to improve the design flow of a SoC. It includes virtual platforms as they are now part of the design flow. The research topic of this PhD focused on the improvement of virtual platforms. Thus, several missing points of virtual platforms were explored and solutions were proposed. Contributions are illustrated in Figure 5.17. They are also summarized below.

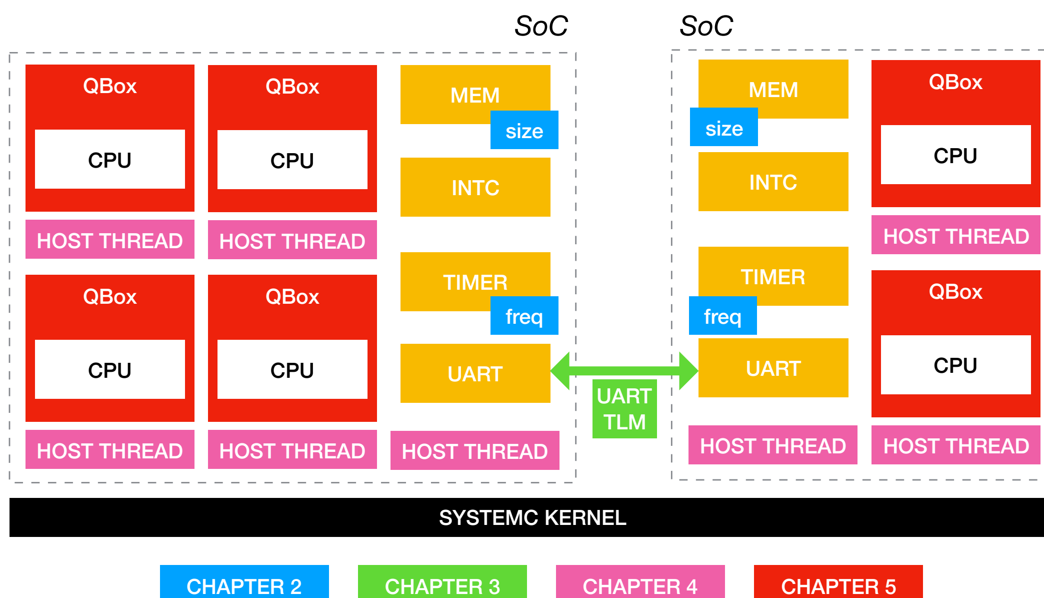


Figure 5.17 – Summary of contributions through virtual platforms

The 1st chapter provides an overview of a SoC design flow. SoCs are now commonly used in current objects. They include more and more features. Thus, this chapter pointed an increase in the design complexity of these chips. To help designers, methods based on the SystemC and TLM standards have been introduced. These methods enable to secure the development as soon as possible. They also aim to reduce the risk of failure of an architecture. Consequently, they are becoming essential methods. Virtual platforms are now at the heart of these methods. However, this chapter also pointed that some current requirements in the virtual platform domain are not met.

The 2nd chapter manages one first issue of current virtual platforms: the configuration. Indeed, related works have highlighted a demand for configuration of SystemC models. It has detailed some configuration solutions for SystemC / TLM. However, the solutions were not all equivalents. Some were specific, others had basic features, etc. Moreover, they did not necessarily have

the vocation of being interoperable. This makes it more difficult to use several solutions inside a same simulation. That is why, an interoperable solution has been proposed. This solution aims to meet the needs in terms of features as well as backward compatibility with current solutions. This solution is part of the CCI standard. It has been based on previous works that was paused for several years. To propose a complete solution, a parameter storage mechanism has been defined. Parameters have been integrated natively with SystemC hierarchy. Fortunately, other parameter storage solutions can be integrated, thanks to the standardizing of the interfaces. A broker mechanism has been introduced to manage the parameters. Callback functions enable the monitoring of the parameters for their values as well as their creation / destruction. The CCI parameters have been evaluated. It showed that the use of CCI parameters is non-negligible compared to a native type. However, these results have to be put into perspective in a concrete use of a virtual platform, not solicited exclusively.

The 3rd chapter contain various solutions to model non memory mapped protocols in virtual platforms. These protocols are commonly used for communications between different SoCs. However, the related work highlighted a lack in the modeling of these non memory mapped protocols in the virtual platforms. In a similar way as TLM-2.0 standardization for memory mapped protocols, it was naturally envisaged to apply a similar principle for different families of protocols while maintaining the backward compatibility. For this, several points have been clarified, beginning with the definition of a TLM-2.0 transaction. Then, a study of different non memory map protocols highlighted new requirements. A thorough analysis of TLM-2.0 standard then showed that it is necessary to update it in order to facilitate the management of these new protocols. Indeed, the current architecture of the standard does not make it easy to extend. The current version of TLM-2.0 required to redevelop existing code, or to use inappropriate parts with non memory mapped protocols. That is why, an improvement in the architecture of the standard has been proposed. The backward compatibility has been maintained. Then, to better manage the static parts of a communication protocol, a new method has been proposed. It consists of the use of CCI parameters to manage the meta-data of a protocol. It enables to reduce the number of fields exchanged during each transaction. It also enable to detect more easily and quickly a bad configuration.

The 4th chapter details a proposal to speed up the simulation of virtual platforms. Since SoCs are composed of always more features, it is required to simulate in reasonable times. For this purpose, relate works listed the various existing solutions. Several families of parallelization were highlighted. However, as with the configuration, the solutions were not built in order to be interoperable. In addition, mixing several of these solutions could result in deadlock. This is why an interoperable mechanism is required. To be independent of the synchronization algorithm between the different blocks running in parallel, the reflection takes place on a fundamental element of SystemC: the event. The proposal is to add the support of asynchronous event in the scheduler, missing in the current version of SystemC. They aim the scheduler to pause and resume the models between different threads / kernels. Integrating this solution in the SystemC kernel, the lock is no longer specific to the solution but rather integrated into the kernel. A first contribution enabled to add a new type of event, asynchronous event (`sc_async_event`). However, the semantics of this proposal are complex and deadlocks can appear in some cases. That is why, another proposal was made. It does not adds natively asynchronous events but instead enables to create them. It allows to simplify the semantic and its usage. This mechanism finally proved to be also faster by a simpler implementation in the SystemC kernel. This contribution was integrated into the SystemC 2.3.2 POC. Finally, an improved version of the quantum keeper has been proposed. It enables a notification system from a TLM-2.0 initiator, likely a CPU.

Finally, the 5th chapter finally provides a concrete application to illustrated the different contribu-

tions. A complete IOT like platform has been modeled and simulated. The platform consists of different low power nodes and a gateway. The first contribution enables to configure the position of the different modules on the memory map, the size of the memories, the binaries running on the different nodes, etc. The platform elements are connected themselves by non memory mapped protocols. The second contribution enables to model these protocols easily and neatly. SPI communications with the memory as well as UART communications with the gateway were implemented. A contribution, presented in the 5th chapter, enables to use QEMU as a native SystemC CPU model. Unlike other solutions, SystemC remains the master of simulation. QEMU is seen like any other SystemC model. In order to take advantage of the power of the host machine, the simulation of these different nodes has been split on different threads. Each CPU model instance run in a different threads. It enables an almost linear acceleration.

To conclude, even if the virtual platform concept has been presented many years ago, a constant evolution is required to meet new requirements. SystemC and TLM-2.0 standard constitute the heart of the virtual platform principle. For theses reasons, these standards have been extensively studied to be improved during the PhD work. Indeed, they constitute the basic of the modeling. The presented contributions should facilitate the designer work and improve the interoperability while saving its precious time.

Perspectives

Following the contributions presented in this manuscript, many perspectives exist. A list is given below.

First, the configuration standard has been presented in the 2nd chapter. It enables the configuration of models through parameters. Parameters can be used for many applications. For example, it includes the input clock frequency of a timer, the size of a memory, the type of a cache, etc. All these parameters are key point of models. They can change during the simulation time. Callback mechanism can be used to track any changes. However, the CCI standard does not only aim configuration. It also includes the control and the inspection. Indeed, another domain is currently missing in this standard: register modeling. With a standard for register modeling, it would be possible to create, update and track in a standard way all registers as parameters. It would increase the interoperability between different solutions. Moreover, some parts of CCI standard could be re-used like the callback mechanism. Indeed, to model the behaviour of a register, pre/post read or pre/post write callback can be applied. Luckily, this part is not linked to the configuration but instead part of the core of the standard.

Second, the improvement of TLM-2.0 standard facilitates the modeling of non memory mapped protocols. The CCI standard has also been used for meta-data exchange. However, this improvement has still not be pushed to the LWG as a proposal. This is the logical next step. Software protocol has been quoted in the 3rd chapter. It is currently hard to handle these protocols without transactors. This issue can be linked to the pin multiplexing remaining issue. Even if some tracks have been given, it does not constitute a concrete solution. Instead, this issue should be considered as a whole. Finally, all existing non memory mapped protocols are not modeled. Instead, the scope has been reduced to the most used ones. Some protocols are open and others are proprietary. Even if the protocol standard is available freely, an implementation provided by the standard owner themselves would be a benefit for the modeling community. This constitutes today a political issue.

Third, to speed up the simulation performance, an asynchronous mechanism has been presented in the 4th chapter. This mechanism has been merged in SystemC 2.3.2. It is currently considered as experimental. This stage enables to bench a feature, get feedback and then consider or not its integration in the IEEE standard. This is the next logical step. Contrary to the POC, a mechanism in the standard means that any other implementations of SystemC must include it. Currently, only the POC is guaranteed to have it. The presented mechanism is efficient and simple to use. However, it implies a manual split of the simulation. In the case of really complex simulations containing a lot models, it can be time consuming to find the best balance in the split in term of performance. An automatic split of models in different threads should help designers.

Appendix

***

A.1 CCI context

A.1.1 The working group

The CCI standard for SystemC has been in the making for almost a decade within Accellera, yet it is a fundamental and relatively simple requirement. The CCI WG is in charge of the definition of standards that allow tools to interact with models in order to perform activities such as setup, debug and analysis. It aims to introduce features to enhance the exchange of information between SystemC models and external tools. The CCI WG aims to cover configuration, inspection, control and the interoperability of model interfaces with current tools. The working group initially focused on the configuration part (since 2009). After a year collecting requirements, work began based on [163]. However, the WG has been dormant for a number of years and has only recently revived. In the meantime, requirements have changed and my activities with the WG began here.

The CCI WG has summarized its initial requirements in [38]. The initial motivation was focused on the instrumentation of models from tools of different vendors. CCI aimed to provide a model configuration mechanism that support both creation-time and run-time configurations, not just at compile-time, configuration. A standard interface to interact with tools and to allow parameter values to be loaded from through a tool (eventually from a configuration file or GUI). Some examples that included the dynamic instantiation of models, parameters such as the size of memories, the dynamic memory map, etc. The main concept of a parameter is assumed by the CCI WG requirements, as a Name-Value Pair (NVP). It means that a *parameter* is the association of a *name* and a *value*.

The fundamental building block of the CCI standard is a SystemC element that can be configured, interrogated. Changes can be notified, through a standard interface (both from the tool and from other parts of the model). These elements are generally referred to 'parameters' (or *params*). While this basic approach is quickly agreed upon, the history of the CCI standard, much like any other standard, is about the details. There are many aspects of a parameter that are not at first apparent. For instance, it should be accessible both via a typed, and an un-typed (e.g. JSON [179]) interface in order to enable the read and write of valuse without know its type. Parameters should be available from different places in the SystemC simulation and from external tools (also called tools). A mechanism to find parameters is also required as it has been stated in [118].

A.1.2 The CCI standard

This section can be considered as a an introduction of the standard. It walks through the more important concepts behind CCI and explains how it can be used quickly by showing a simple example in action. If a configuration mechanism has been already used before, the Listing 4

should be natural to read. The example is composed of two modules. The first module is a timer that contains two parameters : the frequency of the clock and the duty cycle of the clock. The second module is the configurator that specifies the timer. One of the main goal of the standard is to keep the parameter organized and well structured in the entire hierarchy. It enables a complete design to easily evolve over the time by avoiding the mixing of different configuration systems in various parts of the entire design. The responsibility of the designer is to write the code that maps configurable elements of a module to the CCI standard. The code to execute is defined as C++ methods and C++ classes to instantiate. The required code to use the standard is a header inclusion as listed in Listing 4.

