Université Libre de Bruxelles
Faculté des Sciences Appliquées

Année Académique 2008-2009

Developing Multi-Criteria Performance Estimation Tools for Systems-on-Chip

Promoteur:
Prof. F. Robert

Thèse présentée par Alexis Vander Biest
en vue de l'obtention du titre de
Docteur en Sciences Appliquées

Consultation ***AUTORISEE*** *(biffez la mention inutile)*

~~***INTERDITE***~~

Signature :

Université Libre de Bruxelles
Faculté des Sciences Appliquées

Année Académique 2008-2009

Developing Multi-Criteria Performance Estimation Tools for Systems-on-Chip

Promoteur:
Prof. F. Robert

Thèse présentée par Alexis Vander Biest
en vue de l'obtention du titre de
Docteur en Sciences Appliquées

Acknowledgments

My deepest gratitude goes to Prof. Frédéric Robert for his guidance and numerous technical advices all along the thesis.

I would like to thank Pr. Dragomir Milojevic for his constant involvement in my Ph.D. thesis and precious support during these last years.

I am also grateful to Prof. Pierre Mathys for his permanent availability and hosting me in the Bio, Electro and Mechanical Systems division at the ULB.

I also thank Cédric Hernalsteens for his precious help on Nessie's case study implementation and strong involvement during his few weeks work in our lab.

A very special thanks goes to Martin, Cédric and Michel who cheered me up when I was down and highly participated in the success of this work.

Aliénor, Anthony and Geoffrey helped me a lot by reading over and over several parts of this dissertation and giving me their opinion about different technical issues when I needed, my deepest thanks go to you.

My gratitude also goes to all my colleagues and friends at ULB: Gatien, Marc, Kevin, Vincent, Axel and Manu.

This research was funded by the Fond Régional de l'Industrie et de l'Agriculture (FRIA).

I would also like to thank all my friends for giving me support and cheering me up all the times I thought about giving up: Gilles, Noémie, Sébastien, Arnaud and Nicolas, I really owe you.

My deepest gratitude goes to my grandmother whose encouragement during all these years helped me to succeed in my studies and the achievement of this present work.

Finally, I am greately indebted to Sabrina for her patience and support even when I was working all day and night long: you've shown me that there is so much to expect from life after this Ph.D.

Abstract

The work presented in this thesis targets the analysis and implementation of multicriteria performance prediction methods for System-on-Chips (SoC).

These new SoC architectures offer the opportunity to integrate complete heterogeneous systems into a single chip and can be used to design battery powered handhelds, security critical systems, consumer electronics devices, etc. However, this variety in terms of application usually comes with a lot of different performance objectives like power consumption, yield, design cost, production cost, silicon area and many others. These performance requirements are often very difficult to meet together so that SoC design usually relies on making the right design choices and finding the best performance compromises.

In parallel with this architectural paradigm shift, new Very Deep Submicron (VDSM) silicon processes have more and more impact on the performances and deeply modify the way a VLSI system is designed even at the first stages of a design flow.

In such a context where many new technological and system related variables enter the game, early exploration of the impact of design choices becomes crucial to estimate the performance of the system to design and reduce its time-to-market.

In this context, this thesis presents:

- A study of state-of-the-art tools and methods used to estimate the performances of VLSI systems and an original classification based on several features and concepts that they use. Based on this comparison, we highlight their weaknesses and lacks to identify new opportunities in performance prediction.

- The definition of new concepts to enable the automatic exploration of large design spaces based on flexible performance criteria and degrees of freedom representing design choices

- The implementation of a couple of two new tools of our own:

  - Nessie, a tool enabling hierarchical representation of an application along with its platform and automatically performs the mapping and the estimation of their performance.

  - Yeti, a C++ library enabling the definition and value estimation of closed-formed expressions and table-based relations. It provides the user with input and model sensitivity analysis capability, simulation scripting, run-time building and automatic plotting of the results. Additionally, Yeti can work in stan-

dalone mode to provide the user with an independent framework for model estimation and analysis.

To demonstrate the use and interest of these tools, we provide in this thesis several case studies whose results are discussed and compared with the literature.

Using Yeti, we successfully reproduced the results of a model estimating multi-core computation power and extended them thanks to the representation flexibility of our tool. We also built several models from the ground up to help the dimensioning of interconnect links and clock frequency optimization.

Thanks to Nessie, we were able to reproduce the NoC power consumption results of an H.264/AVC decoding application running on a multicore platform. These results were then extended to the case of a 3D die stacked architecture and the performance benefits are then discussed.

We end up by highlighting the advantages of our technique and discuss future opportunities for performance prediction tools to explore.

Publication list

This thesis presents the results of my research; part of this work has been published in the following conference papers:

- A. Vander Biest, A. Richard, D. Milojevic, F. Robert, "A multi-objective and hierarchical exploration tool for SoC performance estimation", Lecture Notes in Computer Science, Vol. 5114, Pages 85-95, 2008
- A. Vander Biest, D. Milojevic, A. Richard, F. Robert, "Framework for fast performance evaluation of flexible models applied to interconnect delay", Proceedings of the DASIP conference (workshop on Design and Architectures for Signal and Image Processing), Grenoble, France, 27-29 September 2007
- A. Vander Biest, A. Richard, D. Milojevic, F. Robert, "A framework introducing model reversibility in SoC design space exploration", Lecture Notes in Computer Science, Vol. 4599, Pages 211-221, 2008
- A. Vander Biest, A. Leroy, D. Milojevic, F. Robert, "A flexible system-level design methodology applied to NoC", Special Inaugural Workshop on Future Interconnects and Networks on Chip, DATE (The Design, Automation, and Test in Europe conference), Munich, Germany, March 4-5 2006
- A. Vander Biest, D. Milojevic, F. Robert, "Key enablers for next generation system-level design in microelectronics", Proceedings of the 10th World Multi-Conference on Systemics, Cybernetics and Informatics conference (WMSCI), Orlando (Florida), 16-19 July 2006
- A. Vander Biest, F. Robert, D. Verkest, S. Vernalde "A taxonomy for technology extrapolation", Proceedings of the 10th World Multi-Conference on Systemics, Cybernetics and Informatics conference (WMSCI), Oulu, Finland, 21-22 November 2005

Contents

List of Tables

List of Figures

List of Acronyms

ASIC	Application-Specific Integrated Circuit
ASIP	Application Specific Instruction Processor
AVC	Advanced Video Coding
CFSM	Codesign Finite State Machine
CIF	Common Image Format
CPI	Cycles Per Instruction
CPU	Central Processor Unit
CDFG	Control Data Flow Graph
CSDF	Cyclo-Static Data Flow
CSP	Communicating Sequential Processes
CT	Continuous Time
DCT	Discrete cosine transform
DE	Discrete Event
DFG	Data Flow Graph
DoF	Degree of Freedom
EMIF	External Memory Interface
FCFS	First Come First Served
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
HCDFG	Hierarchical Control Data Flow Graph
HDTV	High Definition Television
HW	Hardware
IDCT	Inverse Discrete cosine transform
ILP	Instruction Level Parallelism
IP	Intellectual Property
IPC	Instructions Per Cycle
KPN	Kahn Process Network
MIPS	Million Instructions Per Second
MoC	Model of Computation
MPSOC	Multiple Processor System-on-Chip
MPU	Micro-Processor Unit

NIU	Network Interface Unit
NOC	Network On Chip
OBIS	Optimal Buffer Insertion and Sizing
PN	Petri Networks
RTL	Register Transfer Level
SAIF	Switching Activity Interchange Format
SDF	Synchronous Data Flow
SoC	System-on-Chip
SW	Software
TLM	Transaction Level Modeling
TLP	Thread Level Parallelism
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
XML	eXtensible Markup Language

Introduction

When introducing the general context of a thesis dissertation, it's always difficult to chose between presenting the naughty problem that we try to solve or the great opportunities we could benefit from. So why wouldn't we simply present two different versions dealing with each aspect?

Version 1: the bright side

Imagine you in five years: you are walking out of the shop with your brand new handheld in the pocket. It's a wonderful mobile device that has all the features you have ever imagined: integrating a GPS, this phone disposes of powerful wireless functionalities like videoconference, web browsing and communication with all other surrounding embedded devices allowing their control. Beside these common functionalities, it is also a powerful organizer, a 3D gaming station and a MPEG 4 video decoder making it very convenient for entertainment.
You think back nostalgically on the past ten years: people had their pockets full of devices to benefit from all these functionalities but now everything is integrated. In fact, you don't know that this evolution is the product of researches in microelectronics that lead to System-on-Chips allowing the integration of the whole electronic system into one single die.
Lost in your thoughts, you even forget the numerous and almost invisible sensors inserted in your garments that monitor your heart frequency, your body temperature and can communicate with the handheld to contact the nearest hospital in case of emergency. Since you own your garments, you have never changed their batteries: they are low-powered and some are even taking their energy from the body movement.
The technological future is bright and full of conveniences, isn't it?

Version 2: the dark side

Imagine you in five years: you are exhausted and worried, working day and night to finish your project before the deadline. No, you're not a PhD student, it's much worse: you are an integrated circuit designer. Don't tell me you haven't been warned enough: during your studies and even after, people around you kept talking about increasing IC complexity, unmanageable projects and technological issues. Researchers wrote tenths of

papers to describe these wicked Deep sub-micron (DSM) effects that occur when transistor size scale down under 100nm. Who could believe that a few billion transistors 35nm chip could draw a total current of a few tenths A in its 10 km wires labyrinthine network? But now, the fiction has come true : DSM effects make integrated circuits suffer from behaviour uncertainties due to process variations, burn a lot power even when the chip is not processing, make the manufacturing times and costs explode... Now, you're going back to work with your team in the building where 800 persons work on the same project as you since one year. The design tools that you are using are still optimising your integrated circuit and you know that this optimisation step will once again fail leading you to iterate this operation once again.

Could your situation be worse ? Yes, you could have been one of the project leaders.

The subject of this thesis revolves around opportunities, methods and tools for multicriteria performance estimation of heterogeneous System-on-Chip architectures. The main contribution of this work lies in the design and implementation of two different tools called Nessie and Yeti whose combination enables fast design space exploration and flexible performance estimation.
Yeti is a tool enabling the flexible representation, combination and use of analytical/table-based models to capture the link between performance and all their related parameters. Aside from enabling dynamic model building for model-based sensitivity analysis, Yeti enables fast input sensitivity studies combined to powerful scripting and integrated plotting functionalities.
Nessie is an automatic design space exploration tool capturing in a hierarchical way the functionality along with the platform and enabling their automatic mapping. Tailored to support user-defined performance criteria, Nessie banalizes all possible design choices as possible degrees of freedom and enables the use of any customizable exploration policy.

The thesis is organized as follows:

Chapter 1 introduces the context of the thesis.

Chapter 2 describes the concepts and implementation of Yeti.

Chapter 3 studies different design cases to demonstrate the main features of Yeti.

Chapter 4 defines the state of the art related to performance estimation tools and methods.

Chapter 5 describes the concepts and implementation of Nessie.

Chapter 6 demonstrates the features of Nessie on different case studies including the performance estimation of an H.264/AVC application running on a MPSoC.

Chapter 7 identifies future opportunities for the development of our tools.

Chapter 8 concludes this thesis.

Chapter 1

Context and motivation

Abstract

In this first chapter, we introduce the context of this work and explain why performance prediction is so important in a VLSI design process. We first start by presenting a brief history of the design process representations from which we extract the main aspects of current VLSI design and challenges while focusing our talk on design hierarchy and functionality/platform separation. To highlight the close interaction between technology and the first steps of the design process, we discuss some new technology shifts and their impact on design.

Based on this context, we present our formalization of the design process through which we explain the concept of design iteration, its causes and why it is harmful the design. We finally propose performance prediction as a viable yet simple solution to enhance design process and quality and briefly summarize our personal contribution to this domain that will be presented in this thesis.

1.1 Introduction

Nowadays, Moore's law[1] has enabled such a high level of silicon integration that we have reached a point where a complete microcontroller is taking a ridiculously small amount of square millimeters. Now that more than 1 billion transistors can fit the same chip, designers are able to integrate several components on a single die and build up heterogeneous platforms called *systems-on-chip*[2]. These architectures combine the respective advantages of different components allowing the co-existence of FPGA's and processors communicating through complex networks-on-chip or mixing analog and digital blocks onto the same chip.

All this extra silicon is not too much to implement all the new ideas that the system level guys can come up with: multi-standard video decoding platforms, software defined radio, wireless reconfigurable handhelds and many more. The question of converting ef-

ficiently this silicon into chips running these new fancy applications is however far from being obvious putting a lot of pressure on the design. Not only does it have to result in a chip running the desired application but non-functional aspects like power consumption and reliability also need to be taken into account. Furthermore, the design itself must be optimized to ensure reasonable complexity, minimize cost, lower time-to-market and guarantee high yield and cheap manufacturing. Considering this new perspective and all the possibilities offered by extra silicon, the scope of the problem becomes way larger than it was before in VLSI design: this introduction chapter focuses on these issues and identifies opportunities for this dissertation.

In this chapter, we will first describe in Sec.1.2 the evolution of the design process representation over the last thirty years which will give us some insight inside nowadays design methodology and the rise of the abstraction levels used for designing. Sec.1.3 will show us how technology related issues begin to deeply impact the first steps of a design which makes system-level design decisions more and more dependent on silicon processes and may compromise time-to-market by adding extra iterations to the design process (see Sec1.4). Finally, Sec.1.5 discusses methods to optimize and guide the design process by using a priori performance estimation: our contribution to this domain is then presented.

1.2 Design

In this section, we will present different design and design process representations and see how their evolution over time has progressively emphasized two major current trends of EDA: design abstraction and functionality/platform co-design.

1.2.1 A bit of history: design process representation evolution

We will now present some of the major design process representations in chronological order and put the spell on how they captured the hierarchical representation of an application and its platform along with the progressive rise over time of the associated abstraction levels.

Y-chart First introduced in the early eighties, the first Y-chart[3] is the most famous design representation widely used in microelectronics courses to introduce VLSI design to student[1]. The Y-chart depicted in Fig.1.1 is organized in three axes:

- The *behavioural* axis describes what the designed system is supposed to do without any information about its implementation.
- The *structural* axis is the bridge between the behavioural and physical axis: it represents the mapping of the functionality onto a set of components and communication

[1] There is an extension of the Y-chart defined a few years later and called the X-chart[4] that adds one axis for *testing* representation to the original representation. We thought it useful to mention it even if the X-chart is much less used than the Y-chart.

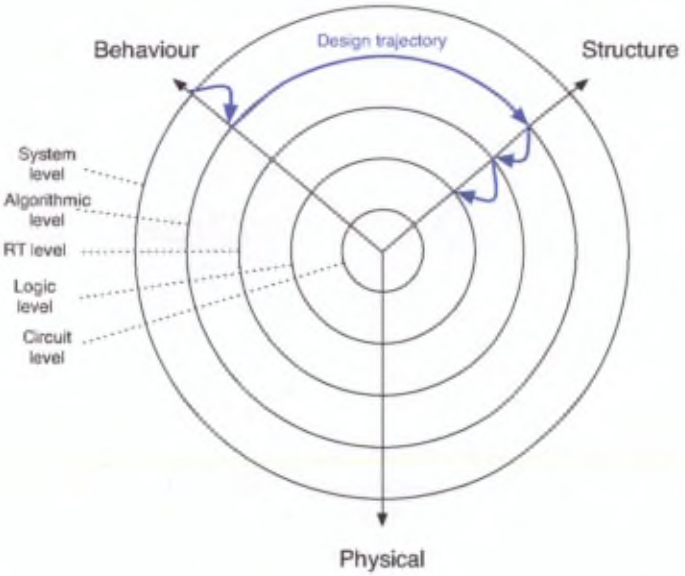

Figure 1.1: Representation of a design using the 3-axis Y chart[3]

primitives. Physical components are represented as concurrent processes interacting to execute the functionality.

- The *geometrical* or *physical* axis is totally abstracted of the functionality and focuses on the spatial organization of the physical blocks described by the structural axis.

Concentric circles surrounding the axes describe the different levels of abstraction: the closer we are to the Y-chart centre, the less implementation details we hide in the system representation. Originally the Y-chart abstraction levels were limited to the register transfer, logic and circuit levels[5] but with the evolution of VLSI design in twenty five years, the algorithmic and system level were progressively added on top of it.
If the Y-chart succeeds in hierarchically representing the design itself, it is however not very appropriate for describing the design process: only simple design trajectories roughly capturing the design activity can be represented on the Y-chart[6]. Refinement (describing an axis in the lower abstraction level) and synthesis (mapping of the functionality on the physical architecture i.e. moving from behavioural to structural axis) can be represented while any parallel design task or step simultaneously involving different aspects of the design cannot. These limitations, recognized by Gajski, the author himself, lead him to rethink its design process representation and propose a new one called the Rugby model.