Listing 4 – A timer and its configurator using CCI

```
1  #include <systemc>
2  #include <cci_configuration>
3
4  SC_MODULE(timer) {
5  public:
6      SC_CTOR(timer):
7          clock_frequency("clock_frequency", 10),
8          clock_duty_cycle("clock_duty_cycle", 0.2)
9      {
10         clock_frequency.add_description("Frequency of the clock");
11         clock_frequency.add_metadata("unit", cci::cci_value("Hz"),
12                                      "Unit of the parameter (Hertz)");
13
14         clock_frequency.add_description("Duty cycle of the clock");
15         clock_duty_cycle.add_metadata("min", cci::cci_value("0"),
16                                       "Minimum value");
17         clock_duty_cycle.add_metadata("max", cci::cci_value("1"),
18                                       "Maximum value");
19
20         clock_frequency.register_pre_write_callback(
21             &timer::duty_cycle_update, this);
22
23         SC_THREAD(execute);
24     }
25
26     bool duty_cycle_update(const cci::cci_param_write_event<float> &ev)
27     {
28         if(ev.new_value >= 0 && ev.new_value <= 1) {
29             return true;
30         }
31         return false; // Reject the new duty cycle if the value is
               invalid
32     }
33
34     void execute()
35     {
36         // ...
37     }
38
39 private:
40     cci::cci_param<int> clock_frequency;
41     cci::cci_param<float> clock_duty_cycle;
42 };
```

```
43
44 SC_MODULE(timer_configurator) {
45 public:
46     SC_CTOR(timer_configurator):
47         m_broker_handle(cci::cci_broker_manager::get_broker())
48     {
49         SC_THREAD(execute);
50     }
51
52     void execute()
53     {
54         wait(20, sc_core::SC_NS);
55
56         // Configure the frequency
57         cci::cci_param_handle frequency_param_handle =
58                 m_broker_handle.get_param_handle("timer.clock_frequency
                    ");
59         sc_core::sc_assert(frequency_param_handle.is_valid());
60         cci::cci_param_typed_handle<int> frequency_typed_param_handle =
61                 cci::cci_param_typed_handle<int>(frequency_param_handle
                    );
62         frequency_typed_param_handle = 100;
63
64         // Configure the duty cycle
65         cci::cci_param_handle duty_cycle_param_handle =
66                 m_broker_handle.get_param_handle("timer.
                    clock_duty_cycle");
67         if (duty_cycle_param_handle.is_valid()) {
68             // Update the param's value to 0.3
69             duty_cycle_param_handle.set_cci_value(cci::cci_value(0.3));
70
71             // Update the param's value to 1.4 (will be rejected)
72             duty_cycle_param_handle.set_cci_value(cci::cci_value(1.4));
73         }
74     }
75
76 private:
77     cci::cci_broker_handle m_broker_handle;
78 }
79
80 int sc_main(int sc_argc, char* sc_argv[])
81 {
82     timer Timer;
83     timer_configurator timerConfigurator;
84     sc_start();
85 }
```

Parameters in the timer module are declared as private members of the class in order to avoid direct access from any other modules. It ensures the use of the broker to access them. Each parameter is initialized in their respective constructors with their name and their default value. However, in order to help other modules to configure parameters, descriptions but also metadata have been added. In this example, metadata defines the unit and the valid range of parameters. Metadata are visual information and not a format checking mechanism. That's why, a callback mechanism has been added to the duty cycle parameter in order to validate the value. This check is done in a "pre write" callback. It enables to reject writes in case of an invalid value.

In the configurator module, parameters are requested from the broker handle by their name. If their validity is checked, then the parameter does not exist. The frequency is set through a typed API converting an untyped parameter handle to a typed parameter handle as we suppose the parameter type is know. The configuration of the duty cycle is similar, except it uses the untyped API. The first write is validated from the pre write callback in the timer module as in the range. However, the second update is rejected as out of the range. It means that the value of the duty cycle does not change.

This example is a more concrete introduction to the CCI standard but there is a lot more to explore depending of use cases : private parameters, read only, locking... More details are available in the LRM [8].

A.1.3 More about CCI callbacks

Callbacks can be typed or untyped like the parameters as showed in Listing 2.6 in Chapter 2. Initially, the callback mechanism does not support lambdas and it was a missing piece as seen in [163]. Indeed, Lambdas enables the definition of behaviour in a code block without the specification of an explicit function that is never used outside of the callback context.

The initial version of callbacks in the precursor of CCI is based on shared pointers and can lead to memory leaks. It increased the number of dependencies due to C++03 barriers. The new version has been introduced by Philipp A. Hartmann. The general idea is to hide the specifics of the CCI callbacks and especially the duality of the parameter callback signatures (a `cci_value` vs a strongly typed one) behind a common set of classes. The typed signature recovery is then done inside the parameter implementation again. An untyped callback handle can be converted back to a typed handle again. However, tag parameters are necessary to disambiguate the callback parameters, which can be built from arbitrary callable types to support C++ lambdas transparently. Indeed, the compiler cannot properly distinguish which type to try instantiating.

CCI introduced a callback mechanism used to track updates (write) or introspection of parameters (read). The callback mechanism has been elaborated in order to support lambda mechanisms that will be supported next by SystemC itself by adding the support of C++11. The architecture has been thought to be generic and has then been tailored to work with CCI parameters. However, this mechanism can also be used for other requirements inside the TLM standard itself. A move of this mechanism to the SystemC standard would be helpful in the future.

A.1.4 The CCI standardization process

The delay of the CCI standard has (inadvertently) changed the very nature of the standard that is required. The working group is shooting at a moving objective, and it is moving as a result of the working group itself. The typical process for an Accellera working group is to capture requirements, discuss, propose code, and then refine.

For CCI, the single biggest reason that development stalled (for almost 3 years) was that funding stopped. The 3rd party actors were no longer engaged in the processes, and without them, code refinement effectively stopped. Once funding was re-established, the actual standard code was finalized within months.

Accellera has also recently changed it is licensing policy. It now supports the open Apache II

license. This may well encourage more open development. In the end, this may even radically alter the approach to standardization as potentially proposed coded solutions may become available first, prior to a 'formal' requirements capture. It can greatly increase the speed and re-activeness of the standardization mechanisms. But this is still to be seen in practice.

A.1.5 CCI parameter lifetime: destruction and resurrection

Although SystemC modules cannot be destroyed during the simulation, SystemC processes can be dynamically created. It means that parameters can be created and used only during a certain amount of (simulation) time between two events such as the send and the receive of a TLM transaction. It implies that parameters can be destroyed at the end of this ephemeral process. Effectively parameter can be created and destroyed dynamically. Hence, parameter handles can become orphaned and must be able to detect a parameter destruction. A typical use case is a module state reset like a flash in a SoC. If the flash memory is reset, parameters that define characteristics or states can be destroyed and then resurrected with the newly provided initial values.

Unfortunately, the callback mechanism would be inefficient here. Hence parameter handle creation and destruction are not handled by callbacks. The issue is to notify parameter handle when the original parameter is destroyed to not enable them to access anymore to the pointer of the original parameter in order to avoid segmentation fault. It also implies the contrary. The original parameter have to be notified that parameter handle has been destroyed to avoid to notify it. Giving a kind of notifications to designers do not add any value and added some complexity. Indeed, the required notifications are only related to the link between the parameters and the handles. Instead, a behind-the-scenes dedicated mechanism has been implemented in order to ensure the validity of parameter handles when original parameters are destroyed and the reverse. It has been implemented in the parameter handle. It is part of the standard and cannot be redefined by vendor and in the default implementation. It means that any custom implementation has to meet the requirements as explained in the LRM [8].

The mechanism ensures that parameter handles become "invalid" (and modules using a handle should, as per normal, check their validity before using them). On the other hand, in case the same SystemC process is created again, the parameter will be resurrected (based on the name matching). Parameter handles are automatically "linked" again to the parameter. Some security has been added in order to deny the use of parameter handle while the original is not more existing and has not been resurrected yet.

A.1.6 Broker details

In general, the CCI standard tries to avoid passing parameter pointers in order to improve the tracking of parameter updates or reads. In order to solve this issue, our first solution was a broker stack. The principle was to track the existence of brokers in the SystemC hierarchy in order to automatically provide the right broker to each module. However, the stack mechanism was broken. While the stacking worked going down to the hierarchy, there was no clean way of "unstacking". A more robust solution has been considered with a simple registry. The main idea is to force modules to use a private broker to register first the private broker in the registry in order to be associated with their place in the hierarchy. This is coupled with a mechanism for each module to search for the appropriate broker in the registry based on the module SystemC hierarchical name.

It simplified the global configuration API by increasing the uniformity of the way in which modules find their broker and use parameters. In order to support different private broker implementations, the registry is based on the broker interface and so enables interoperability.

The precursor of CCI introduced a notion of public and private global parameter API, now called the broker. However, the proposed mechanism was not well integrated into a uniform way to use the broker in SystemC modules. Indeed, it was required to pass the instance of the private broker to child modules in their constructor through a pointer or a reference.

As discussed in the section A.1.5, the dynamic parameter destruction and resurrection during the simulation is close to the broker. Indeed, when a parameter is destroyed, all parameter handles are notified. However, when a parameter handle is used but the original parameter was previously destroyed, it will try to attach again with the broker before to process the request. In that case, it will use the current broker associated to the module. Only one broker per module simplifies this mechanism and the broker hierarchy.

While it is clear that only one broker should be used by a module at once for consistency, the mix of private and public parameters are not clearly defined. Initially, it is possible to use multiple brokers in the same module. It makes complex the API's and breaks the uniformity of parameter access.

A.2 TLM-2.0 improvements

A.2.1 OSI

From bottom to top, the first OSI layer, named the physical layer, processes communication at the digital bit level as raw data on a communication channel. This layer has to define physical characteristics like voltage levels and timing. Physical properties can be modelled with SystemC-AMS [4]. The second OSI layer, the data link layer, acts as a binder. This layer describes the transformation of the physical atomic information, called bits, into sequences of bits, called frames. This layer has to detect frame borders and manage errors during the transmission. This is the layer at which data congestion and buffering is handled. The unit of information of this OSI layer is the frame. However, this layer does not take data routing feature into account when multiple devices are interconnected. The third OSI layer is the network layer. This layer manages routing to support networks and sub networks. It also manages resource conflicts and interconnect between networks. The unit of the information to this OSI layer is the packet. This layer is used for “One to Many” or “Many to Many” communications. Higher layers like the transport layer define flow control of data and systems for higher level error checking and recovering. Indeed, they are frequently considered already part of the software domain.

The OSI model defines a packet as a bank of data that contains control information and the payload. The control part contains information to deliver the payload like the address (if applicable) to destination, error and sequences. The packet provides enough information to route the data between multiple nodes for “One to Many” or “Many to Many” communications. A comparison of communication modeling and the OSI layers has been done in [71].

A.2.2 Protocols

A.2.2.1 I2C

The I2C bus can be busy while another master try to send a transaction. In that case, there is an arbitration mechanism. It allows multiple master to handle the bus at the same time without data corruption. Arbitration starts as soon as two or more master place information on the bus at the same time. It stops (arbitration is lost) for the master that intends to send a logical one while the other master sends a logical zero. As soon as arbitration is lost by a master, it stops to send data. It also follows the bus in order to detect a stop. When the stop is detected, the master which has lost arbitration may put its data on the bus by respecting arbitration. Arbitration is managed by each master device. It waits that the bus is free to send the data (queue) and the arbitration can be linked to an interrupt as soon as it got detected. A time-out mechanism is provided to prevent strange behaviour that happens on the SCL line and so to reset the transaction. This protocol can also be software emulated using a GPIO controller for example.

A.2.2.2 CAN

The CAN bus is based on two speed modes (High speed and Low speed) with a bit rate up to 1Mbit/s. They happen on differential lines (called CANL and CANH). In order to receive the data from the bus, a CAN transceiver is required. The data is broadcast to all nodes. That’s why most

CAN controllers include a notion of message queue, also called mailboxes. It enables to filter the messages according to an identifier available in the frame. The baud-rate can be auto detected and there are four categories of frames: data, remote, error and over loader. All controllers on a CAN bus have the same bit rate and bit length. There are some error detection mechanisms based on the CRC in the frame but not only. The frame is described in Figure 3.16 in Chapter 3. The ID length is smaller in case of a non extended CAN frame.

A.2.3 Improved TLM Quantum Keeper

An implementation of the improved quantum keeper, also called QKP is presented in Listing 5.

Listing 5 – Improved TLM Quantum Keeper

```
1  template<typename MODULE> class QuantumKeeperPlus : public
     tlm_quantumkeeper {
2  public:
3      typedef void (MODULE::* NotifyPtr)(sc_core::sc_time&);
4      QuantumKeeperPlus(): tlm_quantumkeeper(), m_notify_ptr(0), m_mod(0)
         , m_notify_time(0, sc_core::SC_NS) { }
5      virtual void sync() {
6          if(m_notify_time >= get_global_quantum())
7              m_notify_time -= m_local_time;
8          tlm_quantumkeeper::sync();
9          if(m_local_time >= m_notify_time)
10             notify();
11     }
12     virtual void inc(const sc_core::sc_time& t) {
13         tlm_quantumkeeper::inc(t);
14         if (m_notify_ptr)
15             if(m_local_time >= m_notify_time)
16                 notify();
17     }
18     virtual void set(const sc_core::sc_time& t) {
19         tlm_quantumkeeper::set(t);
20         if (m_notify_ptr)
21             if(m_local_time >= m_notify_time)
22                 notify();
23     }
24     void notify() {
25         assert(m_mod);
26         NotifyPtr notify_ptr_copy = m_notify_ptr;
27         m_notify_ptr = 0;
28         (m_mod->*notify_ptr_copy)(m_local_time);
29     }
30     void register_notify(MODULE* mod, void (MODULE::*cb)(sc_core::
         sc_time&), const sc_core::sc_time& time) {
31         m_notify_time = time;
32         assert(!m_mod || m_mod == mod);
33         m_mod = mod;
34         m_notify_ptr = cb;
35     }
36     MODULE* m_mod;
37     NotifyPtr m_notify_ptr;
38     sc_core::sc_time m_notify_time;
```

```
39  };
```

A.2.4 Improved TLM-2.0 blue print

The Accellera OCP SLD kit is a full TLM kit that is available for the community. The kit is based on TLM-2.0's generic protocol. It contains a set of extended phases and payload extensions. OCP is a large family of protocols with many options. The kit supports the automatic negotiation of which options will be used in communication. In addition, the kit provides transactors between the physical layer (OSI layer 1) and AT or LT levels of the protocol. Thus, the kit can be directly applied in mixed simulations. The kit also contains protocol checkers and analyzers, examples and a documentation.

By taking the OCP kit as the basis of what should be expected in a TLM interface kit, the following is suggested:

- OSI analysis : An analysis of the protocol in terms of it is OSI layer 2/3 characteristics. This should directly guide the choice of payload and phases. It should form part of the documentation of the kit. This is absent from the OCP kit.
- Protocol : This is the only mandatory part of the interface definition as it defines the API. The protocol phases and payload must be defined. Specifically it means for a payload to define all the fields of the transactions and associated setters/getters. An enumeration of TLM response status has to be defined and passed as a template parameter to `tlm_base_payload`. A phase class specific to the protocol also need to be defined (except if the default phases are ok). It should specify an enumeration of phases. Again, where possible the phases should be defined referencing the phases defined by the TLM WG.
- Convenience sockets : Protocol kits should provide convenience sockets to cover common use cases.
- Transactors : In order to support software protocol emulation, RTL or even real hardware, a transactor down to OSI layer one (physical) should be provided. It becomes especially important for interfaces such as SPI that are commonly implemented using GPIO where it is necessary to abstract up from the lower level.
- Host Bridges : In addition to the connection to RTL or real hardware, for many interfaces, it also makes sense to provide connectors to host interfaces. In other words, an Ethernet interface kit might provide a way to connect to the host Ethernet.
- UVM (Universal Verification Methodology) : UVM-SystemC [164] is currently under public review. Interface kits should provide transactors to work with this standard.
- Routers : For routed and broadcast protocols, a router provides a convenient way to model the transport medium, and should be provided in the kit as it is a “generic” part for the protocol.
- Documentation and Examples. Note that the documentation should inherit mostly from the TLM documentation itself. To the degree that phases and payload fields are used from the pallet of existing items, their documentation can also simply be referenced.
- Legal : It is perhaps the most important feature. The interface kit should make it clear under what license the kit is made available. IP that uses this interface kit is become a derivative work, hence the license is exceedingly important.

A.2.5 Software emulated protocol

A.2.5.1 Introduction

Non Memory Mapped Protocols (NMMP) available in SoCs are commonly implemented inside hardware IP. A huge number of NMMPs IP directly impacts the physical size of the chip. That is why, due to physical constraint, the number of NMMPs available is physically limited. In the case of Figure 3.2, the number of UART is two, as well as SPI or I2C. This SoC is an ASSP. It is used by many companies with different families of applications. According to application constraints, NMMP protocols can be enough. However, if physical NMMPs do not fill application requirements, a bigger SoC can be envisaged. However, it is not always possible according, for example, to money constraints. In that case, others solutions have to be found.

One option to this issue is the ability to implement required NMMPs in software. This solution is commonly used to increase the flexibility of the chip. In that case, it is called "software protocol" or "software emulated protocol". The software drives pins, most of the time in parallel with a clock, in order to emulate the protocol as showed in Figure 3.28 in Chapter 3. This flexibility also enables to designer to implement customized versions of a protocol that has for example a different numbers of bits. In the end, software protocol cannot be ignored and should be supported in a virtual platform.

A.2.5.2 The definition of software protocol with TLM

Hardware peripherals that implement a communication protocol should provide a specific TLM socket. It means that a protocol can be statically detected in the virtual platform. Compatible protocol device(s) can so be bound to the peripheral. It is an ideal case where protocol sockets are available on both sides. However, if the device has to be connected to the GPIO(s), then few issues appear.

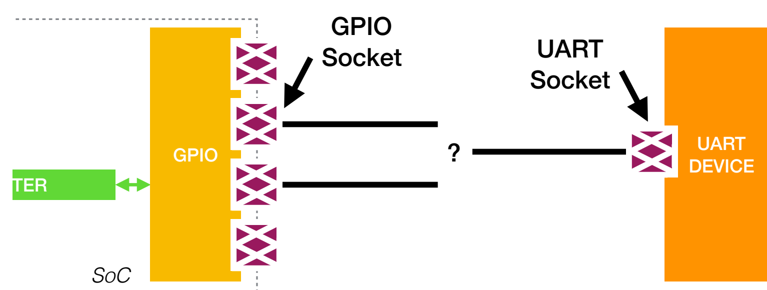


Figure 18 – TLM and software protocols

- First, if the device uses the TLM protocol socket it will not be able to connect it directly to the GPIO(s). Indeed, as showed in Figure 18 with an UART device, the socket cannot be directly connected to multiple GPIO ports.
- Second, TLM socket aims to abstract as most as possible wires and so un-required details of the transactions. The GPIO port, and so wires, are closer to the RTL level. As showed in Figure 3.12 and discussed in Section 3.4 of Chapter 3, the GPIOs transaction(s) cannot be abstracted as a whole. Indeed, for UART peripherals, it usually contains a buffer with the entire data. It enables to build an entire UART transaction and sent it as a whole. However, a GPIO is driven temporally for each bit from the software. From a GPIO controller

point of view, the transaction is totally blind and it is not be able to build a TLM transaction as the emulated protocol remains unknown. The protocol is entirely dynamic, software dependent, and can be totally undefined during the platform design. An automate discovery of the protocol used has been envisaged but the idea has quickly found its limits due to the numbers of existing protocols and the undefined usage from the software.

- Third, Figure 18 assumes that the ended device is directly connected to the GPIO device. However, a same GPIO controller can be used for multiple devices and different protocols at different times. GPIO pins can also be directly connected to another GPIO controller as shown in Figure 19. In this case, the used protocols cannot be easily detected from a hardware point of view, and so hardly abstracted.

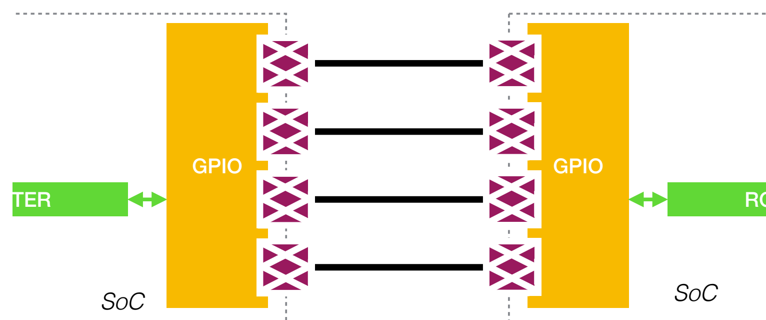


Figure 19 – Software protocol(s) connected to another software protocol(s)

Finally, the protocol negotiation based on CCI cannot be directly applied here. As the protocol is dynamically driven from the software, meta-datas cannot be set in the GPIO controller according to the emulated protocol. Instead, new mechanisms are required.

A.2.5.3 A first approach

A level adapter is required to ensure the communication between the TLM socket at AT or LT level and the GPIO controller at another level. The adapter is represented in Figure 18. It is called a transactor. It converts a RTL level frame to an entire TLM transaction and the reverse. Thus, due to time constraint, a SystemC time or an accurate TLM time is required to build or decode a frame.

PINs can require a combination in order to build a more abstracted transaction at the TLM LT or AT level. Each PIN drives the line from 0 to 1. All can be synchronized from the software if multiple PINs need to be driven at the same time. In that case, the required degree of accuracy increases. Indeed, a translation from the RTL level to TLM AT or LT level has to happen. TLM AT cannot be used here to improve the accuracy because the limits of the entire transaction are unknown. A combination of TLM does not offer the accuracy required to ensure the right decoding. Instead, plain SystemC data types are required.

Finally, the retained solution is a transactor as showed in Figure 18 combined with plain SystemC data types. The GPIO controller exports frame from a SystemC interface like `sc_inout`. It implies that if the controller is not used to drive a software protocol, it reduces the performance of the platform contrary to a TLM transaction.

A.2.5.4 Limitations

Software emulated protocols have some limitations. Depending of the frequency of the CPU and the GPIO controller, the trigger speed can be limited. This is a hardware limitation that should be reflected on a virtual platform. With the current approach, this temporal aspect is not supported. As the transceiver encores/decodes transaction, it does not care about the GPIO capabilities.

From a hardware point of view, the usage of GPIO for a certain family of protocol is completely blind. GPIO controller cannot abstract itself the transaction and so directly build the TLM transaction without sending it bit after bit. Unfortunately, during transactions bit by bit, the number of interactions with the SystemC kernel increases. In the end, the usage of a software protocol reduces the simulation speed.

Finally, according to the nature of the information, it is possible to conclude that it will be hard to do better than the usage of a transceiver. The software dynamically changes the behaviour of the hardware. The GPIO controller can be customized if its usage is clearly defined at the head of the design cycle but it decreases the portability of a model.

A.2.5.5 Conclusion

The software emulated protocol represents a non negligible part of virtual platforms. Even if the emulated protocol cannot be directly handled, some mechanisms are required to support it. A solution has been presented based on transactors. However, it implies some drawbacks on the simulation performance. Moreover, as discussed in the limitation subsection, the difficulty of unknown protocol detection is really hard. So, this option has been totally ignored. At the end, there is no perfect solution and the degree of optimization is left to the designer in order to speed up the simulation.