The Rugby model This model was developed in 1999 to extend the deprecated Y-chart to current design methodologies and enable a clarified distinction between the representation of the design and the design process itself[7].
Depicted in Fig.1.2, the rugby models introduces four different domains:

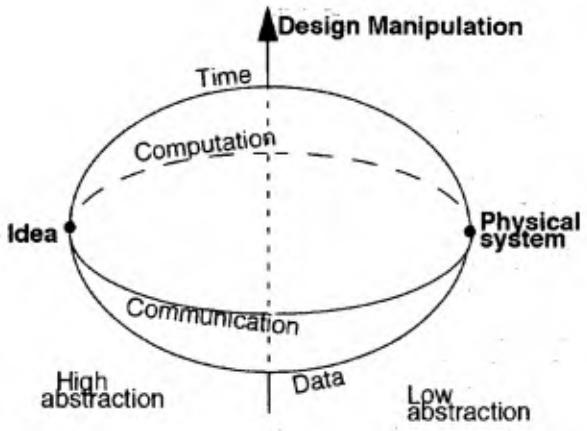

Figure 1.2: Design and design process representation using the 5-axis rugby model[7]

- The computation domain is a restricted version of the Y-chart functional axis
- The communication domain merges structural and physical axes that were indeed a bit redundant since they represented the same thing but at a logical and geometrical point of view
- The data domain focuses on the data types used at the different levels of abstraction
- The time domain deals with important aspects of timing (constraints, causality, clocked cycles etc.)

Additionally a fifth axis orthogonal to all the other ones represents the design manipulations allowing a separate description of the design process in different identified steps.
The rugby model is a first step towards the description of more heterogeneous systems since each axis may be divided in several branches with specific semantics that are more appropriate depending on the targeted design style (FPGA or microprocessor for instance). The general idea behind this representation is the following: as we further refine the idea of the system, the distinction between the four different aspects described by the axis becomes clearer and more details are added to the design. When we get close to the silicon implementation, the distinction between functionality and platform becomes more blur as they become deeply merged. This is why the model has a rugby shape with its axes diverging and converging for the higher and lower abstraction levels. This concept shows the fact that the first steps of the design could benefit from anticipated information of the lower abstraction levels where more details are available to characterize both the design and design process performances.

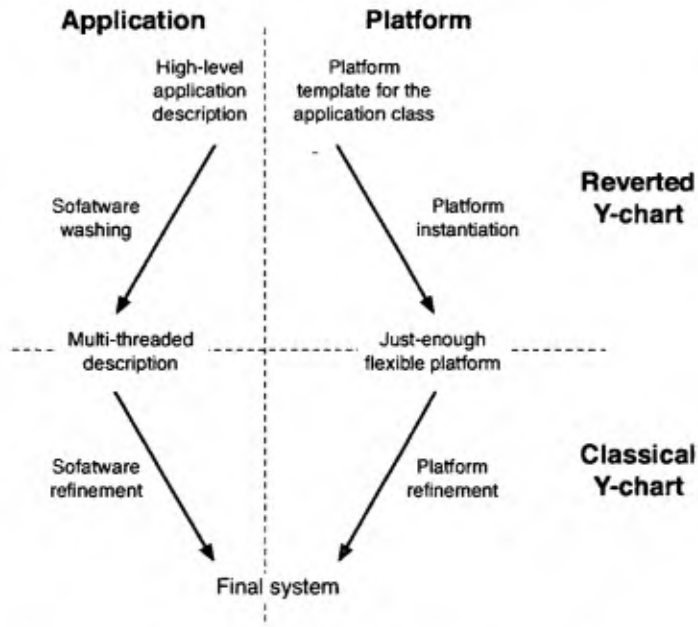

Figure 1.3: Double-Y chart methodology

Double Y-chart In 2004 IMEC[2] came up with the idea of adding a reverted Y-chart on top of the existing Y-chart in order to propose an efficient methodology for the design of complex systems[8]. As depicted in Fig.1.3, the left top branch consists in optimizing an input code and transforming it so that it can be efficiently mapped onto a generic platform. The right top branch customizes a just flexible enough platform template and instantiates it by defining the computation and communication resources to propose a tailored platform for the functionality. Once these operations have been performed, the classical bottom Y-chart methodology can be applied.
What's interesting about this double Y-chart is the recurring idea of the functionality/-platform separation in design along with the definition of a platform template easy to tune for a specific application[9].

VSIA taxonomy In 2001 the VSIA alliance, an international organization constituted by many industrial actors involved in SoC design, released a document describing a taxonomy for the description of VLSI systems. This taxonomy consists of five different axes describing the various aspects of the system with different resolutions corresponding to the abstraction levels (see Fig.1.4):

- *Temporal resolution*: captures time with different accuracy levels (causality, transactions, clock cycles)

[2]IMEC is a research center located in Leuven Belgium and is currently one of the major actor in microelectronics research.

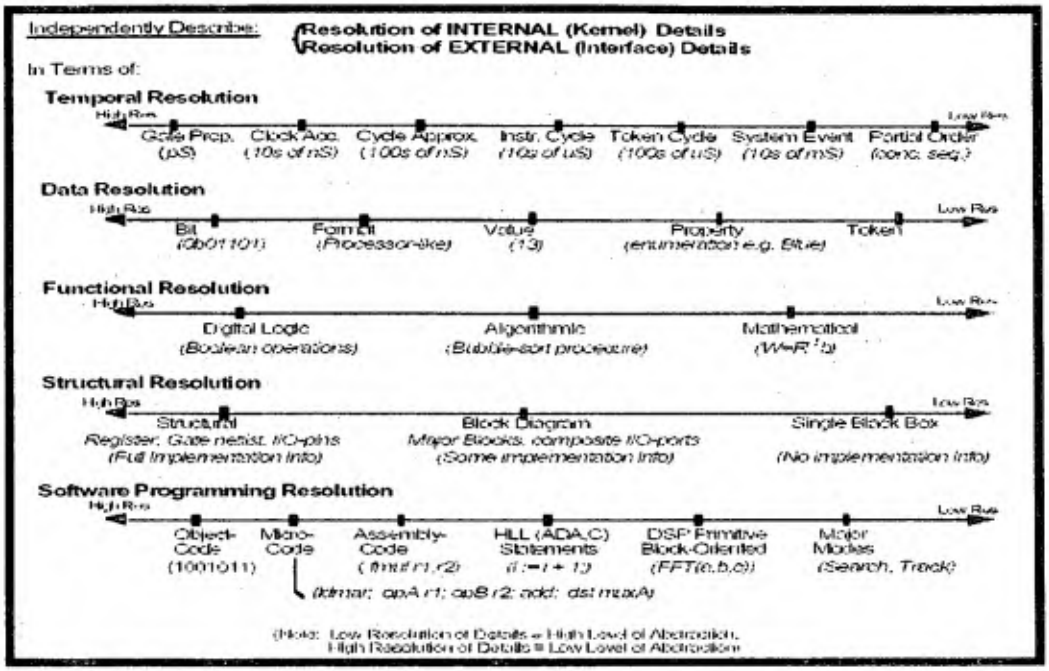

Figure 1.4: The five resolutions axes of the VSIA taxonomy defining a representation of a VLSI system[10]

- *Data resolution*: defines the granularity and types of data used for computation
- *Functional resolution*: defines the basic operations that are used at each abstraction level to specify the functionality (bitwise operations, boolean operations, mathematical operators, algorithms etc.)
- *Structural resolution*: captures the description of the physical structure of the system
- *Software programming resolution*: defines the level of abstraction used to program the behaviour of the system (binary code, assembly code, objects etc.)

This taxonomy can be useful to classify different computation models or languages[11] by considering the five different aspects described earlier. Interestingly we can see that this VSIA taxonomy again emphasizes the hierarchy composed out of different abstraction levels in the context of VLSI design and makes a clear distinction between what is functional-related (functional, data and temporal axis) and platform-related (structural axis).

From these previous examples, we can see that there is no unique design process representation: all of them have evolved over the last thirty years according to the different

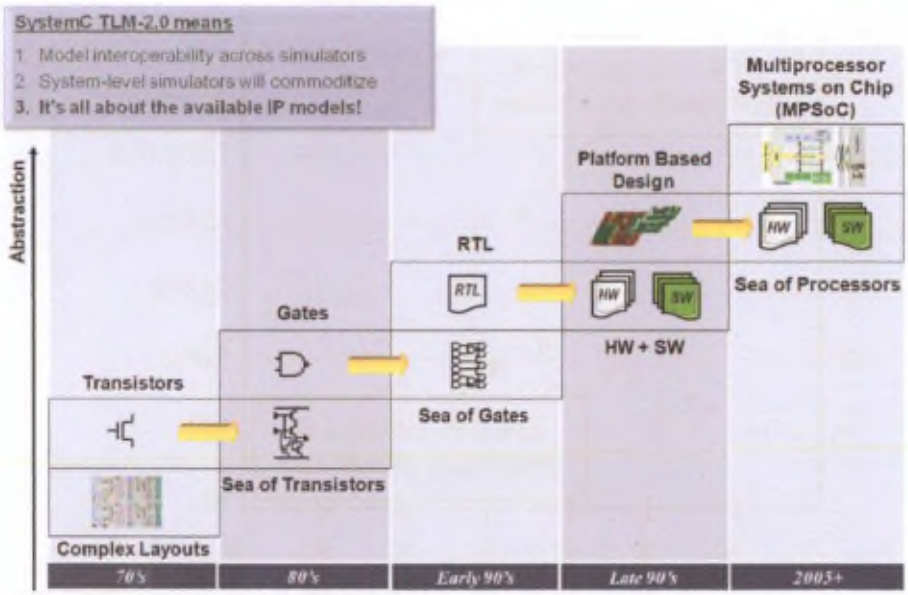

Figure 1.5: Evolution of the abstraction levels during the forty last years in the context of VLSI design[9]

design trends and are thus focusing on different aspects. We can however see two emerging common points that are at the centre of any of these representations: the hierarchy and the functionality/platform separation of concerns.
In our next section, both these topics in nowadays design will be further discussed as they are central elements of VLSI design and will be widely used during throughout this thesis.

1.2.2 VLSI Design nowadays

Design hierarchy During the last forty years, the growing integration density has incredibly increased the design complexity making it impossible to design a system transistor by transistor. This has lead to a progressive rise of the abstraction level used as a first step of the design as depicted in Fig.1.5. Each abstraction level is based on a combination of the primitives of the lower level so that the transistor and gate levels have not disappeared from nowadays design flow but are now the latest performed steps. At the top of this hierarchy we have now multiprocessor system-on-chips architectures offering high performance and allowing the designer to mix different design styles (FPGA's, microprocessors, ASIC's). System-level design is now the first design task consisting in splitting the system in different blocks, choosing the IP cores that will need to be used and the blocks that will have to be designed from scratch. This step is left to the designer while lower abstraction levels design activities require less and less manual intervention due to the increased number of blocks that have to be manipulated at these low levels.
With the growing complexity of VLSI systems and their design, we have increased the

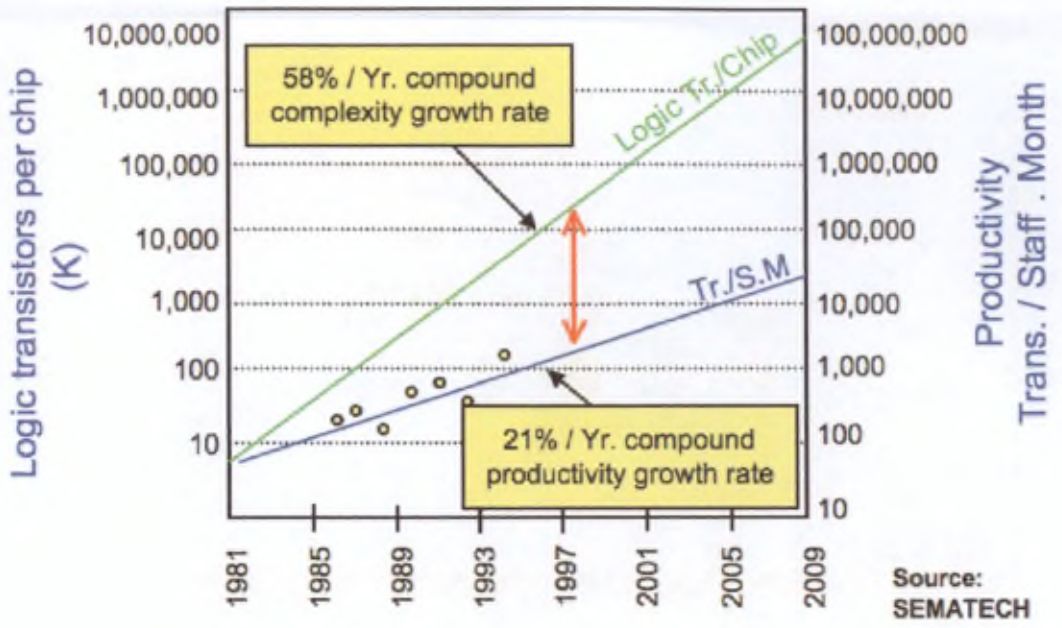

Figure 1.6: Design productivity gap: design complexity versus designer productivity[12]

number of design decisions that have to be made through the whole flow. From the partitioning of the system to the choice of the technology used to manufacture the die, each abstraction level proposes a growing number of options over the years which makes design decision very difficult to make.

Abstraction is not only a way to deal with complexity but also increases design productivity. Fig.1.6 is commonly used to present the so-called *Design Productivity Gap*[12] and has been originally drawn by Sematec based on numbers taken from existing designs. This graph compares the evolution of the design complexity defined by Moore's law with the design productivity i.e. the numbers of transistors designed per man-month (based on previous observations and future estimations). As we can read from Fig.1.6, the complexity grows at a 68% rate per year while the productivity only increases at a 21% pace per year: this graph thus predicts an exponential growth of the design cost due to dramatic staff growing to compensate for the slower productivity growth rate. The rise of the abstraction levels is a part of the answer to that problem allowing the designer to be more efficient by leaving the repetitive and time-consuming work of the lower abstraction levels to automated EDA tools. Furthermore the use of existing IP blocks and multicore architectures makes the complexity grow slower than the number of transistors in practice since all these transistors don't need to be designed from scratch. In return, the designer has many design options to examine which makes the first choices much more crucial than before as they will condition the performance results.

Functionality and platform Since the last fifteen years, complex systems are now divided into two different parts during the first steps of the design flow: the functionality and the platform executing this functionality. This trend is clearly illustrated by the Rubgy model throwing out the confusing functional and structural axes for the computation/communication axes separating clearly what the system is supposed to do from the way it does it. With the new abstraction levels, this separation of concerns between the functionality and the platform becomes more and more important as it allows the designer to focus on the functionality (using for instance verification techniques from the software world) at the very first stages of the design without having to be bothered by detailed hardware concerns. However the functionality/platform interaction still remains crucial as it defines the non-functional costs (power consumption, computation power etc.) and thus needs to be tested at every stage of the design process to evaluate the performances. In this context, a new standard at the transaction level modeling has recently emerged with the adoption of SystemC TLM 2.0[13] in an effort to replace heterogeneous proprietary solutions used throughout the flow by a common language enabling increased model interoperability. At such high abstraction levels as TLM, it's already possible to estimate the timing performances of a functionality/platform combination which is very valuable. However non-functional aspects like area, power and cost are not taken into account limiting at the moment the use of TLM to the early validation of functional properties of the system.

1.3 Technology evolution

The constant silicon manufacturing evolution has enabled a growing silicon integration level and largely increased the available number of transistors per chip. As the feature size has reached values below 90nm, some new effects (called DSM or *deep-submicron*) begin to deeply modify well known and established rules of thumb[14]. Getting rid of the problems or limitations involved by the use of these technologies will however, as we discuss in the following, require some pretty heavy design modifications at much higher abstraction levels than silicon process.

Static power Twenty years ago, CMOS-based transistors were known for their impressive ability to consume a ridiculous amount of static power compared to the dynamic power. From a designer point of view, optimizing the power consisted in minimizing the dynamic power P_{dyn} defined by Eq.1.1 with C_{switch} being the total capacitance to switch, f_{clock} the clock frequency and V_{dd} the supply voltage.

$$P_{dyn} = C_{switch} * f_{clock} * V_{dd}^2 \tag{1.1}$$

With the feature size shrinking, the gate capacitance scales quadratically and the voltage supply is also reduced so that the dynamic power has been reduced over years. The static power however grows at a very fast pace beyond 90nm due to current leaking in the OFF

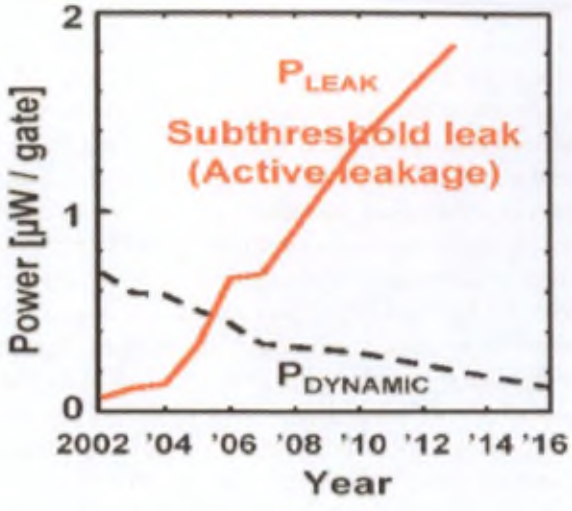

Figure 1.7: Dynamic and static power evolution over different technological nodes[15]

state[3]. This relative evolution of the static and dynamic power over the technology nodes is illustrated in Fig.1.7 where we can indeed see that gates begin to consume more static than dynamic power. Many techniques can be used to reduce the impact of static power at the process level but this shift has also some major implications at the architectural level. Minimizing the gate count to run a given functionality, using and managing mechanisms to bias the gate threshold voltage, modifying the scheduling to shut down as long as possible computation nodes and memory banks etc. are some of the techniques that deeply modify the design and the associated CAD tools.

Wire delay Looking back in the past, transistors had a very large delay compared to wires: it was all about minimizing the number of gates of the critical path regardless of the number of wire segments connecting them. But this trend has changed with feature size shrinking as it can be seen on Fig.1.8: while the gate delay decreases, the wire delay increases to the point where it becomes greater for technology nodes smaller than 250nm. If local wire delay remains constant with scaling because their length scales down with the feature size, global wires still have to cross the whole chip so that their delay grows if these wires are scaled[18]. This global wire delay increase has dramatic consequences on the layout and architectural design:

- Different scaling schemes are proposed for semi-global and global wires[19] to make the delay scale smoothly
- Since we expect global wire to work at higher frequencies for each new technology

[3]Many different current sources contribute to the total leakage current and result from very different phenomena: for more information, we advise the reader to refer to [16] and [17].

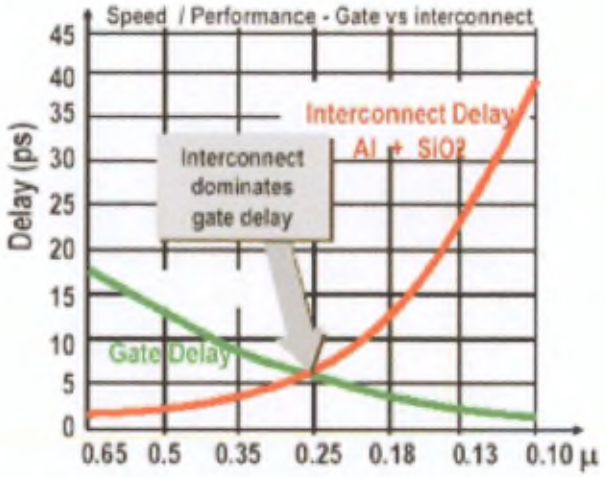

Figure 1.8: Gate and wire delay evolution over different technology nodes[23]

node, the concept of a completely synchronous chip with a single clock becomes obsolete. New architectures known as GALS (Globally Asynchronous Locally Synchronous) solve that issue by enabling synchronous communication at the local level and asynchronous communication at the global level where more than one local clock cycle is required to transmit information to another location of the chip[20]. This has lead to the paradigm of network-on-chips[21] and has deeply modified the way global communication is performed between the different components of a chip

- In an effort to reduce the wire power consumption and delay, new techniques called 3D stacking[22] are studied to stack several dies and reduce the average wire length. This makes new design questions arise like the partitioning of the different parts of the system over the different layers, the number of layers, their size and so on.

From these two examples it is clear that the technology shift involves many changes at the highest abstraction levels of the design. It makes early design choices very difficult since the system-level designer has to make very basic system partitioning choices whose relevance and performance will however depend on the technology. This system/technology gap is even more difficult to fill due to the specialization of the different people involved in the design: a process guy barely knows about RTL while a system-level engineer would have some pain to figure out what dual damascene is good for. This communication issue is at the centre of a design quality problem: silicon processes need architectural information to understand how they should be optimized and system-level should know about process to understand how we can use them and take benefit from them.
From these two examples of technology shift, we can conclude that early design choices cannot be made anymore without any consideration for silicon process: this exacerbated dependence between very different abstraction levels is not without consequence on the

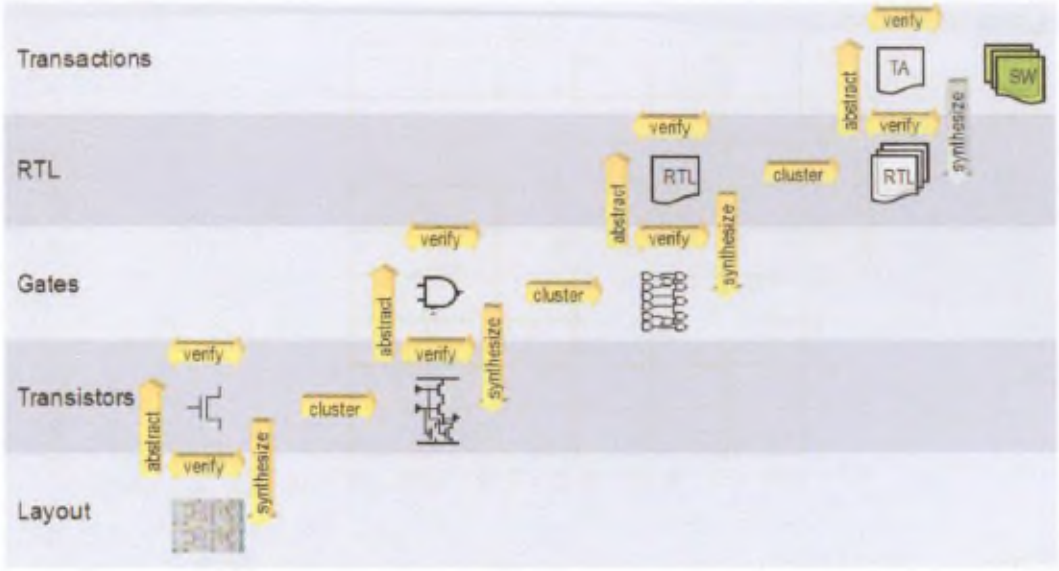

Figure 1.9: Design flow hierarchy and associated design steps[9]

whole design process as we will see in the next section.

1.4 Design: the big picture

Now that we have discussed the design representation, some main characteristics of the design process and some technology issues, it is time to draw the big picture of VLSI design flow. We would like to answer the crucial question of how straightforward it is to move from a very abstracted system-level description to the complete layout of the chip.

1.4.1 From system-level to transistors

As previously mentioned, designing a VLSI system consists in a progressive refinement through the different abstraction levels of an initial pencil and paper description of the functionality and its platform down to the physical implementation of the chip. A clear illustration of this design activity is given in Fig.1.9 where we can see all the most commonly referred abstraction levels from TLM down to layout[4]. This figure highlights that a top-down design methodology always proceeds in the same way to move from one abstraction to the lower one:

1. A synthesis operation is performed: functionality and platform descriptions are refined using primitives of the lower abstraction level

[4]Fig.1.9 represents top-down and bottom-up design flavours: we will only focus in the first one ignoring on this figure the arrows from the bottom to the top of the design hierarchy.

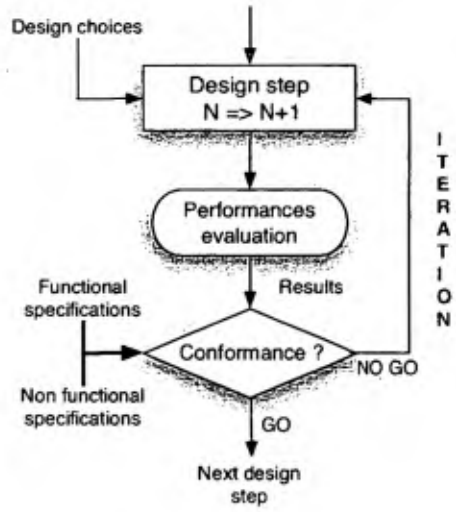

Figure 1.10: Formalization of a generic design step allowing to move from abstraction level N to lower level $N+1$

2. A verification step takes place to make sure that the synthesis has left the functional properties of the system intact.

If this description can be generalized to each design step, the precise design activity is however hidden under the *synthesize* term: its focus may move from functionality to platform and become more and more automated as we go down in the design hierarchy. Based on this simplified representation of design process, we made the hypothesis that each design step moving the system description from an abstraction level N to the lower level $N+1$ can be represented as illustrated in Fig.1.10. We can identify different the following operations:

1. The design step itself consists in refining the description of the functionality and the platform from level $N+1$ to level N and implies different design choices. For instance, we could use different algorithms to define the functionality, define different platforms by changing the communication architecture or the number of processors of a MPSoC platform or modify the way the functionality is mapped onto the platform. All these choices will have an impact on the performances.

2. The performances are evaluated either manually or automatically using for instance simulations tools or thanks to reports generated by synthesis tools

3. The performances are compared to the functional and non-functional specifications and we test if they match. If this is the case, we are able to go on to the next abstraction level. If not, we have to iterate and call some previously made design choices into question until performances match the requirements.

So as we can see, a design flow is something not so straightforward with its iterations that may be required to meet the performance specifications. These iterations are very harmful to the achievement of a project since they add extra design time to the initial planning and may well jeopardize time-to-market estimations. Many factors can lead to these iterations through the design flow: some of the most important reasons are discussed below.

Lack of information At the very beginning of a design, we don't have many information about the system so that it is very difficult to make confident design choices that will meet the requirements. Most of the time, system-level designers take decisions based on their own experience or previous similar designs and set reasonable bounds on timing, power and silicon area that each part of the system should meet. Through the following design steps, more and more information are added to the system description so that it becomes possible to make more accurate estimations of its performances: that's precisely when we may discover that some non-functional requirements cannot be satisfied with the design choices that have been made. Iterations are thus required to remove this performance bottleneck and previous design choices need to be invalidated. Without any guideline rules and understanding of the system, this process could well turn into a blind try-and-test nightmare until we hopefully find a good solution. Iterations are thus intimately linked to the nature of VLSI top-down design flows and result from the truncated and partial knowledge of the system that we have at the very stages of this design.

The system/technology gap The system/technology gap that we have previously described is an iteration prone factor making things even worse. Indeed fractioning the design into a growing number of steps with many new possibilities at each abstraction level makes the number of possible combinations simply tremendous. The tight coupling between architectural choices and technologies (see Sec.1.3) along with the increased specialization of the many people involved in a whole design flow makes design choices more and more difficult to make. Missing the big picture could well lead to suboptimal decisions in terms of performances and eventually entail iterations if the requirements are too tight.

Lack of tool support A last difficulty comes from the lack of tool support to help system-level designers to make their decisions. This is illustrated by Fig.1.11 where is schematically represented the evolution of the tool support for different generations. As we move towards higher abstraction levels and design manipulations more oriented towards methodology, we see that tools offer a weaker support to the user while design steps very close to the implementation benefit from a much larger tool support. As years pass by, the gap is not filled since design begins at more and more abstracted levels while tools cannot keep up with such that pace. As a result, very few tools are available at the top abstraction levels so that system-level designer have to consider and compare all design options almost by hand.

These three previously discussed points are crucial causes to design process iterations and need to be removed in order to improve the quality of design. To do so, we can rely

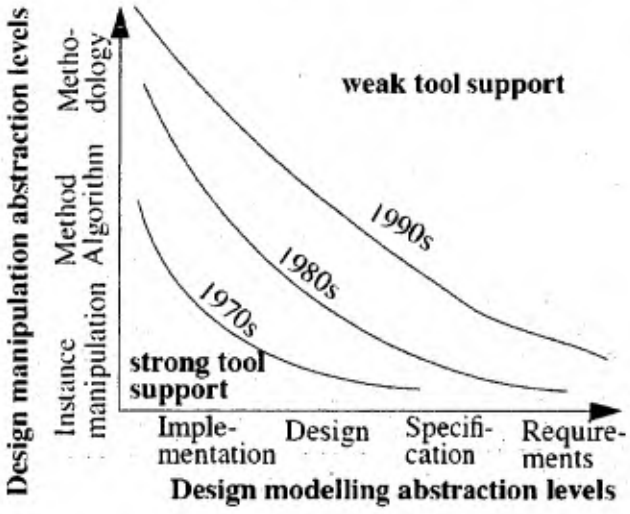

Figure 1.11: Evolution of the EDA tool support over the last thirty years[7]

on a very simple and logical idea: if you have many possible solutions to solve a problem, you should probably first try to evaluate their relative potential outcome to avoid wasting time in successively implementing each of them. And that's what performance prediction is all about: saving time by quantitatively comparing the different design solutions thanks to modeling in the fastest possible way.

1.5 Performance prediction for better design

To find better performance compromises for the final system and avoid iterations as much as possible, we would thus like to be able to estimate performances at the earliest stages of a design flow in order to discard as soon as possible dead-end solutions. Indeed the fundamental choices made during the first steps of the design flow should be the result of objective and quantitative comparisons of the performance attributes of each solution.

Fig.1.12 illustrates how modeling can take part in the classical design process: while the left part of this figure shows one step of the design flow as it was described earlier in Fig.1.10, the right part depicts a *model* of this step trying to reproduce the non-functional performances resulting from it. As it is a model, we expect it to require much less time to estimate than going through the design flow itself so that it becomes possible to explore many more solutions in a same amount of time and therefore perform efficient and larger design space exploration. The whole interest of modeling lies in the ability of making information from lower abstraction levels available for the very first steps of the design flow and enabling early testing and quantification of design choices on performances. Using the results from this performance model exploration, we are then able to feed these information back to the design flow and therefore guarantee better results and avoid as much as possible iterations.

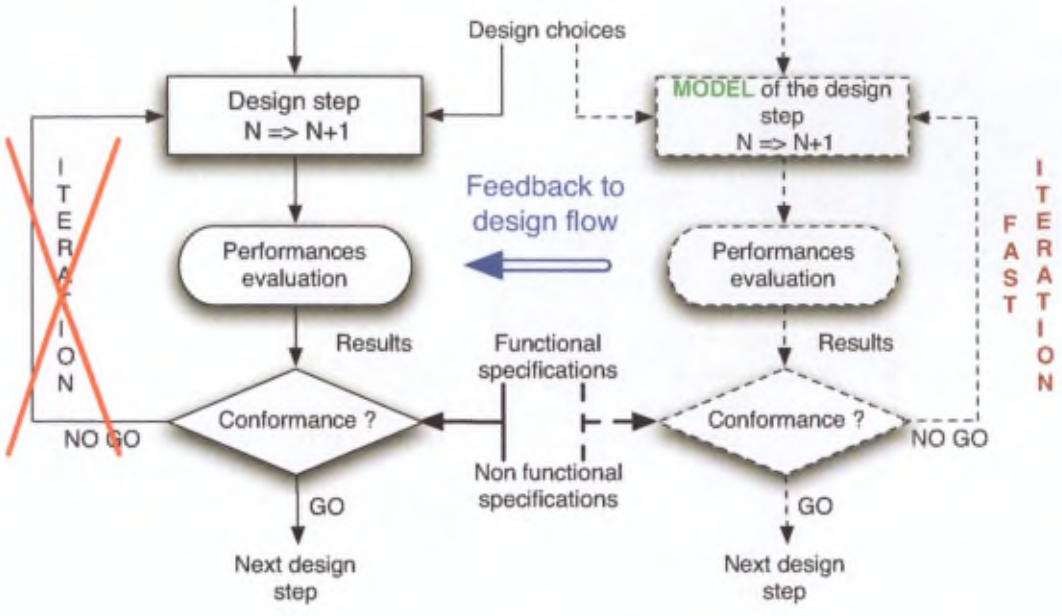

Figure 1.12: Representation of the design exploration

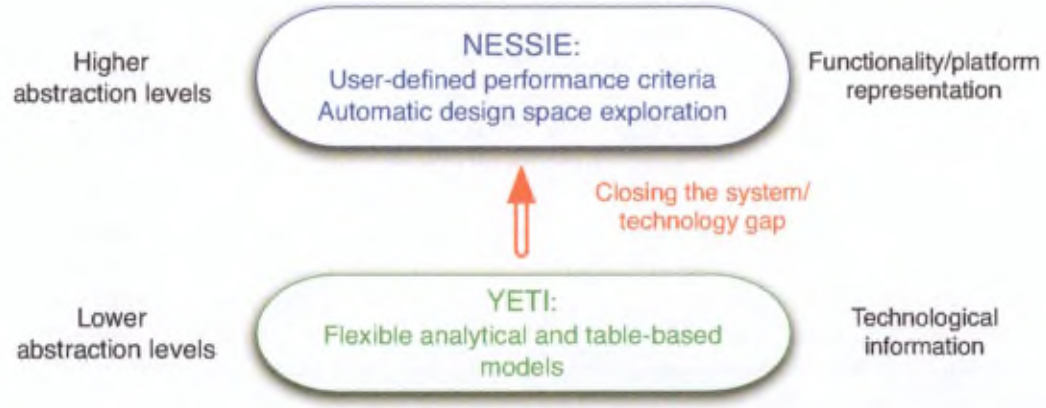

Figure 1.13: Combination of out two tools Yeti and Nessie for performance prediction

To contribute to VLSI design performance prediction, this work proposes an original approach relying on the combination of two different tools called Yeti and Nessie as depicted in Fig.1.13. While Nessie focuses on a detailed description of the system by separating the functionality, the platform and its mapping, Yeti provides Nessie with models to estimate the performances based on lower level parameters. Using this combination, we are able to gather information from very different abstraction levels inside the same framework which highly contributes to fill the system/technology gap.
In this dissertation, we will discuss the state-of-the-art in VLSI performance prediction, detail the principles that we will use to build our tools and propose many case studies to validate our different concepts and demonstrate how we can propose good solutions for design various problems.
We will start in the next chapter with the description of our Yeti framework.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, April 1965. [Online]. Available: http://dx.doi.org/10.1109/JPROC.1998.658762

[2] N. Muralidharan, S. Wunnava, and A. Noel, "The system on chip technology," in *Pro. of second LACCEI International Latin American and Caribbean Conference for Engineering and Technology*, june 2004.

[3] D. D. Gajski, "design methodology for systems-on-chip," in *proceedings of OCCS 2002*, Aug 2002.

[4] F. J. Ramming, *Systematischer Entwurf Digitaler Systeme.* B.G. Teubner, 1989.

[5] D. D. Gajski and R. H. Kuhn, "New vlsi tools," *Computer*, vol. 16, no. 12, pp. 11–14, 1983.

[6] D. Stroobandt, *A Priori Wire Length Estimates for Digital Design.* Boston / Dordrecht / London: Kluwer Academic Publishers, 4 2001.