A.3 Parallelism

A.3.1 Two SystemC kernels without time synchronization

The implementation of the "Add" and "AddExport" modules is presented in Listing 6 and 7.

Listing 6 – Add module

```
SC_MODULE(Add) {
    SC_CTOR(Add) {
        SC_THREAD(doJob);
        SC_THREAD(consumeTime);
    }

    sc_in<int> i;
    sc_out<int> o;

    void doJob() {
        int iValue = 1;
        while(1) {
            o.write(iValue);
            wait(i.value_changed_event());
            iValue = i.read();
            wait(50, SC_MS);
            o.write(iValue + 1);
        }
    }

    void consumeTime() {
        while(1) {
            wait(10, SC_MS);
            wait(40, SC_MS);
            wait(160, SC_MS);
            wait(30, SC_MS);
            wait(80, SC_MS);
        }
    }
};
```

Listing 7 – Add export module

```
class AddExport : public sc_core::sc_module, public AddExportIf {
    SC_HAS_PROCESS(AddExport);

public:
    AddExport(sc_module_name name):
            sc_module(name),
            AddExportIf() {
        SC_THREAD(receiveData);
        SC_THREAD(sendData);
    }

    sc_in<int> iExt;
```

```
13      sc_out<int> oExt;
14
15  private:
16      void receiveData() {
17          while(1) {
18              sc_core::sc_module::wait(ipc->receiveValueAsyncEvent());
19              oExt.write(ipc->getValue());
20          }
21      }
22
23      void sendData() {
24          while(1) {
25              sc_core::sc_module::wait(iExt.value_changed_event());
26              int iExtValue = iExt.read();
27              ipc->sendValue(iExtValue);
28          }
29      }
30  };
```

Bibliography

***

[1] Aboubacar Diarra. *OSI Layers in Automotive Networks*. 2013. URL: http://www.ieee802.org/1/files/public/docs2013/new-tsn-diarra-osi-layers-in-automotive-networks-0313-v01.pdf.

[2] Accellera. *About SystemC*. URL: http://www.eda.org/community/systemc/about-systemc.

[3] Accellera. "IEEE Draft Standard for Universal Verification Methodology Language Reference Manual". In: *IEEE Standard 1800.2-2017* (2017).

[4] Accellera. "IEEE Draft SystemC Analog/Mixed-Signal (AMS) extensions Language Reference Manual". In: *IEEE P1666.1/D4, October 2015* (Jan. 2015), pp. 1–233.

[5] Accellera. "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Standard 1666-2011* (2012), pp. 1–638.

[6] Accellera. "IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual". In: *IEEE Standard 1666.1-2016* (Apr. 2016), pp. 1–236.

[7] Accellera. *OCP Modelling Kit*. URL: http://accellera.org/downloads/standards/ocp.

[8] Accellera. *SystemC Configuration, Control & Inspection*. Accellera. 2017.

[9] M. Alassir, J. Denoulet, O. Romain, and P. Garda. "A SystemC AMS model of an I2C bus controller". In: *Proceedings of the International Conference on Design and Test of Integrated Systems in Nanoscale Technology (DTIS)*. Sept. 2006, pp. 154–158.

[10] Analytics Engines. *Xilinx Announce New Zynq Architecture*. 2015. URL: http://www.analyticsengines.com/developer-blog/xilinx-announce-new-zynq-architecture/.

[11] I. A. Aref, N. A. Ahmed, F. Rodriguez-Salazar, and K. Elgaid. "RTL-level modeling of an 8B/10B encoder-decoder using SystemC". In: *Proceedings of the International Conference on Wireless and Optical Communications Networks (WOCN)*. May 2008, pp. 1–4.

[12] ARM. *ARM Processors*. 2017. URL: https://www.arm.com/products/processors (visited on 09/15/2017).

[13] ARM. *Dhrystone Benchmarking for ARM Cortex Processors*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dai0273a/DAI0273A_dhrystone_benchmarking.pdf.

[14] ARM Limited. *Cycle Accurate Simulation Interface (CASI) Specification*. ARM Manual. 2010.

[15] S. Balaji and M. S. Murugaiyan. "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC". In: *International Journal of Information Technology and Business Management* (2012).

[16] N. Bannow, K. Haug, and W. Rosenstiel. "Automatic SystemC design configuration for a faster evaluation of different partitioning alternatives". In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Vol. 2. Mar. 2006, pp. 217–218.

[17] A. Barreteau. “System-Level Modeling and Simulation with Intel® CoFluent™ Studio”. In: *Proceedings of the International Conference on Complex Systems Design & Management (CSD&M)*. Ed. by G. Auvray, J.-C. Bocquet, E. Bonjour, and D. Krob. Cham: Springer International Publishing, 2016, pp. 305–306. URL: https://doi.org/10.1007/978-3-319-26109-6_32.

[18] D. Becker, M. Moy, and J. Cornet. “Challenges for the Parallelization of Loosely Timed SystemC Programs”. In: *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*. Amsterdam, Netherlands, Oct. 2015.

[19] D. Becker, M. Moy, and J. Cornet. “Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study”. In: *Electronics* 5.2 (May 2016), p. 22. URL: https://hal.archives-ouvertes.fr/hal-01321055.

[20] I. Bennour. “Petri nets framework for analyzing the communication behavior of TLM modules”. In: *Proceedings of the International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. May 2012, pp. 1–4.

[21] I. E. Bennour. “SystemC TLM2-protocol consistency checker using Petri net”. In: *Proceedings of the International Design Test Symposium (IDT)*. Dec. 2016, pp. 193–198.

[22] G. S. Beserra, S. H. A. Niaki, and I. Sander. “Integrating virtual platforms into a heterogeneous MoC-based modeling framework”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Sept. 2012, pp. 143–150.

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. “The gem5 Simulator”. In: *Special Interest Group on Computer Architecture (SIGARCH) Computer Architecture News* 39.2 (Aug. 2011), pp. 1–7. URL: http://doi.acm.org/10.1145/2024716.2024718.

[24] E. Blokken, J. Vounckx, and M. Eyckmans. *System Design Methodologies for System on Chip and Embedded Systems*. https://www.design-reuse.com/articles/6850/system-design-methodologies-for-system-on-chip-and-embedded-systems.html.

[25] N. Bombieri, F. Fummi, and V. Guarnieri. “Automatic synthesis of OSCI TLM-2.0 models into RTL bus-based IPs”. In: *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*. June 2010, pp. 105–112.

[26] N. Bombieri, F. Fummi, and V. Guarnieri. “Model checking on TLM-2.0 IPs through automatic TLM-to-RTL synthesis”. In: *Proceedings of the IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC)*. Sept. 2010, pp. 61–66.

[27] N. Bombieri, F. Fummi, V. Guarnieri, and G. Pravadelli. “Testbench Qualification of SystemC TLM Protocols through Mutation Analysis”. In: *IEEE Transactions on Computers* (May 2014), pp. 1248–1261.

[28] J. Borland, P. Dawkins, D. Johnson, and R. Williams. *Embedded Systems Market (Embedded Hardware and Embedded Software) Market For Healthcare, Industrial, Automotive, Telecommunication, Consumer Electronics, Defense, Aerospace and Others Applications: Global Industry Perspective, Comprehensive Analysis and Forecast, 2015 - 2021*. Zion Market Research, 2016.

[29] F. Brandner. “Precise simulation of interrupts using a rollback mechanism”. In: *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. Nice, France, Apr. 2009, pp. 71–80.

[30] D. Brier, R. Venkatasubramanian, S. Rangarajan, A. Arun, D. Thompson, and N. Muralidharan. “Verification Methodology of Heterogeneous DSP+ARM Multicore Processors for Multi-core System on Chip”. In: *Proceedings of the International Workshop on Microprocessor Test and Verification (MTV)*. Dec. 2013, pp. 112–117.

[31] J. S. Brothers. “Integrated circuit development”. In: *Radio and Electronic Engineer* 43.1.2 (Jan. 1973), pp. 39–48.

[32] E. R. Brown and K. Baker. “Integrated Circuits for Thermal Imaging Applications”. In: *Proceedings of the First European Solid State Circuits Conference (ESSCIRC)*. Sept. 1975, pp. 48–49.

[33] Y.-H. Bu, Z.-Z. Tao, M.-J. Lei, C.-T. Wu, and C.-F. Wu. “A configurable SystemC virtual platform for early software development and its sub-system for hardware verification”. In: *Proceedings of the International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. Apr. 2010, pp. 29–32.

[34] B. E. Buckingham. “New Developments in Integrated Circuits for Television and Other Consumer Systems”. In: *Proceedings of the First European Solid State Circuits Conference (ESSCIRC)*. Sept. 1975, pp. 26–27.

[35] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. “Transaction-level models for AMBA bus architecture using SystemC 2.0”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2003, pp. 26–31.

[36] G. Castilhos, E. Wachter, G. Madalozzo, A. Erichsen, T. Monteiro, and F. Moraes. “A framework for MPSoC generation and distributed applications evaluation”. In: *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*. Mar. 2014, pp. 408–411.

[37] CCI WG. *Issues for Configuration*. Tech. rep. Accellera, 2009.

[38] CCI WG. *SystemC CCI Configuration Requirements Specification*. Tech. rep. Accellera, 2009.

[39] L. Charest and P. Marquet. “Comparisons of different approaches of realizing IP block configuration in SystemC”. In: *Proceedings of the IEEE International New Circuits and Systems Conference (NEWCAS)*. June 2005, pp. 83–86.

[40] Z. Chen, Y. Wang, L. Liao, Y. Zhang, A. Aytac, J. H. Müller, R. Wunderlich, and S. Heinen. “A SystemC Virtual Prototyping based methodology for multi-standard SoC functional verification”. In: *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2014, pp. 1–6.

[41] S. Y. Chien, W. K. Chan, Y. H. Tseng, C. H. Lee, V. S. Somayazulu, and Y. K. Chen. “Distributed computing in IoT: System-on-a-chip for smart cameras as an example”. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. Jan. 2015, pp. 130–135.

[42] J. E. Coffland and A. D. Pimentel. “A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems”. In: *Proceedings of the ACM Symposium On Applied Computing (SIGAPP)*. SAC ’03. Melbourne, Florida: ACM, Mar. 2003, pp. 666–671. URL: http://doi.acm.org/10.1145/952532.952663.

[43] N. Concer, L. Bononi, M. Soulie, R. Locatelli, and L. P. Carloni. “CTC: An end-to-end flow control protocol for multi-core systems-on-chip”. In: *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*. May 2009, pp. 193–202.

[44] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. Gürkaynak, M. Muehlberghuber, M. Gautschi, I. Loi, G. Haugou, S. Mangard, and L. Benini. "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2017), pp. 1–14.

[45] CoWare. *SystemC Modeling Library (SCML)*. CoWare Manual.

[46] David C. Black. *SystemC from the ground up*. URL: https://fr.slideshare.net/tuanhuynh16906/system-c-from-the-ground-up-david-c-black.

[47] D. Desmet, D. Verkest, and H. De Man. "Operating System Based Software Generation for Systems-on-chip". In: *Proceedings of the Design Automation Conference (DAC)*. DAC '00. Los Angeles, California, USA: ACM, June 2000, pp. 396–401.

[48] D. V. D. R. Devi, P. K. Kondugari, G. Basavaraju, and S. L. Gangadharaiah. "Efficient implementation of memory controllers and memories and virtual platform". In: *Proceedings of the International Conference on Communication and Signal Processing (ICCSP)*. Apr. 2014, pp. 1645–1648.

[49] N. N. Dlamini and K. Johnston. "The use, benefits and challenges of using the Internet of Things (IoT) in retail businesses: A literature review". In: *2016 International Conference on Advances in Computing and Communication Engineering (ICACCE)*. Nov. 2016, pp. 430–436.

[50] R. Dömer, W. Chen, and X. Han. "Parallel discrete event simulation of Transaction Level Models". In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. June 2012, pp. 227–231.