[7] A. Jantsch, S. Kumar, and A. Hemani, "The rugby model: a conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems," in *DATE '99: Proceedings of the conference on Design, automation and test in Europe.* New York, NY, USA: ACM, 1999, p. 54.

[8] *IMEC Conceives "Double Y" Methodology for Design of Multi-Functional Devices*, 2004.

[9] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EEDesign of EETimes*, February 2002. [Online]. Available: http://www.gigascale.org/pubs/141.html

[10] VSIA, "Vsia system level design model taxonomy document," 2001. [Online]. Available: http://www.vsi.org/documents/vsiadocuments.htm#sld221

[11] I. Panagopoulos, "Models, specification languages and their interrelationship models, specification languages and their interrelationship for system level design," HPCL,The George Washington University, Tech. Rep., 2002. [Online]. Available: http://hpc.gwu.edu/%7Ehpc/iptools/pub.htm

[12] A. B. Kahng, "Futures for dsm physical implementation: Where is the value, and who will pay?" July 2000, 12th DA Show keynote.

[13] O. commitee, "Systemc tlm 2.0 standard," Open SystemC Initiative, Tech. Rep., 2008. [Online]. Available: http://www.systemc.org/downloads/standards/tlm20/

[14] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 1998, pp. 203–211.

[15] T. Sakurai, "Perspective of power-aware electronics," in *proceedings of ISSCC 2003*, February 2003, pp. 26–29.

[16] J. A. Butts and G. S. Sohi, "A static power model for architects," in *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. New York, NY, USA: ACM, 2000, pp. 191–201.

[17] R. Krishnamurthy, A. Alvandpour, S. Mathew, M. Anders, V. De, and S. Borkar, "High–performance, low–power, and leakage–tolerance challenges for sub–70nm microprocessor circuits," *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pp. 315–321, Sept. 2002.

[18] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr 2001.

[19] J. D. Meindl, J. A. Davis, P. Zarkesh-Ha, C. S. Patel, K. P. Martin, and P. A. Kohl, "Interconnect opportunities for gigascale integration," *IBM Journal of Research and Development*, vol. 46, no. 2-3, pp. 245–264, 2002.

[20] S. F. Smith, "The middle path: Globally-asynchronous locally-synchronous (gals) design," in *IEEE Computer Society, Boise Section*. Department of Electrical and Computer Engineering, Boise state university, 2003.

[21] A. Leroy, "Optimizing the on-chip communication architecture of low power systems-on-chip in deep sub-micron technology," Ph.D. dissertation, Université Libre de Bruxelles, 2006.

[22] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3d) microarchitecture," in *MICRO 39: Proceedings of*

the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–479.

[23] "International technology roadmap for semiconductors 2003." [Online]. Available: http://www.itrs.net/

Chapter 2

Yeti: Concepts, Design and Implementation

Abstract

In this chapter, we present Yeti, our C++ library and standalone tool for the flexible representation and execution of analytical and table-based relations. From a review of the state-of-the-art, we first identify a lack of support for such a kind of tool and explain the different limitations of the actual tools. Based on that survey, we define a new model representation structure that will used to build Yeti. Based on this new model representation organized in 3 layers, we propose a lot of new interesting features like easy model/input sensitivity analysis, the possibility of defining scripted simulations, a method to formalize analytical models into hypergraphs extending their use, automatic generation of resulting plots and the use of a strict XML grammar to prevent the user from defining inconsistent input data. Thanks to these features, Yeti removes most of the limitations of state-of-the-art tools in the domain and offer the reliable and efficient mathematical engine that we require to build Nessie upon.

2.1 Introduction

At the beginning of our literature survey, we quickly found several papers revolving around performance prediction among which the very interesting [1] where its author defined the concept of System-level performance modeling as follows:

> System-level performance models can be defined as first order models that attempt to capture the majority of relevant system design issues in order to provide useful predictions or early feedback to designers. As this perfectly fell into our research concern, we gathered several related papers focusing on

very simple systems, tools and simulators relying on very simple models like closed-formed relations to predict the performance of a VLSI system.

At the same moment, we were also looking for a way to provide our C++ framework (that would later become Nessie) with a dynamic mechanism to represent and evaluate analytical relations.

The opportunity of improving the tools related to system-level modeling gave us the idea to extend our initially simple C++ library into a standalone tool providing additional services than the simple representation and evaluation of analytical relations: that's the story of Yeti.

We will first start this chapter with a small state-of-the-art in system-level modeling tools to highlight the different features that they propose and explain why we felt necessary to implement a new tool to replace existing systems.

2.2 State of the art

2.2.1 SUSPENS

SUSPENS (Stanford University System Performance Simulator) has been developed in 1990 by Bakoglu and is one of the earliest modeling tool for system-level performance prediction. This fairly simple framework is based on a set of *eleven* analytical relations to estimate the most important performance factors of a generic CMOS CPU (maximum clock frequency, power dissipation and system size). The model is based on technology, design and packaging parameters and relies on the following assumptions.

- Clock period (T_{clock}) is evaluated using 2.1 where T_{gate} is the average gate delay (based on interconnect load), f_{ld} the logic depth and T_{skew} the clock skew. This delay model may seem quite simple but was sufficient at that time to deliver first order estimations (due to the limited impact of interconnect on stage delay as previously explained in Sec.1.3).
- All wire-length and silicon surface estimations rely on Rent's rule[1].
- Power consumption only takes into account the dynamic contribution.

$$T_{clock} = T_{gate} * f_{ld} + T_{skew} \tag{2.1}$$

Because SUSPENS is quite old now, it ignores many aspects that are now unavoidable in any performance prediction system targeting CPUs (cache memory hierarchy, interconnect layer hierarchy, no throughput evaluation etc.).

[1] Rent's rule is mathematical expression defining the relation between the number of pins of a circuit and its gate count[2]. Since its introduction in 1971, its main application is the evaluation of the wire length distribution on a chip[3].

2.2.2 Sai-Halasz model

In 1995 Sai-Halasz [4] proposed a simulator to project future trends in high-end microprocessors and used it to compare and predict bipolar and CMOS processors evolution. The simulator enables the estimation of the optimal clock frequency based on the choice between a complex uni-processor and multiple processors exchanging information through off-chip communication. Compared to other prediction systems, a particular focus is put on modeling of package delays, via blockage and multi-tier wiring architectures. Based on the cycle delay model, design parameters for "ultimate" CMOS and bipolar processors are determined (number of processors, die size, gate lithography, wiring strategy for each tier, gate oxide thickness and power supply voltage). The simulator also allows the user to perform parameter sensitivity studies and observe the impact on cycle time.

The main relation for clock cycle 2.2 is composed out of the average wire-length stage delay D_{av}, the longest wire stage delay D_{long} and the package delay between two processors D_{pack}. The average wire-length is modeled using a Rent's rule based relation (2.3) where CP is the circuit pitch, FO the average fan-out, IR the Rent exponent[2], NC the circuit count and FF a numerical factor determined by simulation.

$$T_{clock} = 11 * D_{av} + D_{long} + D_{pack} \quad (2.2)$$

$$NL_{av} = FF * CP * (1 + 0.1 * log(NC)) * (1 + 0.3 * (FO - 1)) * NC^{IR-0.5} \quad (2.3)$$

2.2.3 Takahashi model

Takahashi presented in [6] an interesting extension of SUSPENS. Clock skew model has been modified to take buffered clock trees into account and the estimation of the average line length now relies on Davis's expression[7] rather than Donath's[8] that is older and less accurate[5]. The complete model is available to anyone as a java applet: input parameters are defined by the user using the GUI and results are then computed and displayed. The novelty compared to previous tools comes from the fact that model results can be saved through the Internet on a server database and data can afterwards be recalled: this provides users with a simple way to exchange their simulations results and to share knowledge.

2.2.4 RIPE

The Rensselaer Interconnect Performance Estimator (RIPE) is a tool for early prediction of microprocessor performance[9]. Starting from the assumption

[2]The Rent exponent p expresses the wiring topology and is a measure proportional to the interconnect density of the chip[5].

that interconnect will the most limiting factor in increasing chip performance, RIPE gives the designer the opportunity to explore different interconnect design decisions and determine their impact on performances. Based on architectural and technology information (feature size, interconnect materials and dimensions) the wirability[3], performance and power dissipation are estimated. The results show relatively good accuracy (25% on average) for different tests with the IBM, Alpha and Intel processors families. The author points out that chip manufacturers often provide few information about their microprocessor so that parameters values like the gate activity factor required for power consumption estimation and critical path characteristics (number of stages, average fan-out) have to be "guessed". If letting the user define the value of several parameters could at first sight seem a limitation of an integrated prediction system like RIPE, it emphasizes the fact that parameter input sensitivity should be used in order to evaluate how this uncertainty impacts the output parameters values. This particular issue will be further discussed in Sec.2.2.8.

2.2.5 GENESYS

Genesys[10] is a performance prediction tool developed between 1994 and 1998 that particularly learns the lesson from previously designed systems. Focusing on the exploration of technological and architectural design choices, Genesys proposes hard-coded physical and semi-empirical models for ASIC and processor performance estimation. The main outputs of the model are the power dissipation, the throughput, the silicon area and the cycle time based on input information available from the ITRS roadmap[11] and from the literature. The tool can either be used in a text-mode or through a win95 graphical user interface.

The main contribution of Genesys consists in the introduction of a hierarchical model organization in 5 different levels : namely the material, device, circuit, interconnect and architectural layers. Besides lowering the model complexity by distributing partial models among the different corresponding layers, it also allows the user to visualize some useful information relative to intermediate layers that would otherwise be hidden in classical modeling tools. The main features of the layers are the following :

- *Material models* include all the material properties (dielectrics constant, resistivity...) as well as information about silicon properties (carrier concentration, mobility, breakdown field etc.).
- *Device models* take into account many important effects on channel current (field effect on carrier mobility, subthreshold voltage degradation etc.) for bulk Si MOSFET devices.

[3]The wirability of a chip refers to information like the routing ability, metal layer repartition, clock distribution, via blockage ratio etc.

- *Circuit models* are available for static CMOS logic family only and enable the estimation of the delay propagation based on MOSFET switching delay and distributed RC interconnect delay.
- *Interconnect models* include estimation of the wire-length distribution using Rent's Rule, predefined wiring schemes for the user to choose among (optimized single driver, optimal cascaded driver, optimal repeater insertion), router efficiency estimation and inter-level wiring blockage ratio.
- *System architecture models* mainly consist in the definition of the critical path (logic depth and region of synchrony) and a Cycle per Instruction model (based on pipeline depth, stalls per executed instruction, superscalar properties and instruction latency).

2.2.6 Codrescu model

Codrescu published in 1999 a paper[12] exploring the microprocessor design space for 100nm microprocessors to find architectural candidates to keep up with Moore's Law for the coming years. The author sweeps the architectural space from one big superscalar processor to a network of 256 simple parallel processors while keeping for each solution the total silicon area constant. Performance is reflected by total computing power expressed in $Mops$ (Eq.2.4) where IPC is the number of issued instructions per clock cycle, $Frequency_{clock}$ the clock frequency and $Speedup$ the multiplying performance factor accounting for the presence of multiple processors. While IPC is based on empirical data, clock frequency estimation (Eq.2.5) relies on technological models for gate D_{gate} (using GENESYS, see 2.2.5) and interconnect delay of the longest possible wire (manhattan distance) D_{wire}.

In opposition to many other prediction performance systems, Codrescu not only takes platform related information into account but also includes a software dependency thanks to the thread-level parallelism present in the application expressed by $Parallel\_fraction$. Using this information combined with the number of available computing nodes $\#nodes$, Amdhal's law (Eq.2.6) is used to compute the $Speedup$ ranging from 1 for single-threaded application to $\#nodes$ for fully parallel applications.

$$Performance = Frequency_{clock} * IPC * Speedup \tag{2.4}$$

$$Frequency_{clock} = 10 * D_{gate} + D_{wire} \tag{2.5}$$

$$Speedup = \frac{1}{\frac{Parallel\_fraction}{\#nodes} + (1 - Parallel\_fraction)} \tag{2.6}$$

The paper concludes that only massively parallel processor architectures coupled to multi-threaded software will be able to sustain the increasing demands in computation performance imposed by Moore's Law.

2.2.7 BACPAC

BACPAC[13] stands for *Berkeley Advanced Chip Performance Calculator* and is probably one of the most advanced system-level performance prediction tool available yet. To allow the user to explore design choices and experiment *what if* predictions, BACPAC proposes to predict maximum clock frequency, silicon area, power consumption (including dynamic, leakage and short-cirucuit power), yield and chip wirability. Compared to other tools, it mainly focuses on DSM issues (leakage current, noise due to Miller effect) to maintain a good level of accuracy even in leading-edge process technologies.

One of the most interesting capability of BACPAC is to consider both local and global interconnect levels while other tools usually focus on only one. Modules of 50k-100k gates contain all the local interconnects wires and are interconnected by global wires to communicate between each other: both levels use Donath's wire distribution. The tool also integrates many concepts from [14] where the total power consumption of a system is evaluated using a hierarchical approach: each component is modeled separately and are afterwards gathered to form the entire system. BACPAC also performs built-in optimization of the driver sizing for optimal interconnect delay and gate sizing for inter-module area minimization, I/O pad drivers optimization by using optimal cascade drivers etc.

BACPAC models are very complete and take into account a large part of previous modeling efforts available through the literature: therefore the information-seeking reader should refer to BACPAC website[15] providing a lot more details about the model and an online executable version of BACPAC.

2.2.8 Summary

In this section we presented several performance prediction tools mostly relying on closed-formed models: they were selected and extracted from the literature according to their relevance and their care for generality. Table 2.1 summarizes the most important features of each tool: besides the output and input parameters, we also included information about the model nature and tool availability but also one example for the critical path. Input parameters are classified in general categories: *technology* covers all the process, material, device and interconnect sizing information, *design* includes all the design aspects (non related to the structure of the chip) that may have any influence on the performances, *architecture* include all the structural information at the chip level (starting at circuit level) while *system* parameters account for structural information outside the bounds of the chip (packaging, MPSoC, SiP information etc.). It may seem surprising that our input parameter classification only consists in four levels while GENESYS uses one more level without taking design into account. This choice has been made on purpose to avoid a certain

redundancy between some GENESYS hierarchical levels and define a mutually exclusive criteria based classification.

Looking back to table 2.1, we can draw the following conclusions:

- All the models use at least technology related information. The main idea behind that is to provide the user with a model capturing the impact of low-level design choices on various performance metrics to enable design space exploration and bridge the "design-technology gap" (see Sec.1.2.2). However very few tools offer the ability to exploit underlying models in another way than a succession of *what if* experiments.
- Many of the critical path models are not completely specified: the number of stages for instance has to be set by the user. The reason for that is explained in [13]: "The critical path logic depth is also user-defined. This is important as it can vary widely from company to company and design to design."

  Since all the systems use different models for the critical path representation, it is very difficult to compare the produced results. However critical path is just an example amongst many others: interconnect, device ,wire distribution models may also vary significantly from one author to another. In such a context no wonder that results differ a lot and are therefore difficult to compare emphasizing the need for input and model sensitivity analysis.
- Most of the underlying models rely on analytical relations but often lack all the information needed to reproduce the experiments.
- Each reviewed tool predicts at least the maximum clock frequency: Sai-Halasz even uses the term *performance* to refer to clock frequency only.

Considering the hard-coded nature of certain tools, the redundant effort spent on building models and the difficulty to compare predicted results due to different model assumptions, the GSRC decided to build a common platform for the specification and execution of system-level models: the GTX framework.

2.3 GTX, the ultimate prediction tool?

Introduction

GTX[16] stands for MARCO GSRC Technology Extrapolation system and has been developed in 1999 in a joint effort by the GSRC group and the MARCO team to cope with all the limitations of previous closed-formed based performance prediction tools. As established in section 2.2.8, most of the earlier attempts to implement this kind of tool unfortunately lead to :

- Incomparable results due to different modeling assumptions

tool	input parameters	output parameters	model nature	availability	critical path
SUSPENS	technology, design, system	power, clock frequency, chip area	analytical relations	X	user defined
Sai-Halasz	technology, architecture, system	clock frequency	simulator	X	11 stages, global wire delay, packaging delay
Takahashi	technology, design, system	power, clock frequency, chip area	analytical relations	java applet	user defined
RIPE	technology, architecture	clock frequency, power, wirability	simulator	X	user defined
Codrescu	technology, architecture, system	clock frequency, CPI	analytical relations and tables	X	11 stages
GENESYS	technology, architecture, design	clock frequency, power, throughput, chip area	analytical relations	WIN95 interface	user-defined
BACPAC	technology, architecture	clock frequency, yield, chip area, power, wirability	analytical relations	web interface	15 stages (by default)

Table 2.1: Summary of state-of-the-art prediction tools

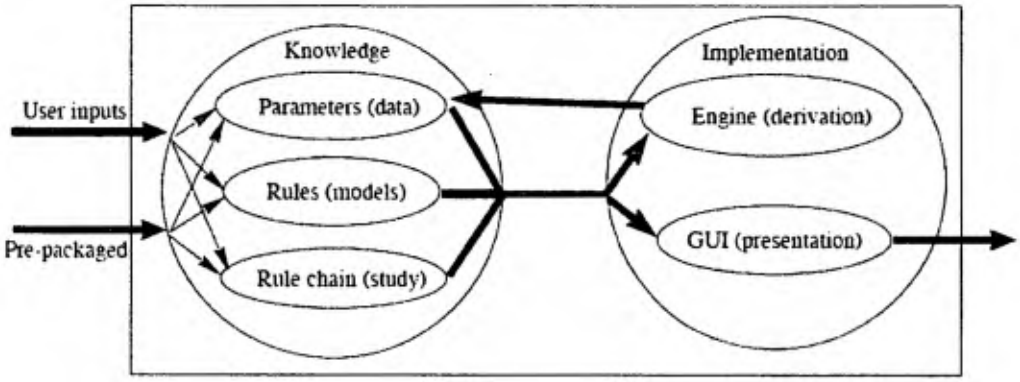

Figure 2.1: GTX internal structure[16]

- Hard-coded model-based tools limiting the flexibility in modeling
- A large redundancy in the effort to develop all these models

GTX copes with all these limitations by "providing an open, portable *framework* for specification and comparison of alternative modeling choices"[16]. In other words the tool doesn't really propose new models but rather provides the user with a way to integrate models from different horizons into a same working environment to compare and execute them more easily. This common modeling platform was also meant to be an opportunity for different research teams across the world to exchange their models on the web and share their knowledge. However GTX never got the success it deserves and it seems that the development stopped a few years ago.

Let us now review in details the features of GTX and what makes it so different from all ever developed performance prediction tools.

Structure

The most fundamental design decision in GTX is to separate model specification (knowledge) from the derivation engine (implementation and execution). This separation allows the user to enter models of its own and use loadable/savable model libraries based on a GTX-defined "human-readable ASCII grammar". The GTX structure is presented in Fig.2.1 and is composed out of 3 different kind of objects:

- *Parameters* are the basic data on which models operate. A parameter has several attributes among which a unique name, a data type, units and a

default value.

- *Rules* operate on data and tell the derivation engine how to compute the value of one unique output parameter given a set of input parameter values. Different types of rules are available: *ASCII rules* (closed-formed expressions, lookup tables, if-then-else structures), *external executable rules* (based on dynamically invoked PERL scripts) or *code rules* (directly integrated into GTX source code hence requiring recompilation).
- *Rules chains* are just a list of rules to form a more extended and complex model. The order of rule evaluation is determined by GTX to turn individual rules into acyclic graphs to avoid infinite loop execution. To make sure that this condition is fulfilled, each output parameter can only be used as input parameter of at most one other rule so that the resultung graph is a simple tree.

Interface

GTX comes as a multi-platform graphical user interface and requires all the data to be entered in the interface. The user is able to define parameters, rules and rules chains. Input values can then be specified along with the type of study (single value, sweep simulations) and results can then be plotted. One last interesting feature concerns the ability of adding constraints to the evaluation: the user can fix a maximum or minimum value for the output parameter. The solutions where values don't fit within the constraints are removed from the final result set.

How interesting and innovating GTX may seem compared to previous tools, it suffers from some annoying limitations:

- As explained above, each rule is defined so that one particular parameter is meant to be the output, the other ones implicitly becoming inputs: GTX then automatically composes the resulting rule chain. However depending on the context of use, we may want to turn some inputs of the model into outputs and inversely.

  For instance, all the microprocessor performance evaluation systems presented in Sec.2.2 estimate the clock frequency based on technological parameters. However it could sometimes be much more useful for a designer to express that same clock frequency as a model input rather than an output so that he can fix constraints on its value and explore the technological parameters values set that fits the requirements.
- Input parameter sensitivity is supported in GTX at the expense of evaluating several times the same rule chain while varying each input parameter value around its nominal value. Its leads to multiply the number of experiments hence the time required to carry them out. More efficient

solutions should be provided to avoid exploding exploration times with the scaling up of model complexity and input parameters number.

- GTX relies on ASCII textual files to specify rules and their meta-information. Some mechanisms to manage parameter/rule name uniqueness were never implemented and probably require a much stronger and more strict grammar.
- Model sensitivity studies require to replace some rules by others: all these operations have to be successively done by hand using the graphical interface. Large campaign simulations can quickly become time-consuming due to manual intervention: a way of automating all these actions could surely relieve the user from repeating those tedious operations again and again.

For our own usage, we decided to implement a C++ library featuring model execution and specification abilities and by the way tried to tackle all the previously mentioned limitations of GTX by respectively:

- Introducing a three-level hierarchical model description to remove model specification restrictions due to fixed input/output parameter formulation and extend model usage both at rules and rule chains levels
- Setting up a mechanism to support fast extrema evaluation by using a tree representation
- Using an XML based grammar to offer automatic verification and powerful element constraint mechanisms
- Providing the user with powerful scripting possibilities.

2.4 Concepts for advanced modeling

In the previous section, we presented GTX, the state of the art performance prediction tool and highlighted its weaknesses and limitations. To cope with these issues, we decided to implement our own tool named $YETi^3$ (or YETI to make it short) which stands for "Yet anothEr Tool for the representatIon of analytIcal relatIons". The tool is basically divided into three different hierarchical levels as represented in Fig.2.2. The levels are from the lowest to the highest:

- The *generic rule* level is the model evaluation core providing the framework with the fundamental mathematical and algorithmic mechanisms
- The *relation* level enables model reuse by introducing the notion of *model flexibility* that will be discussed later
- The *behaviour* level that establishes a convenient interface to access all the services of the underlying including evaluation and the ability to reuse the same model for different problems

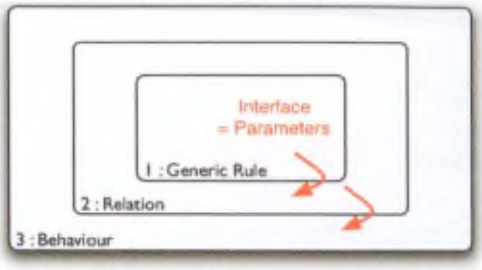

Figure 2.2: The 3-level hierarchy inside $YETi^3$ using *parameters* to communicate data between each other

Before further explaining these three different levels, we start with the *parameter* element that is the bond between generic rules, relations and behaviours.

2.4.1 Parameters

Parameters allow the different hierarchical levels to exchange data and communicate between each other. Therefore they represent any element that may contain a value: technology, application, system-related variables that are quantifiable fall into that category. A parameter has the following properties:

- A *name* that is unique among all relations and behaviours in order to enable the distinction between two different parameters
- A *floating point value*
- *Constraints* that define a lower and higher bound value for the parameter
- Parameters also have no special *orientation* meaning that on the contrary of GTX, they are not fixed inputs or outputs of a model but can change their nature depending on the context of model use (see 2.4.3 for more details).

2.4.2 Generic rules

Generic rules are at the centre of model evaluation and value calculation. Indeed they allow the user to tell $YETi^3$ how to estimate the value of one parameter (output parameter) based on the value of the other parameters (inputs parameters). There is no limitation on the method used to evaluate the output: iterative methods, algorithms, lookup tables, closed-formed expressions etc. Only the executability of the rule must be ensured: that's why we called them *generic* rules. If the number of inputs is not restricted, generic rules may only evaluate the value of one sole output parameter.

The implementation requirements for our generic rules are the following:

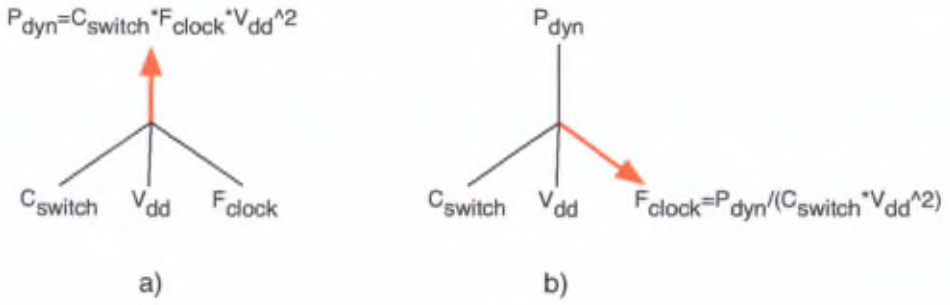

Figure 2.3: Two different generic rules for the same relation

- Dynamically built rules to allow the user to perform model sensitivity analysis by exchanging models at run-time (without requiring the entire code to be recompiled each time a model is added)
- Implementation of the basic analytical rules and table-based rules while offering easy extension possibilities of generic rules types for the future
- Special support for constraints estimation to enable easy and fast input sensitivity analysis
- Efficient generic rules to provide $YETi^3$ with a fast computing engine

To further explain further the important concepts inside $YETi^3$, let us introduce a guiding example that will be used through the whole chapter. We chose the well-known model for P_{dyn} dynamic power consumption (equation 2.7) based on C_{switch} the total capacitance to switch, V_{dd} the supply voltage and f_{clock} the operational clock frequency.

$$P_{dyn} = C_{switch} * f_{clock} * V_{dd}^2 \tag{2.7}$$

2.4.3 Relations

A relation is a link between a set of n defined parameters: it expresses that they are interrelated in such a way that fixing the value of any set of $n-1$ parameters determines the value of the remaining parameter. If a relation implies a dependence between all the contained parameters, it doesn't specify how each particular parameter can be evaluated based on the others: that's precisely why generic rules are encapsulated into relations. Each parameter can be *associated* with at most one generic rule to tell $YETi^3$ how to estimate it.

The *relation* layer is responsible for adding inside $YETi^3$ most of the flexibility that other performance prediction tools usually lack. To illustrate that let us

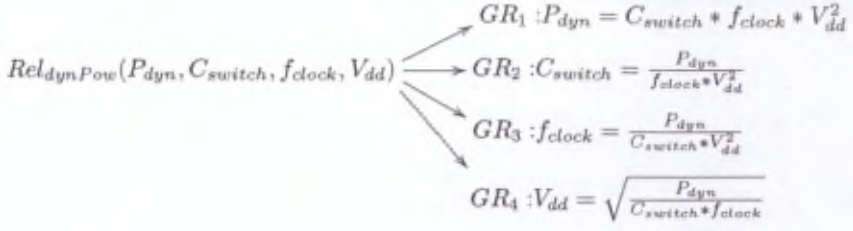

Figure 2.4: The definition of the dynamic power consumption relation with its four associated generic rules

have a look back at Eq.2.7 defining the dynamic power consumption. This expression is very convenient when it comes to evaluate the power consumption based on the other parameters (left side of Fig.2.3) but what if a designer would rather like to fix a certain power consumption and see the impact of his choice on the maximum clock frequency? Using the same orientation as expression 2.7, he should probably set the values of C_{switch} and V_{dd} and then perform a sweep on the f_{clock} values until he reaches the desired P_{dyn}. This method is however as absurd as it is time-consuming when we have a closed-formed expression at disposal. Indeed a much more straight-forward manner would be to *reverse* the direction of the model so that P_{dyn} explicitly becomes an input rather than an output and inversely for f_{clock} (right side of Fig.2.3). Relations precisely represent this notion of *model reversibility* by allowing each parameter to become the output thanks to the use of parameter specific generic rules (illustrated for our case study in Fig.2.4). This feature increases the scope of use of a model while keeping the complexity hidden inside the *relation* element and adds a lot of flexibility compared to other existing performance prediction tools.

For this hierarchical level to be implemented, we only require some association mechanisms and the possibility to integrate heterogeneous generic rules types inside the same relation.

2.4.4 Behaviours

Behaviours offer the support for the representation of complex models based on a collection of relations. A behaviour doesn't allow the user to see the encapsulated relations but only provides him with an access to the inputs and outputs of the modeled system. The way the model is structured remains hidden to the user inside a black-box so that only the *behaviour* can be observed from the outer (hence the name).

Let us illustrate that by taking back our previous dynamic power consumption

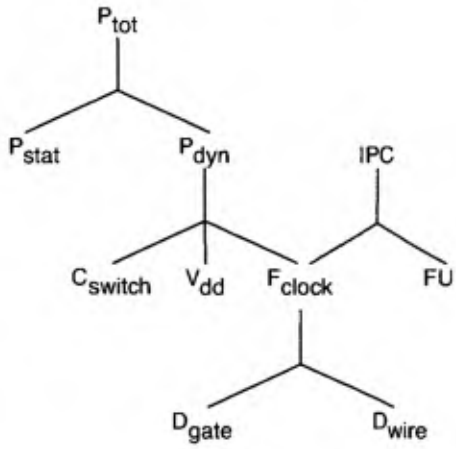

Figure 2.5: One example of behaviour composed out of 4 different relations without orientation

relation example of section2.4.3. This single relation is not of great use on its own but can be combined to other relations to compose a complete system. Therefore we add the following relations to it to compose a very simple microprocessor model example:

- A relation linking P_{tot} the total power consumption, P_{dyn} the dynamic power consumption and P_{dyn} the static power consumption
- A relation linking IPC the number of instructions, f_{clock} the operational clock frequency and FU the number of functional units of the microprocessor
- A last relation linking f_{clock}, D_{gate} the gate delay and D_{wire} the wire delay.

Fig.2.5 illustrates this behaviour composed out 4 non-oriented relations. Each relation shares common parameters with the others allowing them to communicate data by forming a complete graph. However we need to determine which parameters are inputs or outputs before being able to execute the behaviour.

The left part of Fig.2.6 shows the previous behaviour where all relations are associated to a specific output parameter. The resulting behaviour is a system model estimating the output parameters (in green) P_{tot} and IPC based on the input parameters (in red) P_{stat}, C_{switch}, V_{dd}, D_{gate}, D_{wire} and FU. As we can see we obtain a multiple inputs multiple outputs model (contrarily to generic rules that only permit a single output parameter) that has a given orientation for each relation.

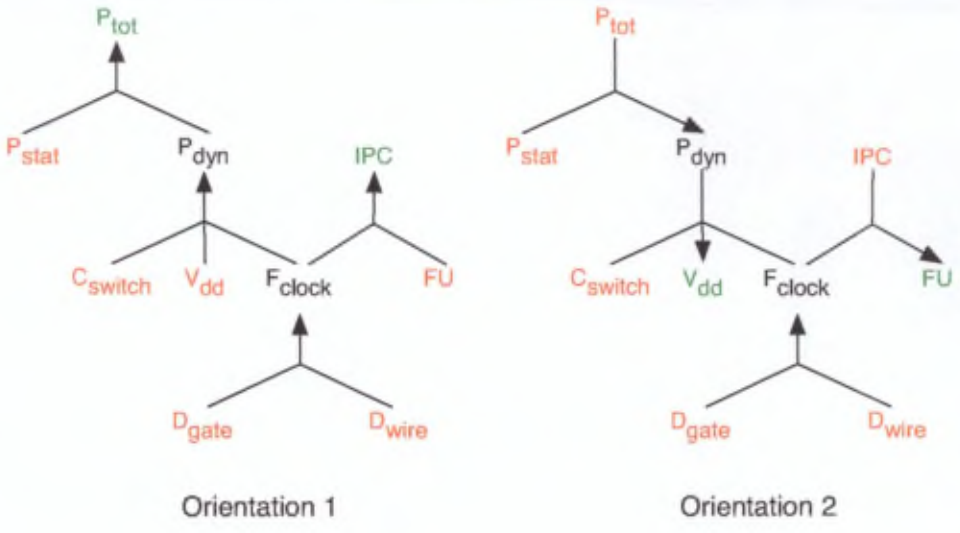

Figure 2.6: Two behaviours differing from each other due to separate relation orientations: input parameters are drawn in red while output parameters are in green

However if this model is interesting on its own, it only answers some of the questions a designer could have. For instance, observing the impact of design parameters on the number of instructions per cycle and total power consumption is easy but what if both these parameters are constrained by the requirements? In this last case, the expression of the model doesn't fit the needs of the designer as he will have to sweep all the possible input values to find the desired output values. Behvaiours offer instead a very simple solution: by reassociating the relations, we can obtain a different set of inputs and outputs. The right part of Fig.2.6 shows a possible behaviour orientation where IPC and P_{tot} are now inputs while FU and V_{dd} become outputs instead. It means that the designer has now the opportunity to set values for the IPC and P_{tot} and see which impact these requirements have on the design parameters.

So behaviours are not just multiple outputs executable relations but they take modeling to a higher level by allowing the reuse of the same knowledge captured by models for different problems without any additional effort. Some questions however remain unanswered:

- Are all the possible relation orientations leading to executable graphs?
- How to generate graphs starting from non-oriented relations?
- How to evaluate the output values of these graphs?

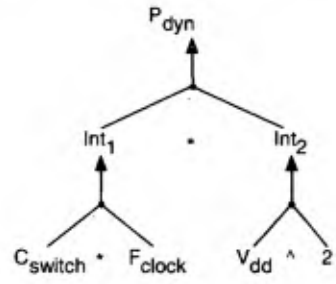

Figure 2.7: A simple hypertree representing the expression $P_{dyn} = C_{switch} * f_{clock} * V_{dd}^2$

The next section deals with all these algorithmic questions related to our 3-layered structured framework.

2.5 Algorithmic and advanced concepts in $YETi^3$

2.5.1 Generic Rules

$YETi^3$ supports at the moment two different types of generic rules namely *analytical* and *table* rules. Because of their specificity and their different scope of use, they deserve in-depth study.

Analytical Rules

Analytical rules allow the user to define and estimate the result of closed-formed expressions. As we want $YETi^3$ to be able to load and build models dynamically, it is very important to find a proper representation to support this functionality without adding too much overhead in terms of execution time. Therefore we chose a tree representation that has the advantage to be scalable at will, easy to execute and fairly simple to build at run-time. Taking back our guiding example of the dynamic power consumption (Eq.2.7), it can be represented by the tree illustrated in Fig.2.7 where the initial expression is decomposed into smaller mathematical operations that we call *basic operations*. Nodes represent parameters while hyper-edges represent basic operations being directed towards the output parameter resulting from this particular basic operation.