[51] R. Dömer, A. Gerstlauer, P. Kritzinger, and M. Olivarez. "The SpecC System-Level Design Language and Methodology". In: *Proceedings of the Embedded Systems Conference (ESC)*. Apr. 2002.

[52] W. Du, F. Mieyeville, and D. Navarro. "Modeling Energy Consumption of Wireless Sensor Networks by SystemC". In: *Proceedings of the International Conference on Systems and Networks Communications (ICSNC)*. Oct. 2010, pp. 94–98.

[53] M. Edwards and P. Green. "UML for Hardware and Software Object Modeling". In: *UML for Real: Design of Embedded Real-Time Systems*. Ed. by L. Lavagno, G. Martin, and B. Selic. Boston, MA: Springer US, 2003, pp. 127–147. URL: http://dx.doi.org/10.1007/0-306-48738-1_6.

[54] Ericsson. *50 billion connections 2020*. 2010. URL: https://www.ericsson.com/en/press-releases/2010/4/ceo-to-shareholders-50-billion-connections-2020.

[55] W. H. Evans, J. C. Ballegeer, and N. H. Duyet. "ADL: An Algorithmic Design Language for Integrated Circuit Synthesis". In: *Proceedings of the Design Automation Conference (DAC)*. June 1984, pp. 66–72.

[56] T. M. Frederiksen. "Limits of integrated circuits". In: *Proceedings of the IEEE Power Electronics Specialists Conference (PESC)*. Apr. 1970, pp. 54–59.

[57] **G. Delbergue**. *SystemC Async Patch*. URL: https://git.greensocs.com/systemc/systemc/tree/await.

[58] **G. Delbergue**, M. Burton, F. Konrad, B. Le Gal, and C. Jego. "QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0". In: *Proceedings of the European Congress Embedded Real Time Software And Systems (ERTS)*. Toulouse, France, Jan. 2016.

[59] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “A SystemC asynchronous wait mechanism enabling multi-threading and multi-simulator support”. In: *Proceedings of the SystemC Evolution Day*. Munich, Germany, May 2016.

[60] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Analysis of TLM-2.0 and it’s Applicability to Non Memory Mapped Interfaces”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. San Jose, USA, Feb. 2016.

[61] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Analysis of TLM-2.0 and it’s applicability to non memory mapped interfaces”. In: *Proceedings of the SystemC Evolution Day*. Munich, Germany, May 2016.

[62] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Beyond QBox: development of virtual platforms based on QEMU and SystemC TLM-2.0”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Barcelona, Spain, Sept. 2015.

[63] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Multi-threaded Virtual Platform Simulation: An open-source approach, using SystemC TLM-2.0, and QEMU”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Barcelona, Spain, Sept. 2015.

[64] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Open Source Heterogeneous AMP Virtual Platforms built using the SystemC and TLM standards”. In: *Proceedings of the Open Source Digital Design Conference (ORConf)*. Bologna, Italy, Oct. 2016.

[65] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Plateformes virtuelles SystemC/TLM : configuration, communication et parallélisation”. In: *Proceedings of the Groupement De Recherche System On Chip et System-In-Package (GDR SoC-SiP)*. Bordeaux, France, June 2017.

[66] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “The missing SystemC and TLM asynchronous features enabling inter-simulation synchronization”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Munich, Germany, Oct. 2016.

[67] **G. Delbergue**, M. Burton, B. Le Gal, and C. Jego. “Virtual platform(s) simulation: an open-source, reusable, affordable and structured approach based on TLM/CCI”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Munich, Germany, Nov. 2015.

[68] **G. Delbergue** and T. Wieman. “SystemC Configuration, A preview of the draft standard”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Nov. 2016.

[69] D. D. Gajski. “System-level synthesis: From specification to transaction level models”. In: *Proceedings of the International Conference on Communications, Circuits and Systems (ICCCAS)*. July 2009, pp. 1134–1138.

[70] Geekbar. *iPhone 7 Teardown*. 2016. URL: https://www.laptopmain.com/apple-iphone-7-teardown/.

[71] A. Gerstlauer, D. Shin, R. Domer, and D. D. Gajski. “System-level communication modeling for network-on-chip synthesis”. In: *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. Vol. 1. Jan. 2005, pp. 45–48.

[72] G. Glaser, G. Nitschey, and E. Hennig. “Temporal decoupling with error-bounded predictive quantum control”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Sept. 2015, pp. 1–6.

[73] W. B. Glendinning. “Silicon Integrated Circuits”. In: *IRE Transactions on Military Electronics* MIL-4.4 (Oct. 1960), pp. 459–468.

[74] R. Goldstein, S. Breslav, and A. Khan. “A quantum of continuous simulated time”. In: *Proceedings of the Symposium on Theory of Modeling and Simulation (TMS-DEVS)*. Apr. 2016, pp. 1–8.

[75] GreenSocs. *GreenLib*. URL: http://git.greensocs.com/greenlib/greenlib.

[76] GreenSocs. *QEMU-SC*. URL: http://git.greensocs.com/qemu/qemu-sc.

[77] P. E. Haggerty, C. L. Hogan, R. N. Noyce, L. C. Maier, J. E. Brown, and C. H. Knowles. “Integrated circuits”. In: *IEEE Spectrum* 1.6 (June 1964), pp. 62–62.

[78] T. Hattori. “SOC design challenges for embedded systems”. In: *Proceedings of the International Conference on ASIC (ASICON)*. Oct. 2007, pp. 15–19.

[79] C. Helmstetter, J. Cornet, B. Galilée, M. Moy, and P. Vivet. “Fast and accurate TLM simulations using temporal decoupling for FIFO-based communications”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2013, pp. 1185–1188.

[80] C. Helmstetter and V. Joloboff. “SimSoC: A SystemC TLM integrated ISS for full system simulation”. In: *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. Nov. 2008, pp. 1759–1762.

[81] R. Hocine, H. Kalla, S. Kalla, and C. Arar. “A methodology for verification of embedded systems based on SystemC”. In: *Proceedings of the IEEE International Conference on Complex Systems (ICCS)*. Nov. 2012, pp. 1–6.

[82] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. “Scalably distributed SystemC simulation for embedded applications”. In: *Proceedings of the International Symposium on Industrial Embedded Systems (SIES)*. June 2008, pp. 271–274.

[83] IBS. *SoC cost evolution*. 2014. URL: http://semiengineering.com/how-much-will-that-chip-cost/.

[84] iFixit. *iPhone 3G Teardown*. 2009. URL: https://ifixit-guide-pdfs.s3.amazonaws.com/pdf/ifixit/guide_600_en.pdf.

[85] S. Inohira, T. Shinmi, M. Nagata, and K. Iida. “Statistical Modeling for Large Scale Integrated Circuit Design”. In: *Proceedings of the International Symposium on VLSI Technology*. Sept. 1982, pp. 76–77.

[86] Intel. *Intel Stratix 10*. URL: https://www.altera.com/products/fpga/stratix-series/stratix-10/features.html.

[87] S. International. “Architecture Analysis & Design Language (AADL)”. In: *AS5506C* (2017), pp. 1–355.

[88] ISO. *ISO/IEC 14882:2003: Programming languages: C++*. Ed. by ISO. ISO, 2003. URL: http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110.

[89] I. O. for Standardization. *Open System interconnection*. Standard. International Organization for Standardization, Nov. 1994.

[90] ITRS. *ITRS Reports 2013*. 2013. URL: http://www.itrs2.net/2013-itrs.html.

[91] P. Jones. “The making of an integrated circuit”. In: *Electronics and Power* 21.21.22 (Dec. 1975), pp. 1179–1182.

[92] S. Kapur and C. Sriprasad. “Software development tools for embedded systems”. In: *Proceedings of the IEEE/AIAA Digital Avionics Systems Conference (DASC)*. Nov. 1995, pp. 331–335.

[93] D. Keppel. *Tools and Techniques for Building Fast Portable Threads Packages*. Tech. rep. UWCSE 93-05-06. University of Washington Department of Computer Science and Engineering, May 1993.

[94] R. S. Khaligh and M. Radetzki. “A dynamic load balancing method for parallel simulation of accuracy adaptive TLMs”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Sept. 2010, pp. 1–6.

[95] R. S. Khaligh and M. Radetzki. “Modeling constructs and kernel for parallel simulation of accuracy adaptive TLMs”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2010, pp. 1183–1188.

[96] W. Klingauf, R. Gunzel, O. Bringmann, P. Parfuntseu, and M. Burton. “GreenBus - a generic interconnect fabric for transaction level modelling”. In: *Proceedings of the Design Automation Conference (DAC)*. July 2006, pp. 905–910.

[97] U. Ko. “Ultra-low power SoC for wearable IoT”. In: *Proceedings of the International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*. Apr. 2016, p. 1.

[98] T. Kogel. “Using the new TLM-2.0 Standard for the Creation of Virtual Platforms for ESL Design”. In: *Proceedings of the Multicore and multiprocessor SoC Forum (MPSoC)*. June 2008.

[99] V. Lapotre. “Toward dynamically reconfigurable high throughput multiprocessor Turbo decoder in a multimode and multi-standard context”. Theses. Université de Bretagne-Sud, Nov. 2013. URL: https://hal.archives-ouvertes.fr/tel-01096975.

[100] J. Lee. “Design methodology for on-chip bus architectures using system-on-chip network protocol”. In: *IET Circuits, Devices Systems* 6.2 (Mar. 2012), pp. 85–94.

[101] T. Lei, Y. Yanhui, and W. Shaojun. “Optimizing SoC platform architecture for multimedia applications”. In: *Proceedings of the International Conference on ASIC (ASICON)*. Vol. 1. Oct. 2005, pp. 94–97.

[102] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel, and M. Vaupel. “Virtual platforms: Breaking new grounds”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2012, pp. 685–690.

[103] O. Levia. “Level synthesis approach to application-specific integrated circuits (ASIC) design”. In: *Proceedings of the IEEE ASIC Seminar and Exhibit*. Sept. 1989, pp. 1–7.

[104] S. Liao, S. Tjiang, and R. Gupta. “An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment”. In: *Proceedings of the Design Automation Conference (DAC)*. June 1997, pp. 70–75.

[105] LIP6. *SystemCASS*. URL: https://www.almos.fr/trac/systemcass.

[106] Lip6. *SoCLib*. URL: http://www.soclib.fr.

[107] Lua Community. *The Programming Language*. URL: https://www.lua.org.

[108] T. M. Madzy. “A Mathematical Model to Predict the Susceptibility of Integrated Circuits to Magnetic Fields”. In: *Proceedings of the IEEE International Electromagnetic Compatibility Symposium Record*. July 1971, pp. 1–6.

[109] E. Maler, J. Paoli, M. Sperberg-McQueen, F. Yergeau, and T. Bray. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. http://www.w3.org/TR/2008/REC-xml-20081126/. W3C, Nov. 2008.

[110] S. Mathur and S. Malik. “Advancements in the V-Model”. In: *International Journal of Computer Applications* 1.12 (2010), pp. 29–34.

[111] MATLAB. *version 7.10.0 (R2016a)*. Natick, Massachusetts: The MathWorks Inc., 2016.

[112] Matthias Jung. *Coulpling GEM5 with IEEE1666 SystemC TLM2.0*. Tech. rep. Microelectronic Systems Design Research Group, 2015.

[113] Matthieu Moy. *Modélisation TLM en SystemC*. 2017. URL: https://github.com/moy/cours-tlm.

[114] D. McCrory. *Heterogeneous symmetric multi-processing system*. US Patent 6,513,057. Jan. 2003. URL: https://www.google.com/patents/US6513057.

[115] Mediatek. *MediaTek Helio X30*. 2017. URL: https://www.mediatek.com/products/smartphones/mediatek-helio-x30.

[116] A. Mello, I. Maia, A. Greiner, and F. Pecheux. "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations". In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2010, pp. 606–609.

[117] P. Menniti and B. Murari. "An Integrated-Circuit Sound for Television Receivers". In: *IEEE Transactions on Consumer Electronics* CE-21.1 (Feb. 1975), pp. 74–84.