Evaluation Analytical trees are very easy to evaluate using a *depth-first* algorithm (see Algo.1). In a few words, this recursive algorithm first explores as deep as possible the tree until leaves are found: when all the nodes of one

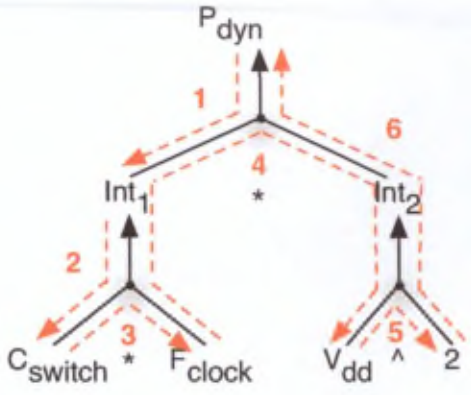

Figure 2.8: Evaluation of the dynamic power consumption tree using a depth-first algorithm

hyper-edge have been explored, the basic operation can then be executed and the result propagated towards the upper part of the tree. Fig.2.8 illustrates this algorithm on our dynamic power consumption tree example: starting from the root node, the tree is explored in the order of the numbered directed arrows until the resulting value is finally returned to the root after six steps (three function calls).

Algorithm 1 The recursive Depth-First Algorithm used to evaluate a tree representation with a first call to $DepthFirst(RootNode)$

```
1: if current node has children then
2:    Depth First(leftNode)
3:    Depth First(rightNode)
4: else
5:    return node value
6: end if
7: Calculate rootNode value based on leftNode and rightNode value
```

Constraints Constraints are part of parameters attributes and are composed out of a lower and an upper bound. They enable input sensitivity analysis by giving the user the opportunity to specify a certain range of uncertainty in parameters input value. There are several practical cases where such a type of study can be useful: for instance, testing the robustness of a model outputs to possible input variations (process variations for instance) or representing the output value uncertainty due to a lack of information in the model (for instance the number of stages in the critical path, see 2.2.8).

Our tree representation allows us to deal, in some some situations, more effi-

ciently with output constraints evaluation than usual tools do. To explain the problem, let us consider a function f representing a N input parameters model with their respective constraints (Eq.2.8).

$$\begin{aligned} & f(X_1, \ldots, X_{N-1}, X_N) \\ & Constraints_{X_i} = [X_{i,min}, X_{i,max}] \quad 1 \leq i \leq N \end{aligned} \tag{2.8}$$

For this function we can compute the partial derivative (Eq.2.9) for all input parameters that represent the sensitivity of the function for each particular parameter.

$$Sensitivity_{X_i} = \frac{d}{dX_i} f(X_1, \ldots, X_{N-1}, X_N) \quad 1 \leq i \leq N \tag{2.9}$$

When the function derivative for one parameter (all the other having a fixed value) has a lot a zeros inside the constraints range, it means that the function has a lot of local extrema. In this situation, finding function global extrema can be very difficult and the use of numeric analysis methods is the only way out.

However, if we face a strictly increasing or decreasing monotonous function for a certain parameter (at least within the constraints range), the search for extrema becomes much easier. Indeed we know that, for an increasing (decreasing) monotonous function, the lower bound of the parameter will result in the minimum (maximum) function value while the upper bound will result in the maximum (minimum) function value. For instance, let us consider the simple two parameters function $C(A, B)$ described by Eq.2.10.

$$C(A, B) = A + B \tag{2.10}$$

If we respectively set $[A_{min}, A_{max}]$ and $[B_{min}, B_{max}]$ constraints values for parameters A and B, it is easy to find the extrema of the function. Since operator $+$ is an increasing monotonous function for both parameters, we only need to look at the extremum constraints values to find the function global extrema (Eq.2.11). Rather than sweeping all the possible input values we are able to find the function extremum values by doing one single evaluation based on the knowledge of the operator properties.

$$\begin{aligned} C_{min} &= A_{min} + B_{min} \\ C_{max} &= A_{max} + B_{max} \end{aligned} \tag{2.11}$$

On this example, this property may seem obvious but it is possible to extend it to a whole hypertree composed out of closed-formed expressions. Indeed we know that each hyper-edge represents a basic operation and each one of them is a monotonous function of their input parameters (Table A.1 in appendix A). So for each hyper-edge, we are able to calculate in one sole evaluation the root

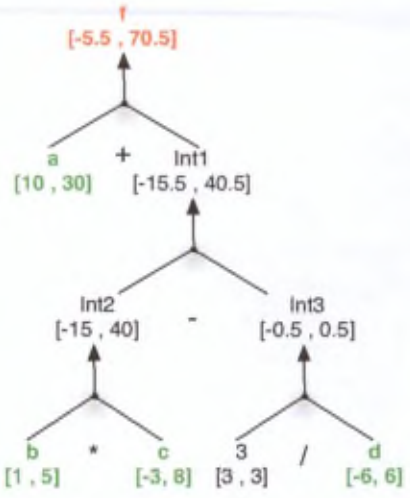

Figure 2.9: Evaluation of the constraints for an hypertree representing closed-formed expressions

node constraints if we know the constraints of the input node (or nodes if the current basic operation is binary). Beginning with the leaves of the tree (whose constraints values are the only one available at this point), we can propagate the constraints values from the bottom to the top of the tree using again the depth-first algorithm (see Algo.1). An example is provided in Fig.2.9 where we can see how the expression is evaluated based on input parameters constraints values (input constraints are in green while output constraints are in red).

Using this method, we only need one single tree evaluation to get the output constraints while usual methods would require much more time to find the same solution. There are however some limitations where it becomes impossible to use this technique. Fig.2.10 shows the tree representation for expression $D = (A + B) * (C - A)$. As we can see, parameter A appears two times in different hyper-edges: as a consequence, D_{max} would be computed using A_{max} for the left hyper-edge (to maximize $A + B$) and A_{min} for the left hyper-edge (to maximize $C - A$). This makes obviously no sense since A may not have more than one value simultaneously: this problem comes from the fact that A appears more than one time in our expression. Indeed we are not able anymore to tell in such a situation if the function is still monotonous for this parameter hence the method could not work anymore. So the only way to guarantee this previous condition is to make sure that the parameter appears only once in the function.

These limitations could seem annoying but our constraints evaluation mechanism takes the most out of the closed-formed expression without requiring any intervention of numerical solver. For relations involving parameters that

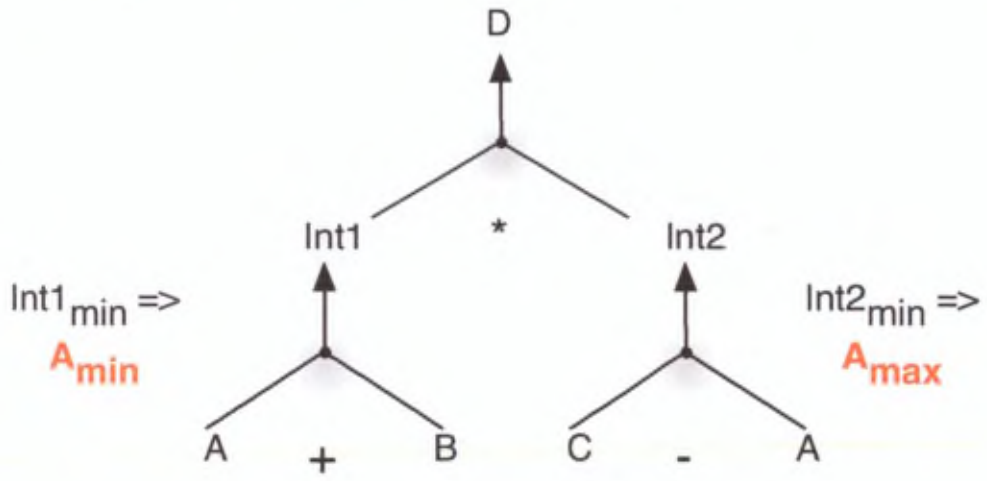

Figure 2.10: Case where constraints evaluation cannot be performed on our hyper-tree: the same input parameter A takes different values

don't fulfill the previous condition, we can use other methods while maintaining our method for eligible parameters: mixing both methods thus accelerates the constraints evaluation time as fast as it could be.

Table rules

Sometimes analytical rules are not suitable for model representation. This is particularly true for functions where we only have a few points at disposal and not a closed-formed expression or even for a list of values where continuous input values don't make sense. For these situations we have introduced *table rules* allowing the user to capture the model knowledge into n-dimensional tables dynamically built at run-time.

Structure Representing a multi-dimensional table is very simple as long as the number of dimensions n is fixed (or known before the building of the table): when this information becomes only available at run-time, it becomes however much more painful. Apart from providing tables with an efficient data structure support (which will be discussed later), we need to find a memory model that enables both fast table search and efficient memory occupation. To justify our final choice we need to anticipate a little bit on the implementation part. Several methods usually exist in C++ Object-Oriented Programming to represent tables:

- *Statically allocated tables* have memory patterns that are completely specified at compiling-time. This type is of course completely useless in our case
- *Dynamically allocated tables* are usually defined using the memory pointer paradigm allowing the programmer to delay the table length choice until run-time. If this mechanism is very powerful, it has the disadvantage of being error-prone by exposing the memory addressing mechanism to the user. Furthermore, if the length of tables is run-time defined, the number of table dimensions still remains a compiling-time choice. Once the table is built, the programmer has no possibility to get the length unless he decided to save it into a special value.
- *Library-based tables* are defined in standard packages like STL to provide the programmer with easy-to-use data structures coming with searching and manipulation features. If the user is now able to dynamically specify the length of the table and retrieve this information afterwards, it isn't still possible to specify the number of dimensions at run-time.

Since no standard library gave us a satisfying solution that fits our requirements, we had to build a data structure of our own. Two different solutions were considered.

First it is possible to use an object-oriented approach to represent a dynamic N-dimensional table. The most obvious solution is to choose a recursive class description where each nested level represents an additional table dimension. Each instance of the table class contains either a dynamic vector of table class instances (except for the class instances related to dimension N) or a vector of values (for the last dimension). Fig.2.11 shows a three-dimensional table built on this concept where dimensions 1, 2 and 3 are respectively 3, 2 and 3 elements long. As we can see each additional dimension requires an exponentially growing number of classes instances (with a complexity of $O(L_{table,tot}/L_{dim,N})$ where $L_{table,tot}$ is the total number of elements contained in the table and $L_{dim,i}$ the number of elements in dimension i). This increasing number of class instances is very annoying as it uses very poorly the memory by adding a lot of class information overhead and lowers the estimation speed due to the numerous class methods calls and inter-class communication mechanisms. Another solution has to be used to answer those issues.

Second we could use one single class to encapsulate the complete data structure since we only need evaluation and building abilities. The structure implementing a N-dimensional table consists in a set of N vector called *dimensional array* (containing in increasing order all the possible input parameter values for each dimension) and one dynamically allocated array of floating point values called the *value array* (holding the table output values). The trick is to gather consecutively all the data into one single array rather than considering a hierarchical data organization that a table would suggest. The structure is

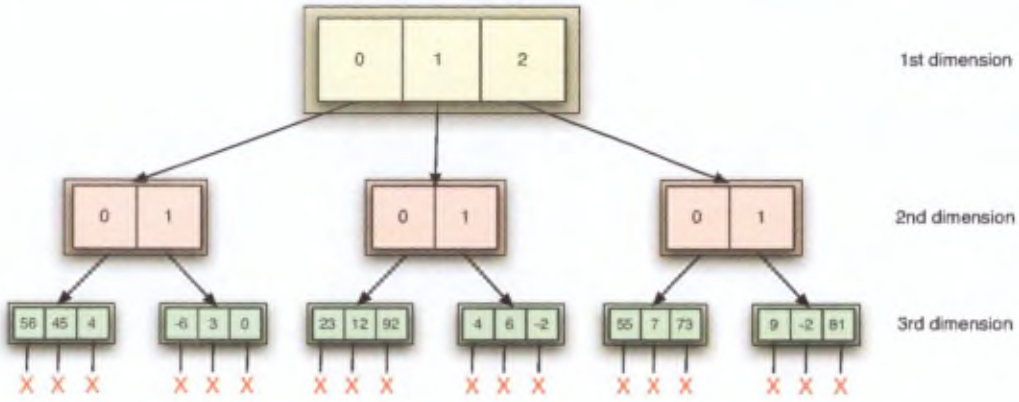

Figure 2.11: Representation of a 3-dimensinal table where dimensions 1, 2 and 3 are respectively 3, 2 and 3 elements long

very simple, introduces almost no overhead in memorizing the data but has the only drawback to occupy a lot of consecutive memory space.

Fig.2.12 shows for the previous example (Fig.2.11) the *value array* and the three *dimensional array* associating each index value with a floating point value. As we can see, the 18 table values are the same in the two representations but how is it possible for the latest to access the elements values based on the index? The next part answers that question.

Evaluation The evaluation of a table consists in retrieving the table values based on the input parameter values of each of the Nth dimension: therefore we need to calculate the *value table* index associated with the value we are looking for. We know that a fully populated N-dimensional table is converted into a *value array* containing $L_{table,tot}$ elements (Eq.2.12) where $L_{dim,i}$ is the number of elements for dimension i. In our example, it gives us $3 * 2 * 3 = 18$ elements which can be seen in Fig.2.12).

To get the *table array* index, we proceed in two steps:

- For each of the N input parameter of the table, we need to search the corresponding *dimensional array* to get the *dimension index* $Index_{dim,i}$ for each one. Several methods are possible but since tables may contain a large number of elements, a simple search starting from the first to the last until a match is found is far too time-consuming. Therefore we took advantage of the fact that values are ordered by increasing values inside the *dimensional tables* and used a dichotomy based search. This method requires at most $\lceil \log_2 L_{dim,i} \rceil$ search steps to get the correct dimension index value.

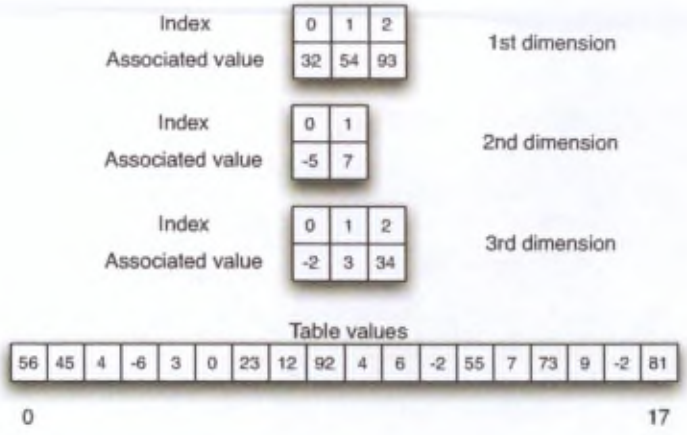

Figure 2.12: Example of a 3-dimensional table using a single *value array* to contain table values and 3 dimensional arrays associating the index with the possible table input values

- Once we retrieved all the index value $Index_{dim,i}$ for each *dimensional array*, we can easily calculate the index of the *table array* to get the output value of the table. To do so, we can use Eq.2.13 consisting in a sum of N products which introduces a very low overhead compared to the previous solution where $O(L_{table,tot}/L_{dim,N})$ method class calls were required.