[118] R. Meyer, J. Wagner, B. Farkas, S. Horsinka, P. Siegl, R. Buchty, and M. Berekovic. "A Scriptable Standard-Compliant Reporting and Logging Framework for SystemC". In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.1 (Oct. 2016), pp. 6–28.

[119] Microsemi. *UART-to-SPI Interface - Application Note AC327*. 2012.

[120] *Modbus Protocol*. PI–MBUS–300. MODBUS. May 1996.

[121] M. G. ModelSim. *2014-15*. Mentor Graphics, 2014.

[122] J. L. Moll. "Integrated Circuits and Microminiaturization". In: *IRE Transactions on Education* 3.4 (Dec. 1960), pp. 141–144.

[123] M. Monton, J. Engblom, and M. Burton. "Checkpointing for Virtual Platforms and SystemC-TLM". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.1 (2013), pp. 133–141.

[124] J. Moreira, F. Klein, A. Baldassin, P. Centoducatte, R. Azevedo, and S. Rigo. "Using multiple abstraction levels to speedup an MPSoC virtual platform simulator". In: *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*. May 2011, pp. 99–105.

[125] M. A. El-Moursy, A. Sheirah, M. Safar, and A. Salem. "Efficient embedded SoC hardware/-software codesign using virtual platform". In: *Proceedings of the International Design and Test Symposium (IDT)*. Dec. 2014, pp. 36–38.

[126] I. Moussa, T. Grellier, and G. Nguyen. "Exploring SW performance using SoC transaction-level modeling". In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2003, pp. 120–125.

[127] M. Moy. "Parallel programming with SystemC for loosely timed models: A non-intrusive approach". In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2013, pp. 9–14.

[128] Mr. A. B. Shinde. *System on Chip*. 2015. URL: https://www.slideshare.net/abshinde/system-on-chip-43988176.

[129] F. Mueller. "Pthreads library interface". In: *Florida State University* (1993).

[130] M. A. Murray-Lasso. "Black-Box Models for Linear Integrated Circuits". In: *IEEE Transactions on Education* 12.3 (Sept. 1969), pp. 170–180.

[131] NXP. *i.MX 8 Family – ARM® Cortex®-A53, Cortex-A72, Virtualization, Vision, 3D Graphics, 4K Video*. 2016. URL: https://www.nxp.com/products/microcontrollers-and-processors/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-8-processors/i.mx-8-family-arm-cortex-a53-cortex-a72-virtualization-vision-3d-graphics-4k-video:i.MX8/.

[132] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. Object Management Group, 2012. URL: http://www.omg.org/spec/SysML/1.3/.

[133] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1.

[134] S. Ornes. "Core Concept: The Internet of Things and the explosion of interconnectivity". In: *Proceedings of the National Academy of Sciences* 113.40 (2016), pp. 11059–11060.

[135] OSCI. *OSCI TLM-2.0 The Transaction Level Modeling standard of the Open SystemC Initiative (OSCI)*. 2009. URL: http://slideplayer.com/slide/4588233/.

[136] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi. "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines". In: *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*. June 2009, pp. 80–87.

[137] E. Painkras, L. A. Plana, J. Garside, S. Temple, S. Davidson, J. Pepper, D. Clark, C. Patterson, and S. Furber. "SpiNNaker: A multi-core System-on-Chip for massively-parallel neural net simulation". In: *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*. Sept. 2012, pp. 1–4.

[138] S. Pasricha, N. Dutt, and M. Ben-Romdhane. "Using TLM for exploring bus-based SoC communication architectures". In: *Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. July 2005, pp. 79–85.

[139] J. Peeters, N. Ventroux, T. Sassolas, and L. Lacassagne. "A SystemC TLM framework for distributed simulation of complex systems with unpredictable communication". In: *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Oct. 2011, pp. 1–8.

[140] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen. "A Survey on Internet of Things From Industrial Market Perspective". In: *IEEE Access* 2 (2014), pp. 1660–1679.

[141] I. M. Pessoa, A. Mello, A. Greiner, and F. Pêcheux. "Parallel TLM simulation of MPSoC on SMP workstations: Influence of communication locality". In: *Proceedings of the International Conference on Microelectronics (ICM)*. Dec. 2010, pp. 359–362.

[142] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini. "A Heterogeneous Multi-Core System-on-Chip for Energy Efficient Brain Inspired Computing". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2017).

[143] *QEMU - Quick EMUlator*. URL: http://www.qemu.org.

[144] M. Radetzki. "SystemC TLM Transaction Modelling and Dispatch for Active Object." In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Sept. 2006, pp. 203–209.

[145] K. Rahmouni, S. Chabanet, N. Lambelin, and F. Pétrot. "Design of a medium voltage protection device using system simulation approaches: a case study". In: *International Journal of Engineering Science (IJES)* 5 (2013), pp. 53–66.

[146] R. Rajsuman. *System-on-a-Chip: Design and Test*. 1st. Norwood, MA, USA: Artech House, Inc., 2000.

[147] R. Ranjith, J. Mathew, and J. K. Murthy. "Host Testing of Drivers Using SystemC Model". In: *International Journal of Electrical Electronics & Computer Science Engineering* (2016).

[148] RapidJSON. *RapidJSON*. URL: http://rapidjson.org.

[149] D. Reed and R. Hoare. "An SoC solution for massive parallel processing". In: *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)*. Apr. 2002, p. 8.

[150] P. Reichel and J. Doge. “Hardware/software infrastructure for ASIC commissioning and rapid system prototyping”. In: *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Aug. 2014, pp. 1–6.

[151] V. Reyes. “TLM Technology for Off-Chip Interfaces on the Automotive domain”. In: *Proceedings of the European SystemC User's Group Events Workshop (ESCUG)*. Sept. 2012.

[152] K. Richter, R. Racu, and R. Ernst. “Scheduling analysis integration for heterogeneous multiprocessor SoC”. In: *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*. Dec. 2003, pp. 236–245.

[153] C. Roth, S. Reder, G. Erdogan, O. Sander, G. M. Almeida, H. Bucher, and J. Becker. “Asynchronous parallel MPSoC simulation on the Single-Chip Cloud Computer”. In: *Proceedings of the International Symposium on System-on-Chip*. Oct. 2012, pp. 1–8.

[154] E. S. S. Baindur and A. Patil. “Dynamic Parameter Configuration of SystemC Models”. In: *Proceedings of the Design and Verification Conference and Exhibition India (DVCon India)*. Sept. 2015.

[155] S. Salaheddine Hamza, I. E. BENNOUR, and R. TOURKI. “TLM Design Framework of Generic NoC for Performance Exploration”. In: *Proceedings of the International Conference on Computer Science and Engineering*. 2009.

[156] A. Sangiovanni-Vincentelli. “System-level design: a strategic investment for the future of the electronic industry”. In: *Proceedings of the International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. Apr. 2005, pp. 1–5.

[157] C. Sauer, H. M. Bluethgen, and H. P. Loeb. “Distributed, loosely-synchronized SystemC/TLM simulations of many-processor platforms”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Vol. 978-2-9530504-9-3. Oct. 2014, pp. 1–8.

[158] C. Sauer and H. P. Loeb. “A lightweight infrastructure for the dynamic creation and configuration of virtual platforms”. In: *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2015, pp. 372–377.

[159] R. R. Schaller. “Moore's law: past, present and future”. In: *IEEE Spectrum* 34.6 (June 1997), pp. 52–59.

[160] G. Schirner and R. Domer. “Result-Oriented Modeling – A Novel Technique for Fast and Accurate TLM”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.9 (Sept. 2007), pp. 1688–1699.

[161] G. Schirner and R. Dömer. “ABSTRACT COMMUNICATION MODELING”. In: *From Specification to Embedded Systems Application*. Ed. by A. Rettberg, M. C. Zanella, and F. J. Rammig. Boston, MA: Springer US, 2005, pp. 189–200. URL: http://dx.doi.org/10.1007/11523277_19.

[162] H. G. Schirner. “System Level Modeling of an AMBA Bus”. PhD thesis. UNIVERSITY OF CALIFORNIA, IRVINE, 2005.

[163] C. Schröder, W. Klingauf, R. Günzel, M. Burton, and E. Roesler. “Configuration and Control of SystemC Models Using TLM Middleware”. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. CODES+ISSS '09. Grenoble, France: ACM, Mar. 2009, pp. 81–88.

[164] S. Schulz, T. Vörtler, and M. Barnasconi. “UVM goes Universal - Introducing UVM in SystemC”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Nov. 2015.

[165] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann. “parSC: Synchronous parallel SystemC simulation on multi-core host architectures”. In: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Mar. 2010, pp. 241–246.

[166] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic. “SoCRocket - A virtual platform for the European Space Agency’s SoC development”. In: *Proceedings of the International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. May 2014, pp. 1–7.

[167] P. J. Schwarz. “An Integrated Circuit for the Telephone Handset”. In: *Proceedings of the First European Solid State Circuits Conference (ESSCIRC)*. Sept. 1975, pp. 91–92.

[168] A. Sedrati and A. Mezrioui. “Internet of Things challenges: A focus on security aspects”. In: *2017 8th International Conference on Information and Communication Systems (ICICS)*. Apr. 2017, pp. 210–215.

[169] Semiconductor Insights. *iPhone 3G Teardown*. 2008. URL: http://appleinsider.com/articles/08/07/12/every_iphone_3g_chip_named_illustrated_in_detail.

[170] C. Shin and Y. Kim. “Development of a virtual platform for IP and firmware verification”. In: *Proceedings of the International SoC Design Conference (ISOCC)*. Nov. 2014, pp. 282–283.

[171] L. Shuping and P. Ling. “The Research of V Model in Testing Embedded Software”. In: *Proceedings of the International Conference on Computer Science and Information Technology (ICCSIT)*. Aug. 2008, pp. 463–466.

[172] P. Sjövall, J. Virtanen, J. Vanne, and T. D. Hämäläinen. “High-Level Synthesis Design Flow for HEVC Intra Encoder on SoC-FPGA”. In: *Proceedings of the Euromicro Conference on Digital System Design (DSD)*. Apr. 2015, pp. 49–56.

[173] *SMART ARM-based MCU*. 11057C. ATMEL. Mar. 2015.

[174] I. Sobański and W. Sakowski. “Hardware/software co-design in USB 3.0 mass storage application”. In: *Proceedings of the International Conference on Signals and Electronic Circuits (ICSES)*. Sept. 2010, pp. 343–346.

[175] H. J. Stolberg, M. Berekovic, L. Friebe, S. Moch, S. Flugel, X. Mao, M. B. Kulaczewski, H. Klussmann, and P. Pirsch. “HiBRID-SoC: a multi-core system-on-chip architecture for multimedia signal processing applications”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2003, pp. 8–13.

[176] R. Swaminathan and V. Goel. “TLM Signal: A non-memory mapped bus model”. In: *Proceedings of Indian SystemC User’s Group (ISCUG)*. Apr. 2013.

[177] S. Swan. *A Tutorial Introduction to the SystemC TLM Standard*. URL: http://www.ti.uni-tuebingen.de/uploads/media/Presentation-13-OSCI_2_swan.pdf.

[178] S. Swan, V. Motel, J. Cornet, and L. Maillet-Contoz. “Beyond TLM 2.0: New Virtual Platform Standards Proposals”. In: *Proceedings of the Design Automation Conference (DAC)*. June 2012.

[179] *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. Oct. 2015.

[180] TI. *OMAP 4470*. 2011. URL: http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?contentId=53243&navigationId=12843&templateId=6123.

[181] Tieto. *Virtual Platforms - Addressing challenges in telecom product development*. 2014. URL: https://www.tieto.com/sites/default/files/files/white_paper_-_virtual_platforms_in_telecom_v1.0.pdf.

[182] TutorialsPoint. *Operating System - Linux*. URL: https://www.tutorialspoint.com/operating_system/os_linux.htm.