The way top calculate the index of our current table is represented in Fig.2.13 which also particularizes Eq.2.13 for our three dimensional case. Let us imagine that we would like to access the value associated with the $54, -5, 34$ input values triplet: from Fig.2.12 and its three dimensional arrays, we can see that the input values respectively corresponds to index values 1, 0 and 2. Injecting these index values into our *value array* formula, we have to perform the following operation: $2 + 0 * 3 + 1 * 3 * 2 = 12$. As a result we know that we can find our final table output value in the value array position defined by the 12th index which gives us 92. This can be verified by taking back our initial representation of the table in Fig.2.11 and following for each dimension the arrow respectively associated with 1,0 and 2 index value until we find back the resulting value 92.

$$Number_{Elements,tot} = \prod_{i=1}^{N} Number_{Elements,Dim_i} \tag{2.12}$$

$$Index_{table} = \sum_{1}^{N} Index_{dim,i} \prod_{j=0}^{i-1} L_{dim,j} \quad with L_{dim,0} = 1 \tag{2.13}$$

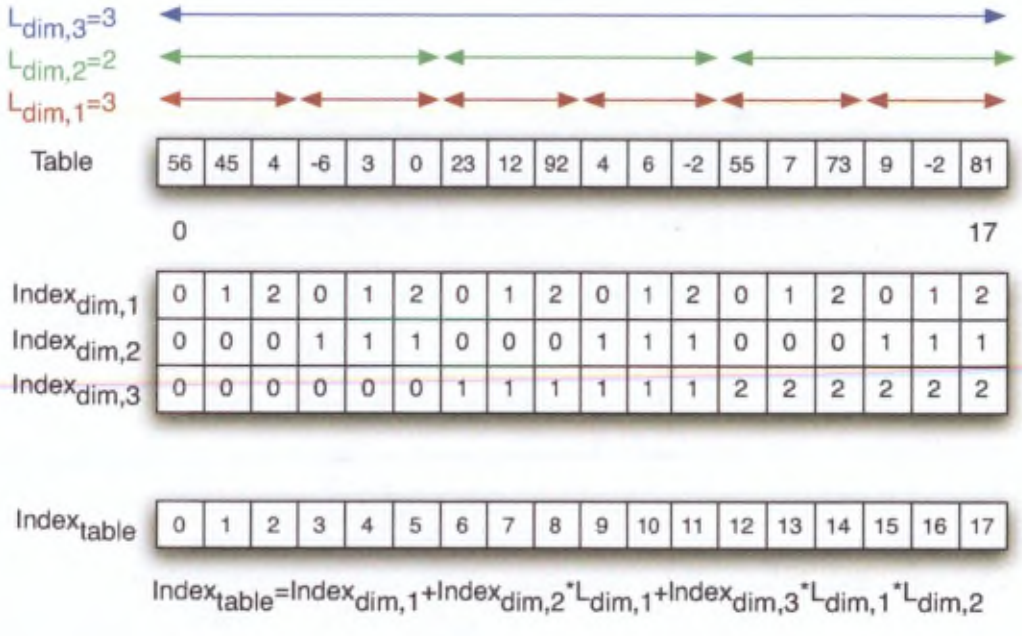

Figure 2.13: Evaluation of a table rule based on the dimensional indexes values: the conversion formula to get the table index is particularized for the case of 3 dimensions

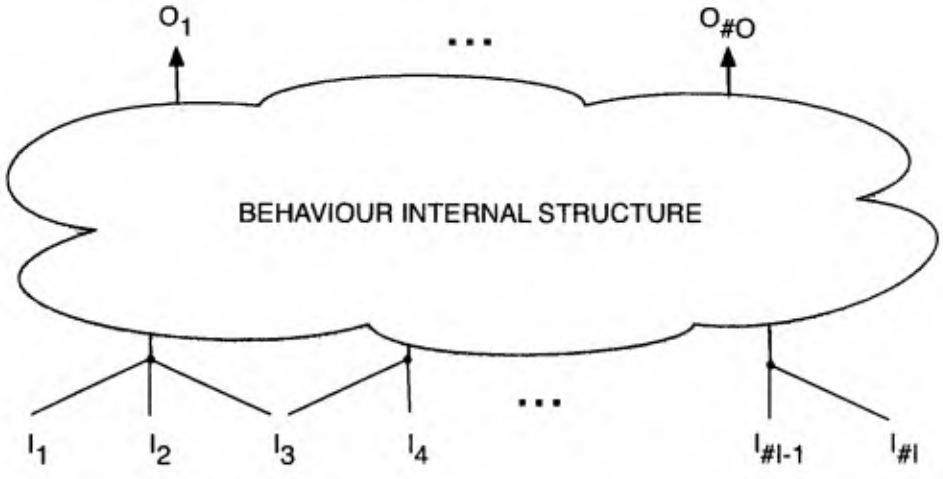

Figure 2.14: Representation of an oriented behaviour with $\#O$ outputs and $\#I$ inputs

2.5.2 Relation

Relations don't require the use of very specific algorithms hence we will skip this section to go directly to the behaviour devoted section.

2.5.3 Behaviour

Structure

Oriented behaviours are composed out of relations representing a structure with I inputs and O outputs as illustrated in Fig.2.14. Vertices pointed by only one oriented hyper-edge represent output parameters while vertices sharing simple or multiple non-oriented hyper-edges are input parameters.

Evaluation

Constraints and value estimation of the behaviour can be carried out by using again a depth-first algorithm at the scale of the relation rather than at the scale of basic operations (see Alg.1) for each successive particular behaviour output.

Orientation search

As explained in Sec.2.4.4, one behaviour can be used to solve different problems by simply changing the orientation of the underlying relations. Given the hypergraph structure of a non-oriented behaviour, we need to find all the possible cycle-free combinations that can be derived from it[4]. Therefore we will first formalize our problem into a mathematical representation to find the conditions that need to be satisfied to produce valid and executable oriented behaviours.

Mathematical representation A behaviour description is based on a list of M relations for a total number of N different parameters. Each relation Rel_i contains a set of P_{Rel_i} parameters which may vary from one relation to another.

If we first consider only *non-oriented* relations, the behaviour can be written as a mathematical system of M implicit equations (Eq.2.13).[5]. Among all the $X_{i,j}$ variables (the jth variable of the ith relation), there are only N different variables, meaning that the $\sum_{i=1}^{M}(P_{Rel_i}) - N$ others are common to at least two equations.

Evaluating the represented behaviour output values simply comes to solve this system of equation. We suppose that all the equations are independent so that the dimension of the system equals M. This statement implies that no equation can be found back using the $M-1$ other equations: since relations represent different models, this condition will likely be satisfied all the time (the user should make sure of that when composing a behaviour). As we now have M independent equations and N different variables, the number degrees of freedom of the system equals $N-M$ so that defining these $N-M$ values will entirely determine the M remaining variables values.

To explore all the possible orientations of the hyperedges, we can turn each of the M implicit equation into one of its explicit form (in other words associate each relation with one of its generic rule) which leads to $\#PotentialSolutions$ (Eq.2.14). Some of these solutions may however lead to mathematical discrepancies as we will illustrate in our following example.

$$\#PotentialSolutions = \prod_{i=1}^{M} P_{Rel_i} \tag{2.14}$$

Indeed let us consider as example one behaviour (Eq.2.16) composed out of four relations ($M = 4$), seven different parameters ($N = 7$) which leads to

[4]It must be emphasized that, although they are very similar, the hypergraph solutions we are looking for are not hypertrees strictly speaking. Indeed several hyperedges may share more than one vertex which violates the condition for an hypergraph to be an hypertree[17].

[5]We used implicit equations to emphasize the fact that related variables are not independent and that this equation form isn't suited to evaluate one variable value based on the others

108 different potential solutions (using Eq.2.14). If we focus on a particular solution by choosing one explicit form for each equation (like for instance in Eq. 2.17).

$$\begin{aligned} Rel_1(X_{1,1},\ldots,X_{1,P_1}) &= 0 \\ &\ldots \\ Rel_j(X_{j,1},\ldots,X_{j,P_j}) &= 0 \\ &\ldots \\ Rel_M(X_{M,1},\ldots,X_{M,P_M}) &= 0 \end{aligned} \tag{2.15}$$

$$\begin{aligned} Rel_1(A,C,D,E) &= 0 \\ Rel_2(B,E,H) &= 0 \\ Rel_3(D,F,G) &= 0 \\ Rel_4(E,G,H) &= 0 \end{aligned} \tag{2.16}$$

$$\begin{aligned} E &= GR_{Rel_1}(A,C,D) \\ E &= GR_{Rel_2}(B,H) \\ D &= GR_{Rel_3}(F,G) \\ G &= GR_{Rel_4}(E,H) \end{aligned} \tag{2.17}$$

There are two important things to notice when looking at the resulting equation system.

1. The variable E is the result of two different equations which makes no sense as it can only have a single value.
2. If we try to solve this system by a simple substitution of rules GR_{Rel_1}, GR_{Rel_3} and GR_{Rel_4} we get the resulting equation 2.18. As we can see, this equation is not explicit because the parameter E appears in both members which makes direct evaluation impossible.

$$E = GR_{Rel_1}(A,C,GR_{Rel_3}(F,GR_{Rel_4}(E,H))) \tag{2.18}$$

These two considerations give us the conditions to enable the equation system to be converted valid hypergraphs:

1. All the explicit equations have to be oriented towards different variables
2. After making all possible equation substitutions, no variable is allowed to appear in both members in any equation

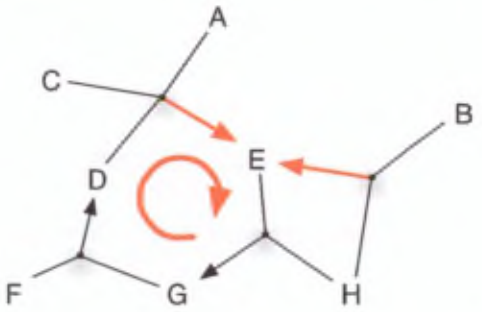

Figure 2.15: Illustration of the two conditions required to have valid executable graphs on our example

Hypergraph transposition Coming back to our behaviour graph representation, these two conditions on the mathematical representation can be respectively translated into two rules that an oriented graph should follow in order to be valid and executable:

1. Each vertex of the oriented hypergraph should have one and only one hyperedge pointing to it
2. The resulting hypergraph must be acyclic (in other words, starting from a vertex, there should be no path of consecutive oriented hyper-edges that could lead to the same vertex).

The previous example based on the equation set 2.17 can be converted into an oriented hypergraph (Fig.2.15). In this graph, both previous conditions aren't fulfilled leading to the same conclusions as for the equation version. Indeed two hyperdges point towards the same vertex (parameter E) and the hypergraph contains a cycle of three relations resulting in infinite loop execution when attempting to evaluate it through a depth-first algorithm.

To find all the solutions eligible for direct execution, we built a recursive algorithm to explore the hypergraph and extract all the oriented hypergraphs compatible with our two rules out of it. This algorithm proceeds in several steps that are described in a pseudocode (see Alg.2). Some particular points however deserve more attention:

- At he beginning of the algorithm, none of the M relations are assigned with an orientation. One by one relations are then associated with each possible orientation to explore the whole hypergraph.
  If a new orientation either breaks the *one oriented hypredge per node* or the *acyclic graph condition*, the current orientation is invalidated and all subsequent relations remain unexplored. This saves a lot of time since we know that, whatever the other relations orientations, it will only result in impossible solutions.

If the proposed orientation is valid, we proceed with the next relation until all of them have been explored.

- The condition of non-cycling hypergraphs is the most painful to verify. One possible way to determine if the adjunction of one relation will form a loop is to simply explore all yet oriented children relations of the hypergraph. The exploration ends up in two cases:
  1. Among the already explored parameters, we find back the oriented parameter of the tested relation. This means that an oriented path exists from at least one the children of the current relation to its root parameter, hence a loop.
  2. The oriented parameter of our relation under test has not been found among the explored parameters and all the hyperedges have been explored: the resulting hypergraph is cycle-free and this solution is eligible for evaluation.

This cycle detection process is illustrated in Fig.2.16: the left side shows an hypergraph with a loop (path of red double arrows) while the right side shows the same hypergraph free of any loop. The double arrows point towards all the parameters of already successfully associated relations that must be tested to detect potential loops when the green circled relation is added to the hypergraph.

Algorithm 2 Recursive algorithm extracting all the valid hypergraphs for evaluation based on a set of M relations and invoked using validHyperGraphs(firstRelation)

```
1: if (index(currentRelation) < M) then
2:   while All associations of currentRelation have not been tested do
3:     if currentAssociation doesn't create a loop inside the hypergraph ∨ parameter
       pointed by currentAssociation isn't already associated then
4:       validHyperGraphs(++currentRelation)
5:     end if
6:     ++currentAssociation
7:   end while
8:   Remove currentRelation association
9: else
10:  Save current solution
11: end if
```

2.6 Implementation

This section discusses some important implementation issues while focusing on maintainability, flexibility and practical questions.

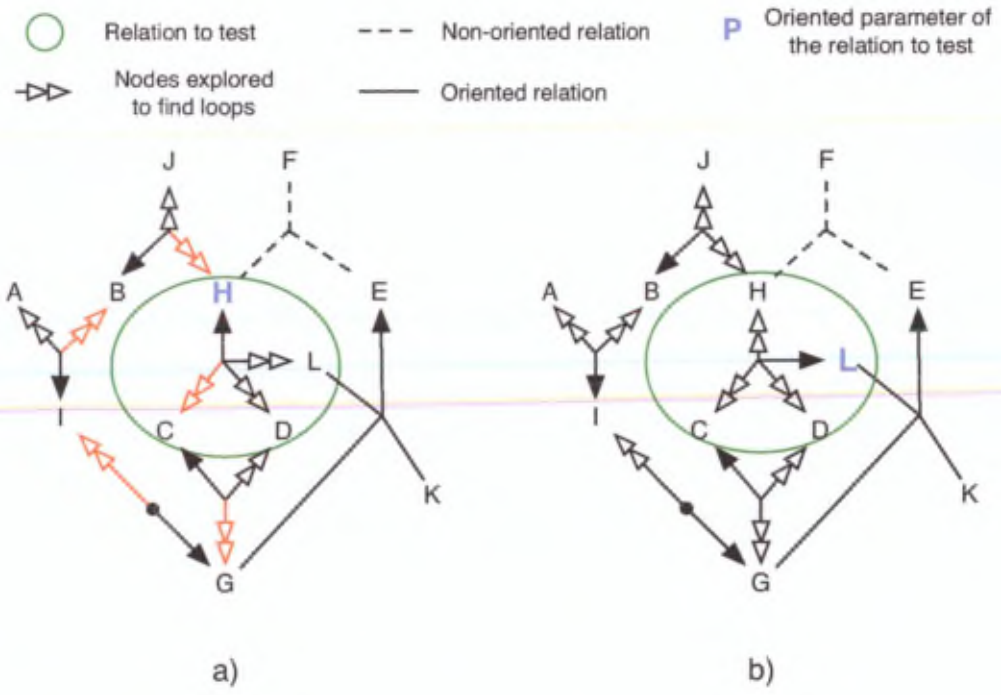

Figure 2.16: Exploration of an hyper-graph to find loops: a) represents a wrong solution while b) shows a cycle-free solution

2.6.1 Introduction

$YETi^3$ has been implemented in C++ using the multi-platform Eclipse 3.2.1 IDE combined to CDT 4.0 (the widest spread C++ plugin for Eclipse) which relies on GCC 4.0.1 compiler. To keep $YETi^3$ hierarchical structure intact, we chose an object-oriented language and went for C++.

The translation of the hierarchical structure into an object-oriented structure is almost immediate: Fig.2.17 represents the general UML class diagram[6] of $YETi^3$. The main classes are:

- The hierarchical representation of the three basic elements is clearly outlined by the UML relationships linking the classes. Indeed *relation* is linked to *behaviour* using a one-to-n composition relationship and so is *genericRule* to *relation*. It means that the lifetime of the aggregated class depends on the lifetime of the other class: for example, all the relation instances will be destroyed if the behaviour is destroyed. Only the behaviour is visible outside of the framework through its methods and both relations and genericRules classes remain hidden to the user.
- *Parameter* class is used as an interface mechanism to exchange data between the three main hierarchical classes. Therefore we used aggregation relationships to link them. The parameter class also contains one value and one constraint class instance.
- *Parameter MetaInformation* and *Relation MetaInformation* contain different temporary information used by the different algorithms but not related to the structure.

Almost all the relationships (except the ones between parameter and constraints/value) are bidirectional (no arrow at the other side of the relationship) meaning that classes mutually know about each other. These mutual class dependences mainly come from the multi-level nested tree structure with shared parameters: if this class structure seems quite obvious, it will however lead to thorny implementation problems that will be further discussed in Sec.A.2.

In the following sections, we present the different hierarchical elements and explain how they provide useful services to the upper layers.

2.6.2 Generic rules

Generic rules are responsible for the computing capability of the framework and offer the following services to higher level layers:

[6]This class diagram is very simplified: it contains only the class names and omits attributes, methods and even some less important classes. We intentionally decided to represent only the relevant information to avoid the diagram from being overloaded with unnecessary details. Through the rest of this dissertation, we will always adapt the detail level of each UML diagram depending on the context of use.

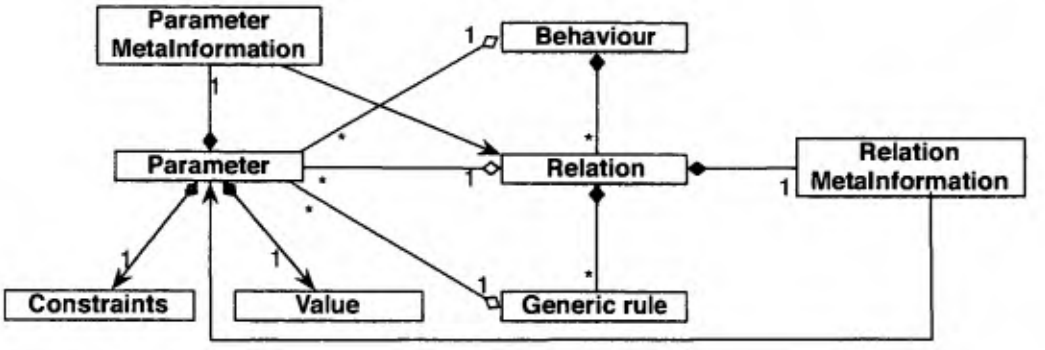

Figure 2.17: General UML diagram for the $YETi^3$ framework

1. *Evaluate* is a service that triggers the estimation of the root parameter of the generic rule based on its input parameter values
2. *Evaluate constraints* triggers in the same way the constraints evaluation of the generic rule root parameter

Fig.2.18 presents the UML class diagram for the generic rule, analytical/table rule an underlying classes.

- *Generic rule* class has been implemented by making intensive use of inheritance mechanisms to provide the framework with a lot flexibility and extension possibilities. Analytical rules and table rules derive from the *generic rule* class enabling easy rule extension if needed: derived classes just have to implement the *evaluate* and *evaluate* constraints virtual functions. These functions return a boolean with a *true* value if the value estimation encounters no error and inversely. The kind of errors depends on the particular type of generic rule: analytical rules triggers an error whenever one of its basic operation generates a mathematical error (division by zero, negative logarithm basis etc.) and table rules whenever one of the input parameter doesn't correspond to any of the predefined values.
- The internal tree structure of analytical rule (see Sec.2.5.1 for more details) is composed out of n smaller *basic operations* that are implemented by the *analytical Element* class. These basic operation are simple tree elements with 1 or 2 leaves that can be cascaded to form deeper trees: parameters play the role of interface between all these basic operations.
- Each particular basic operation is defined inside a separate class that inherits from the common *analytical element* class. This last one is defined as an abstract class so that only derived classes may be instanced.