[183] J. Urdahl, S. Udupi, D. Stoffel, and W. Kunz. “Formal system-on-chip verification: An operation-based methodology and its perspectives in low power design”. In: *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Sept. 2013, pp. 67–74.

[184] N. Ventroux, J. Peeters, T. Sassolas, and J. C. Hoe. “Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations”. In: *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. July 2014, pp. 250–257.

[185] N. Ventroux and T. Sassolas. “A new parallel SystemC kernel leveraging manycore architectures”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2016, pp. 487–492.

[186] S. Wallner. “Design methodology of a configurable system-on-chip architecture”. In: *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2004, pp. 283–284.

[187] X. Wang, W. Shan, and H. Liu. “Uniform SystemC Co-Simulation Methodology for System-on-Chip Designs”. In: *Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. Oct. 2012, pp. 261–267.

[188] Z. Wang, D. Zhang, X. Yu, Z. Yu, and X. Zeng. “A fast multi-core virtual platform and its application on software development”. In: *Proceedings of the International Conference on ASIC (ASICON)*. Oct. 2013, pp. 1–4.

[189] R. Weber. “An integrated hardware and software reuse environment for system development”. In: *Proceedings of the National Aerospace and Electronics Conference (NAECON)*. May 1991, 990–996 vol.3.

[190] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann. “SystemC-link: Parallel SystemC simulation using time-decoupled segments”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Mar. 2016, pp. 493–498.

[191] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G. Y. Wei. “A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for IoT applications”. In: *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2017, pp. 242–243.

[192] A. Wicaksana, C. M. Tang, and M. S. Ng. “A scalable and configurable Multiprocessor System-on-Chip (MPSoC) virtual platform for hardware and software co-design and co-verification”. In: *Proceedings of the International Conference on New Media (CONMEDIA)*. Nov. 2015, pp. 1–7.

[193] Wikichip. *EPYC 7401P - AMD*. 2017. URL: https://en.wikichip.org/wiki/amd/epyc/7401p.

[194] Wikipedia. *INI File Format*. URL: https://en.wikipedia.org/wiki/INI_file.

[195] Wikipedia, the free encyclopedia. *Design flow for a system-on-a-chip*. 2007. URL: https://commons.wikimedia.org/wiki/File:SoCDesignFlow.svg.

[196] Wikipedia, the free encyclopedia. *Systems Engineering Process II*. 2005. URL: https://commons.wikimedia.org/wiki/File:Systems_Engineering_Process_II.svg.

[197] J. Wilson. “Hardware/software selected cycle solution”. In: *Proceedings of the International Workshop on Hardware/Software Codesign*. Sept. 1994, pp. 190–194.

[198] Xilinx. *Zynq-7000 All Programmable SoC Overview*. Xilinx Datasheet. 2016.

[199] L. D. Xu, W. He, and S. Li. “Internet of Things in Industries: A Survey”. In: *IEEE Transactions on Industrial Informatics* 10.4 (Nov. 2014), pp. 2233–2243.

[200] C. C. Yang, N. H. Chang, S. L. Chen, W. D. Chien, C. S. Chen, C. M. Wu, and C. M. Huang. “A novel methodology for Multi-Project System-on-a-Chip”. In: *Proceedings of the International IEEE SoC (System-on-Chip) Conference (SOCC)*. Sept. 2011, pp. 308–311.

[201] T. C. Yeh, Z. Y. Lin, and M. C. Chiang. “Enabling TLM-2.0 interface on QEMU and SystemC-based virtual platform”. In: *Proceedings of the IEEE International Conference on Integrated Circuit Design and Technology (ICICDT)*. May 2011, pp. 1–4.

[202] T.-C. Yeh and M.-C. Chiang. “On the interface between QEMU and SystemC for hardware modeling”. In: *Proceedings of the International Symposium on Next-generation Electronics (ISNE)*. May 2010, pp. 73–76.

[203] S. Yoo, M. W. Youssef, A. Bouchhima, A. A. Jerraya, and M. Diaz-Nava. “Multi-processor SoC design methodology using a concept of two-layer hardware-dependent software”. In: *Proceedings of the Design, Automation, and Test in Europe (DATE)*. Vol. 2. Feb. 2004, pp. 1382–1383.

[204] P. G. Z. Zhou D. Parikh and A. Kwatra. “Switching Mechanism in Mixed TLM-2.0 LT/AT System”. In: *Proceedings of the Design Automation Conference (DAC)*. June 2009.

[205] Z. Zhang and X. Koutsoukos. “Modeling Time-Triggered Ethernet in SystemC/TLM for Virtual Prototyping of Cyber-Physical Systems”. In: *Proceedings of the International Embedded Systems Symposium (IESS)*. Ed. by G. Schirner, M. Götz, A. Rettberg, M. C. Zanella, and F. J. Rammig. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2013, pp. 318–330. URL: http://dx.doi.org/10.1007/978-3-642-38853-8_29.

[206] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun. “A Parallel SystemC Environment: ArchSC”. In: *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2009, pp. 617–623.

Publications

***

International conferences with proceedings

G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego. “QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0”. In: *Proceedings of the European Congress Embedded Real Time Software And Systems (ERTS)*. Toulouse, France, Jan. 2016

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Beyond QBox: development of virtual platforms based on QEMU and SystemC TLM-2.0”. In: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Barcelona, Spain, Sept. 2015

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Analysis of TLM-2.0 and it’s Applicability to Non Memory Mapped Interfaces”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. San Jose, USA, Feb. 2016

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “The missing SystemC and TLM asynchronous features enabling inter-simulation synchronization”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Munich, Germany, Oct. 2016

Invited presentation

G. Delbergue and T. Wieman. “SystemC Configuration, A preview of the draft standard”. In: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Nov. 2016

International conferences without proceedings

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Multi-threaded Virtual Platform Simulation: An open-source approach, using SystemC TLM-2.0, and QEMU”. in: *Proceedings of the Forum on Specification and Design Languages (FDL)*. Barcelona, Spain, Sept. 2015

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Virtual platform(s) simulation: an open-source, reusable, affordable and structured approach based on TLM/CCI”. in: *Proceedings of the Design and Verification Conference and Exhibition Europe (DVConEU)*. Munich, Germany, Nov. 2015

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Analysis of TLM-2.0 and it’s applicability to non memory mapped interfaces”. In: *Proceedings of the SystemC Evolution Day*. Munich, Germany, May 2016

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Open Source Heterogeneous AMP Virtual Platforms built using the SystemC and TLM standards”. In: *Proceedings of the Open Source Digital Design Conference (ORConf)*. Bologna, Italy, Oct. 2016

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “A SystemC asynchronous wait mechanism enabling multi-threading and multi-simulator support”. In: *Proceedings of the SystemC Evolution Day*. Munich, Germany, May 2016

National conferences with proceedings

G. Delbergue, M. Burton, B. Le Gal, and C. Jego. “Plateformes virtuelles SystemC/TLM : configuration, communication et parallélisation”. In: *Proceedings of the Groupement De Recherche System On Chip et System-In-Package (GDR SoC-SiP)*. Bordeaux, France, June 2017

Résumé étendu

***

Introduction

Les systèmes embarqués sont des systèmes dédiés pour une fonction particulière. Le nombre de ces systèmes va continuer d'augmenter dans les prochaines années. Pour des questions de réduction de coût, les SoCs sont apparus. Ils intègrent de nombreuses fonctionnalités. De nouvelles méthodologies ont dû être introduites, notamment à base de simulation et de plateforme virtuelle. Cette thèse propose ainsi des améliorations autour des plateformes virtuelles. Elle est composée de cinq chapitres.

Vue d'ensemble de la conception d'un SoC

Ce premier chapitre commence par un historique sur les circuits intégrés. Ensuite, il s'intéresse plus particulièrement aux systèmes sur puce, et notamment sa conception. En effet, son coût de fabrication mais aussi sa complexité ne cesse de s'accroître. Afin de pallier à cela, des méthologodies ont été mises en place et sont présentées. Il y a notamment la simulation et les plateformes virtuelles. Ces plateformes virtuelles sont des systèmes logiciels qui permettent de simuler entièrement le comportement du matériel (SoC, FPGA, ...) et d'exécuter du logiciel embarqué.

Le langage SystemC/TLM est ensuite présenté. Il s'agit d'un langage basé sur le C++ qui permet de décrire des systèmes matériels. Les classes permettent de décrire des modèles, qui constituent les plateformes virtuelles. Ces modèles peuvent être décrits à différents niveaux d'abstractions et interconnectés à travers des canaux de communication. SystemC se base sur un simulateur à événement discret. Ensuite, TLM-1.0 est présenté. Il permet d'accélérer les communications entre les modèles. Cependant, le standard ne répond pas à tous les besoins en matière d'intéropérabilité dans les communications.

C'est pourquoi TLM-2.0 a été introduit. Il améliore l'intéropérabilité, accélère les communications et ajoute de nouveaux niveaux d'abstraction. TLM-2.0 introduit une nouvelle façon de gérer le temps à travers un mécanisme de quantum. Finalement, SystemC et TLM-2.0 offrent des solutions standard qui permettent de répondre aux besoins pour la modélisation des systèmes sur puce. Les différents défis des plateformes virtuelles sont ensuite présentés. Ils se composent de trois parties.

Tout d'abord il y a la configuration des plateformes virtuelles. Aujourd'hui, durant le début de la conception d'un système sur puce, les spécifications exactes peuvent être manquantes. Pour cela, des modifications dans les modèles des plateformes virtuelles permettent de tester différentes combinaisons. Cependant, il n'existe pas de standard permettant de modifier de

manière intéroperable certains éléments des modèles, comme par exemple la taille de la mémoire.

Ensuite, le problème des communications dans les plateformes virtuelles est présenté. En effet, les systèmes sur puces sont composés de nombreux protocoles, dont les protocoles sans adressage mémoire. Cependant, le standard TLM-2.0 ne répond pas tout à fait au besoin afin de les modéliser.

Finalement, avec la multiplication du nombre de fonctionnalités et de coeurs de CPU dans les SoC, la vitesse de simulation est impactée. A cause des propriétés de SystemC, il n'est pas évident de tirer pleinement parti de la machine exécutant la simulation.

Configuration, Contrôle et Inspection

Aujourd'hui les systèmes sur puce contiennent de nombreux périphériques. Ceux-ci sont typiquement caractérisés par la taille de la mémoire, le binaire, le nombre de processeurs, etc. Cependant, depuis le premier développement de la plateforme virtuelle, une approche de configuration et d'inspection est une fonctionnalité manquante. Une étude des besoins autour de la configuration est décrite dans ce chapitre. L'approche sans solution de configuration est aussi évaluée. Elle montre que les limites en matière de flexibilité sont vite atteintes.

Un état de l'art des solutions existantes est ensuite proposé. Tout d'abord les solutions permettant de configurer uniquement des modèles sont proposées. Ensuite, des solutions plus globales qui permettent de configurer une plateforme virtuelle sont étudiés dans son ensemble. Enfin, des changements dynamiques ainsi que la rétrocompatibilité sont aussi étudies. Finalement, il en ressort que le problème d'intéropérabilité entre les différentes solutions de configuration n'a pas été suffisament adressé. En effet, le problème n'est pas l'implémentation d'une solution de configuration mais d'une solution permettant d'utiliser toutes les solutions de configuration ensemble.

La solution s'appelant CCI est alors présentée. Les éléments-clés de cette solution sont le paramètre et le broker. Le paramètre est le couple d'une donnée et d'un nom. Le broker gère les paramètres enregistrés dans la simulation. Une étude détaillée de l'architecture est ensuite décrite. Elle permet notamment de pouvoir interfacer plusieurs solutions de configuration ensemble. Enfin un mécanisme de notification à base de fonction de rappel est proposé.

Finalement, les performances de la solution ont été analysés. Un premier test brut a permis d'évaluer le surcoût dans l'utilisation de paramètres CCI lors d'écritures ou lectures. Les résultats montrent que les paramètres sont plus lents que des types natifs. Cependant ils offrent une plus grande flexibilité ainsi que des fonctionnalités que les types natifs n'ont pas.

En conclusion, mes contributions ont permis d'introduire une architecture permettant la configuration de plateforme virtuelle. Pour la première fois, l'intéropérabilité et la retro compatibilité avec les solutions existantes a été considérées.

TLM pour les protocoles sans adressage mémoire

Alors que l'ambition de TLM était large, le standard a en réalité principalement permis l'interopérabilité pour les protocoles à adressage mémoire. Cependant, les systèmes sur puces sont composés à

la fois de protocoles avec et sans adressage mémoire. Tout d'abord, la modélisation des communications dans les plateformes virtuelles est étudiée. Ensuite, un état de l'art des différentes solutions permettant de modéliser les communications à haut niveau d'abstraction est présentée. Ensuite, les protocoles sans adressage mémoire sont études selon trois familles : 1-1, 1-n, n-n. Finalement il en ressort que de nombreux points sont manquants dans le standard TLM-2.0 pour une modélisation native des protocoles sans adressage mémoire.

Une solution d'amélioration du standard est donc proposée. Elle consiste en une refactorisation du code afin de pouvoir l'étendre plus facilement tout en conservant la compatibilité avec les modèles existants. Ainsi différentes modifications au sein des parties de transport, socket, binding, et payload sont apportés.

Ensuite, une solution de vérification de la configuration du protocole en utilisant CCI est présentée. En effet, les protocols sont composés de méta données. Ceux-ci doivent correspondre de chaque côté de la transaction. Pour cela, des paramètres CCI sont ajoutés au seins des sockets TLM. Une mécanisme de notification est ensuite utilisée afin de vérifier que les meta-données sont équivalentes de chaque côté.

En conclusion, mes contributions ont permis d'introduire une évolution du standard TLM-2.0 permettant d'améliorer l'intéroperabilité et d'ajouter plus facilement le support des protocoles sans adressage mémoire.

Parallélisation dans SystemC/TLM

Dans ce chapitre, la parallélisation de l'exécution de la simulation est étudiée. En effet, l'augmentation du nombre de coeurs de CPU au sein d'une même puce impacte directement le temps de simulation. Afin de réduire ce temps, l'exécution de modèles en parallèle du noyau SystemC semble être une solution. Tout d'abord, un état de l'art des différentes solutions est proposé. Il étudie les différents travaux scientifiques existant à travers trois types de parallélisation différents : la parallélisation à l'intérieur du noyau SystemC, l'utilisation de plusieurs noyaux SystemC sans le quantum et l'utilisation de plusieurs noyaux SystemC avec le quantum.

Cette étude a permis de mettre en avant les propriétés de l'ordonnaceur du noyau SystemC : la simulation s'arrête lorsqu'il n'y a plus d'événements. Même si SystemC propose un mécanisme permettant de notifier qu'un événement asynchrone a été posté, il n'existe pas de mécanisme permettant de notifier à l'ordonnaceur qu'un événement asynchrone va être posté afin d'éviter la fin de la simulation. Un verrou peut être utilisé pour résoudre ce problème. Cependant, il ne garantit pas l'interblocage dans le cas de l'utilisation de différentes solutions au sein d'une même simulation.

Une première solution modifiant le noyau est proposée. Elle ajoute un nouveau type d'événement asynchrone ainsi que de nouvelles sémantiques `wait` permettant de faire de l'attente sur ces événements. Cependant, après une étude plus approfondie, de problèmes ont été découverts.

Une seconde approche est ensuite présentée. Elle consiste à l'ajout de nouvelles sémantiques pour les canaux primaires de communication mais aussi une modification de l'ordonnaceur. De nouvelles fonctions permettent ainsi d'indiquer au noyau qu'un canal primaire de communication va recevoir des événements asynchrones. Ce mécanisme fait désormais partie de SystemC 2.3.2.

Différentes solutions de synchronisation utilisant le quantum et la deuxième sont présentées.

Deux cas d'études sont présentés dans la partie expérimentale. Ils étudient la synchronisation de deux noyaux SystemC avec et sans synchronisation du temps en fonction des approches.

Application

Les chapitres précédents ont présenté différentes contributions pour apporter des solutions aux nouveaux défis de la modélisation des systèmes sur puce. Ce chapitre présente l'application de ceux-ci dans le cas d'un système IOT permettant de controller les luminaires d'une ville.

Le système est composé d'une passerelle et de noeuds. Chaque plateforme est ainsi composée d'un système sur puce composé d'au moins un CPU. Afin de faciliter la modélisation de CPUs, une solution est présentée : QBox. QBox fourni différents modèles de CPUs en se basant sur QEMU.

Ainsi, le système luminaire est étudié sous différents angles : impact de la parallélisation, configuration, etc. Les résultats expérimentaux montrent tout d'abord qu'il existe un temps de simulation minimal pour une certaine valeur de quantum. Une accélération quasi linéaire est obtenue lors de la multiplication du nombre de noeuds dans le cas de modèles avec une utilisation intensive du CPU. Enfin, une amélioration du Quantum Keeper est proposée. Elle consiste en l'ajout d'un mécanisme de notification afin de diminuer la dépendance entre la vitesse de simulation et la valeur du quantum.

Conclusion

En conclusion, avec la multiplication des fonctionnalités intégrées au sein d'une seule puce, il est nécessaire d'améliorer le flot de conception des systèmes sur puce et plus précisément celui des plateformes virtuelles dans le cadre de cette thèse. Une solution permettant la configuration de modèles SystemC/TLM a été proposée. Cette dernière fait désormais partie du standard CCI. La modélisation de protocoles de communication à un haut niveau d'abstraction a été étudiée et une évolution du standard actuel permettant d'améliorer le support, l'interopérabilité, la réutilisation a été proposée. Enfin, une évolution du standard SystemC et plus précisément du comportement du noyau de simulation a été étudiée pour supporter l'attente d'événements asynchrones. Ce type d'événement ouvre la voie à la parallélisation et la distribution de modèles sur différents threads / machines. Finalement, toutes ces contributions ont été associées à travers la modélisation d'un ensemble d'objets connectés à une passerelle.

Abstract

The market for Internet Of Things (IOT) is on the rise [140]. It is predicted to continue to grow at a sustained pace in the coming years. Connected objects are composed of dedicated electronic components, processors and software. The design of such systems is today a challenge from an industrial point of view [168][49]. This challenge is reinforced by market competition and time to market that directly impact the success of a system [199]. In a current design process involves the development of a specification. Initially, the team in charge of hardware development begins to design the system. Second, the application part can be done by software developers. Once the first hardware prototype is available, the software team can then integrate their part and try to validate the functionality. This step may reveal defects in the software but also in the hardware architecture. Unfortunately, the discovery of these errors occurs far too late in the design process, could impacts the marketing of the system and potentially its success. In order to ensure that the hardware and software designs will work together as early as possible, methodologies based on the SystemC / Transaction Level Modeling (TLM) standard have been widely adopted. They involve the modelling and simulation of the proposed hardware architectures. During the initial phases of a product's design, they enable the software and hardware team to share a virtual version of the (future) system. This virtual version is more commonly referred to as a virtual platform. It facilitates early software development, test and validation; reduces material cost by limiting the number of prototypes; saves time and money by reducing risks. However, connected objects are increasingly incorporating hardware and software features. As the requirements have evolved, the SystemC / TLM simulation standard no longer meets all expectations. It includes aspects related to the simulation of systems composed of many functionality, disparate communication protocols but also complex and time consuming models during the simulation. Some works have already been carried out on these subjects. However, as the number of components increases, all forms of interoperability of models and tools become increasingly difficult to handle. Moreover, most of the research has resulted in solutions that are not inter-operable and can not reuse existing models. To solve these problems, this thesis proposes a solution for configuring SystemC / TLM models. It is now part of the standard Configuration, Control and Inspection (CCI). In a second step, the modeling of high-level abstraction communication protocols (TLM Loosely Timed (LT) and Approximately Timed (AT)) has been studied, as it relates to non-bus protocols. An evolution of the standard to improve support, interoperability and reuse is also proposed. In a third step, a change of the SystemC standard and more precisely of the behavior of the simulation kernel has been studied to support asynchronous events. These open the way to parallelization and distribution of models on different threads / machines. In a fourth step, a solution to integrate Central Processing Units (CPU) models integrated in Quick EMUlator (QEMU), a system emulator / virtualizer, has been studied. Finally, all these contributions have been applied in the modeling of a set of objects connected to a gateway.

Keywords: SystemC, TLM, Virtual Platform, Configuration, Communication, Parallelism

Résumé

Le marché de l'Internet des Objets (IdO) est en pleine progression. Il va continuer à croître et à se développer à un rythme soutenu dans les prochaines années. Les objets connectés sont constitués de composants électroniques dédiés, de processeurs et de codes logiciels. La conception de tels systèmes constitue aujourd'hui un challenge au niveau industriel. Ce challenge est renforcé par la concurrence du marché et le délai de commercialisation qui impactent directement sur le développement d'un système. Le processus de conception actuel consiste en l'élaboration d'un cahier des charges. Dans un premier temps, l'équipe en charge du développement matériel commence à développer le produit. Ensuite, la partie applicative peut être mise au point par les développeurs logiciels. Une fois le premier prototype matériel disponible, l'équipe logicielle peut alors intégrer sa partie et tenter de la valider fonctionnellement. Cette étape peut mettre en lumière des défauts dans le logiciel mais aussi lors de la conception matérielle. Malheureusement, la découverte ce type d'erreurs intervient beaucoup trop tard dans le processus de conception retardant la commercialisation du système. Afin de sécuriser au plus tôt les développements matériel et logiciel, des méthodologies basées sur le standard SystemC/Transaction Level Modeling (TLM) ont été proposées. Elles permettent de modéliser et de simuler du matériel. Durant les phases amont de conception d'un système, elles permettent de mettre en commun une version virtuelle du (futur) système entre les équipes logicielle et matérielle. Cette version virtuelle est plus couramment appelée plateforme virtuelle. Elle permet de tester et de valider le plus tôt possible lors du cycle de conception, de réduire le coût matériel en limitant la fabrication de prototypes, mais aussi de gagner du temps et donc de l'argent en diminuant les risques. Or, les objets intègrent de plus en plus de fonctionnalités aux niveaux matériel et logiciel. Les besoins ayant évolué, le standard de simulation SystemC/TLM ne répond plus à l'heure actuelle à toutes les attentes. Ces attentes concernent plus particulièrement les aspects liés à la simulation de systèmes composés de nombreuses fonctionnalités, de protocoles de communication disparates mais aussi de modèles complexes et consommateur de temps pendant la simulation. Des activités de recherche ont déjà été menées sur ces sujets. Cependant, elles ont pour la plupart abouti à des solutions qui ne sont pas interopérables. Les solutions existantes ne permettent donc pas de bénéficier de la réutilisation des modèles de la littérature. Afin de répondre à ces problèmes, une solution permettant la configuration de modèles SystemC/TLM a été recherchée. Cette dernière fait désormais partie du standard Configuration, Control and Inspection (CCI). Dans un second temps, la modélisation de protocoles de communication à un haut niveau d'abstraction (TLM Loosely Timed (LT) et Approximately Timed (AT)) a été étudiée, et plus précisément des protocoles de type non bus. Une évolution du standard actuel permettant d'améliorer le support, l'interopérabilité, la réutilisation a été proposée dans le cadre de la thèse. Ensuite, une évolution du standard SystemC et plus précisément du comportement du noyau de simulation a été étudiée pour supporter l'attente d'événements asynchrones. Ce type d'événement ouvre la voie à la parallélisation et la distribution de modèles sur différents threads / machines. Enfin, une solution permettant l'intégration de modèles de Central Processing Units (CPU) intégrés dans Quick EMUlator (QEMU), un émulateur / virtualisateur de système, a été étudiée. Finalement, toutes ces contributions ont été associées à travers la modélisation d'un ensemble d'objets connectés à une passerelle.

Mots clefs : SystemC, TLM, Plateforme Virtuelle, Configuration, Communication, Parallélisme