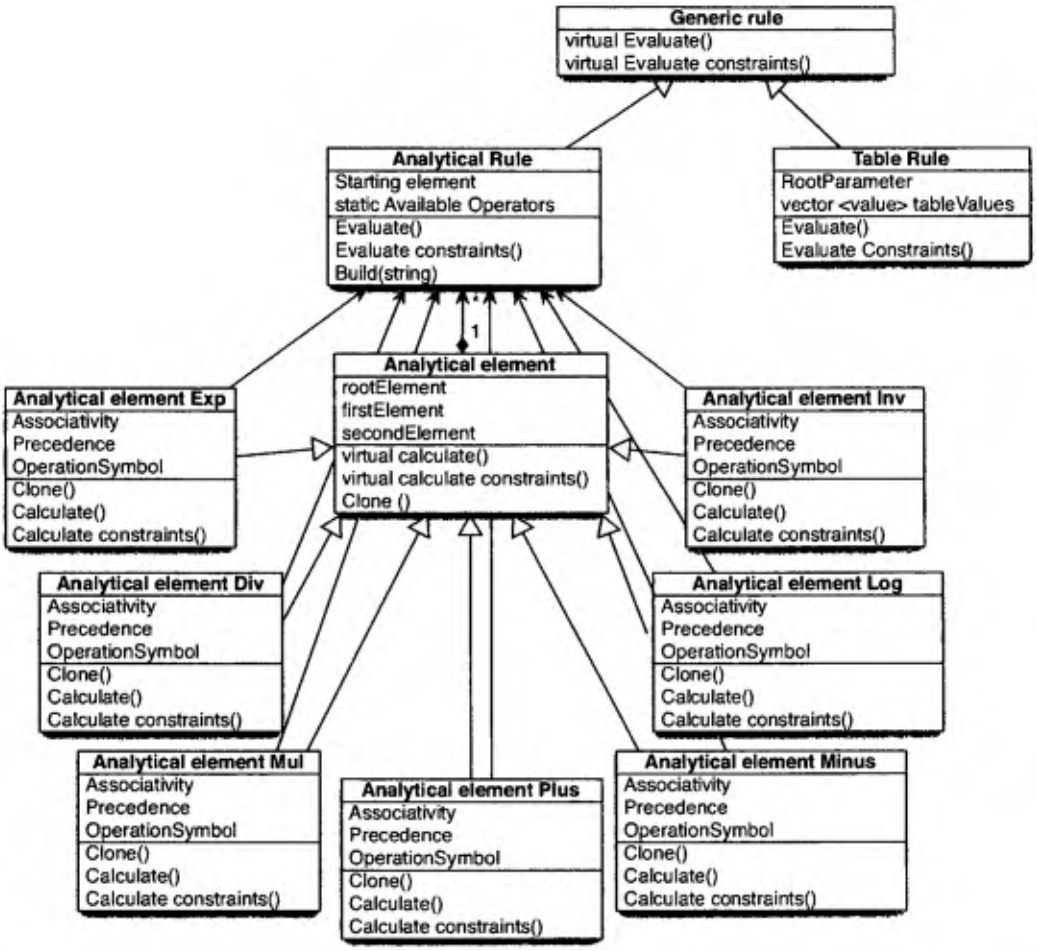

Figure 2.18: UML diagram for the generic rule related classes

Building analytical rules: the price of flexibility

Analytical rules are described by a string capturing the mathematical representation of the rule. To turn them into trees, we used the *shunting yard algorithm* which is explained in Sec.A.1 but some particular implementation points deserve some attention.

Indeed when the input string is parsed, the algorithm needs to be able to identify an operator by its symbol and distinguish it from a parameter name. To make this mechanism more flexible and to enable the later extension of basic operators, we encapsulate all the required information (see tableA.1 in Sec.A) inside each particular derived *analytical element* class element using static class attributes (as this information is common to all instances). However, *analytical rule* class that implements the shunting yard algorithm needs in turn to know about these classes as this is shown by the dependence relationships in the UML class diagram. Hence we included a class attribute named *available operators* which gather all the existing derived analytical element classes into a vector. All these different instances don't enclose any useful data but are just put together so that we have a way to systematically check all the possible basic operators regardless of their number. Furthermore we included this vector as a static attribute to avoid the burden of duplicating it for each instance of the *analytical rule* class.When a match between the parsed string and an analytical element symbol is found, we first need to duplicate it before inserting it into the tree structure. The problem is that we have to instantiate the chosen derived analytical element class and not the super class. To do so, we used a particular mechanism: each derived analytical element implements an inherited *clone* method that returns a pointer to a newly created instance of itself so that we can use this new instance in out tree.

To summarize, the only things that need to be done in order to add a new basic operation to the current set is:

- To create a new class that derives from the analytical element class while defining the clone, calculate and calculate constraints methods and to define symbol, precedence and associativity static attribute values.
- To add the new derived class to the available operators vector of the analytical rule (along with its header declaration).

In return for these slight modifications, the shunting yard algorithm is still working properly even with analytical rules using these new basic operations can be used without any change.

Accelerating analytical relation evaluation

The evaluation speed of analytical relations is crucial in determining the global performance of $YETi^3$. Therefore we used a mechanism to lower execution

times that takes the most out of the mathematical expression representing the model: common terms will be merged into same branches of the tree so that they require only one estimation to get the output value. The underlying mechanism is very simple: whenever we parse an expression, we check if we already came across any equivalent expression given the associativity (see TableA.1). For instance, expressions $A + B$ and $B + A$ are similar since operator $+$ is associative while expressions $A - B$ and $B - A$ will produce different results due to the non-associativity of operator $-$. This "tree compression" procedure doesn't take a lot of time and has to be done once and for all during the analytical rule building phase while increasing in turn model estimation speed.

2.6.3 Relation

Relations have a very simple object structure since they only need to memorize the orientation of each parameter associated and provide the user with a mechanism to dynamically switch the association. Fig.2.19 presents the UML class diagram associated to relation and highlights the main aspects of it:

- The most important class attributes are composed out a vector of *parameters* for the relation and a vector of associated *genericRules*. Only non-constant parameters are gathered into the *parameters* vector since it makes no sense to associate the relation with a parameter that has a constant value. Furthermore, constant parameters are different from one generic rule to another so that none of them may be common to the same relation. Finally some of the parameters could not be associated at all (NULL pointer value at the corresponding index in the *genericRule* vector) if the related model is not reversible for that particular output parameter.
- The *evaluate* and *evaluateConstraints* functions return a boolean value to indicate whenever the *genericRule* evaluation encounters an error (see Sec.2.6.2). Both functions take as input the name of the output parameter to tell the relation which genericRule should be used for the evaluation and modify in turn the value or constraints of this parameter.
- A *setGenericRule* function can be used to change the generic rule associated to one particular parameter allowing model sensitivity study.

2.6.4 Behaviour

Behaviour offer the upper-layer services that will directly be available for the user. Fig.2.20 presents the most relevant methods and attributes of the class:

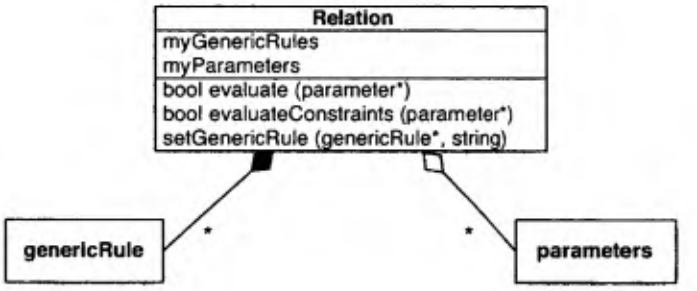

Figure 2.19: UML diagram for the *relation* related classes

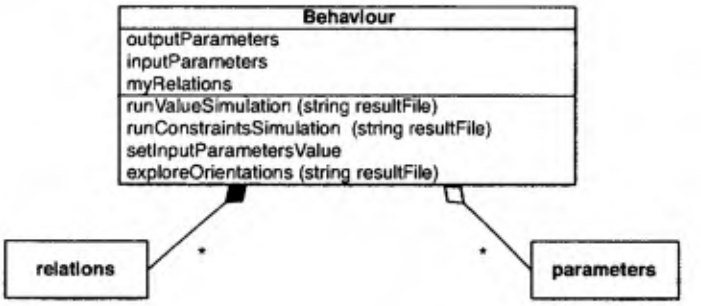

Figure 2.20: UML diagram for the behaviour related classes

- The main attributes are the output/input parameters and the relations composing the behaviour

- *setInputParametersValue* allows the user to set the value of each parameter by specifying a single value, a list of values or a value sweep (start, stop and step value). Values can be defined using the *scientific E notation* for numbers.

- *RunValueSimulation* and *runConstraintsSimulation* trigger a simulation run based on the input parameters values and write the results in the XML *resultFile*. All the combinations of each input parameter possible values are computed to get the complete input values set.

- *exploreOrientations* is a function that calls the algorithm for cycle-free hypergraph extraction (see Sec.2.4.4) and gather the valid orientations into the XML *resultFile*.

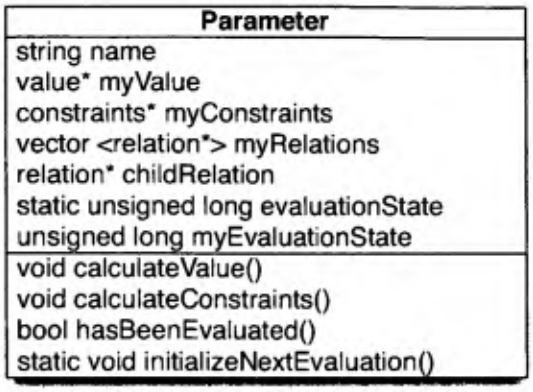

Figure 2.21: UML diagram for the behaviour related classes

2.6.5 Parameters

Parameters are the key to the hierarchical organization of our object structure since they provide a communication interface between all layers. Their UML class description (Fig.2.21) may seem surprisingly simple but contains however a lot of crucial information.

Parameters attributes include the name (unique for each parameter at the relation level), the value/constraints and the orientation. The two methods called *calculateValue* and *calculateConstraints* trigger the computation of the parameter value/constraints.

Since our framework is parameter-centric, we decided to include the information about the orientation of the relation tree inside the parameter. Therefore we used the *childRelation* attribute that points towards the relation that is associated with this current parameter or is assigned with a NULL pointer value (in the case of a leaf parameter).

To speed up the the behaviour evaluation, we set up a mechanism to spare a lot of redundant evaluations associated with common hyperedges. Let us consider one example of a behaviour represented by its hypergraph composed out of relations (see Fig.2.22). As we can see internal parameter named $Int1$ is shared by two hyperedges with root parameters name O_1 and O_2. In order to proceed to the evaluation of the different trees output parameters values we first define the value of each input parameter. Using the depth-first algorithm, we can easily evaluate parameter O_1 value by exploring the whole tree underneath. However, if we do the same for parameter O_2 estimation, we'll waste some precious computation time since the oriented hypergraph starting at parameter Int_1 has been evaluated previously. The solution to this problem is to keep track of the already visited parameters and their output values so that we don't

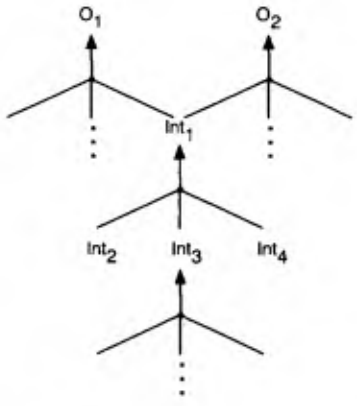

Figure 2.22: Acceleration of behaviour estimation thanks to common branches identification

explore twice the same hyperedge during one hypergraph evaluation.

One solution to implement this mechanism is to tag each parameter with a marker each time that they visited: when we come across a parameter that has already been evaluated, we just get the previously estimated value instead of exploring further the hypergraph.

2.6.6 The big picture

The behaviour value/constraints estimation is a very flexible and easy-to-use process but puts in turn some stress on the message-passing mechanism between objects. Fig.2.23 presents a UML sequence diagram[7] describing how objects exchange information to get all the behaviour parameters estimated:

- The user calls a *runValueSimulation* (the principle remains the same for constraints evaluation)
- For each output parameter, the behaviour calls the *calculate* method that uses the internal *hasBeenEvaluated* method of parameter. If the result of this function is true (meaning that either the parameter has no oriented hyperedge pointing at it or the parameter has already been evaluated previously) no other method call is made. If it is false, the parameter calls the *evaluate* method of relation by passing its name to define the orientation of the relation.
- Relation call the polymorphic *evaluate* method of the *generic rule* class

[7] Object lifetime is intentionally not represented since all objects remain alive during the estimation process.

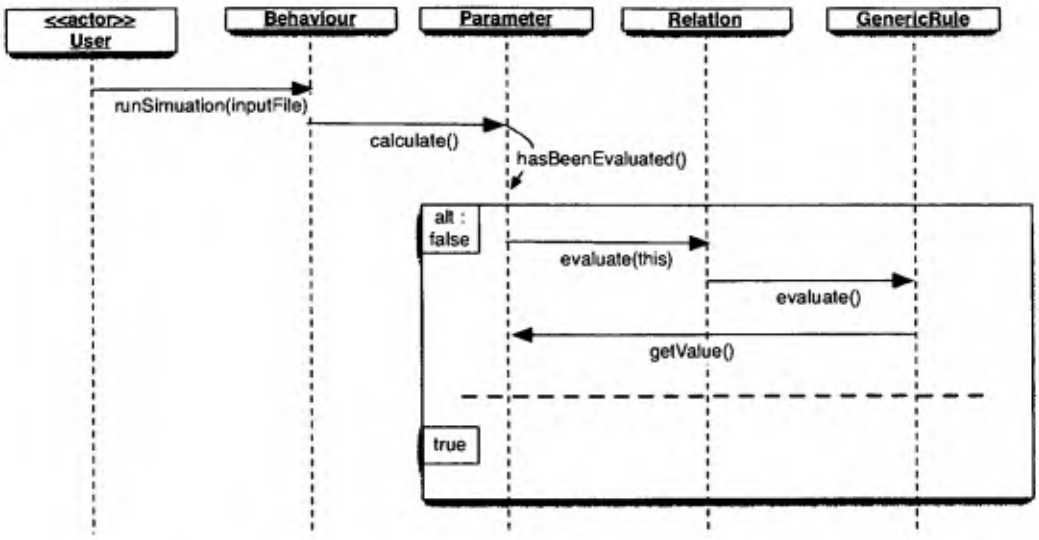

Figure 2.23: UML sequence diagram representing message passing between all the classes during model evaluation

- Any estimation process from a class derived from *genericRule* uses the *getValue* method to get the parameter values. This function works exactly the same way as the parameter *calculateValue()* hence triggers parameter estimation if it hasn't been estimated yet.

The message passing structure could seem unnecessarily complex but it has the enormous advantage of providing extremely easy extension possibilities. Indeed it enables any class deriving from genericRule to be directly used inside a relation without any change on the behaviour depth-first evaluation method. This provides the programmer with a totally invisible yet powerful and very flexible mechanism of evaluation.

2.6.7 Using the framework

$YETi^3$ can be basically used in two different ways: either as a C++ library or as a standalone tool using the XML scripting functionalities. While the first operation mode can be used to easily extend an existing C++ application with enhanced modeling capabilities, the second mode allows the user to use $YETi^3$ without having to compile any code and directly script its experiments. All the services available for the user to choose among are the following:

1. Building a behaviour
2. Specifying the orientation of a behaviour

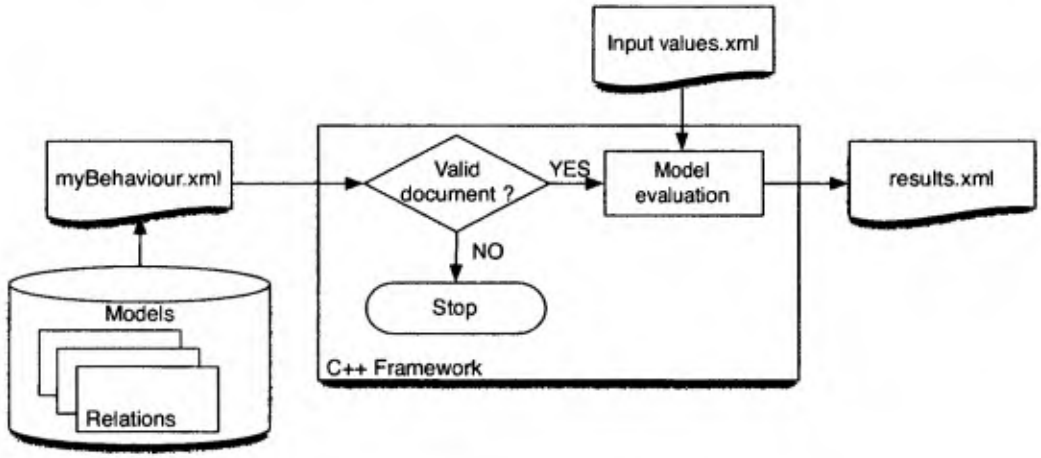

Figure 2.24: Functional view of the $YETi^3$ framework

3. Setting the input values/constraints
4. Performing a constraints/value estimation and generate the result in a file
5. Determining all the valid orientations of a behaviour and write the result into a file
6. Changing the generic rules associated to a relation
7. Creating a ready-to-plot file based on the selection of the desired variables

From a user perspective, the basic behaviour evaluation mechanism of the framework is described in Fig.2.24.

First the user needs to describe the different XML input files (behaviour, input values, behaviour orientation).

Second $YETi^3$ reads the commands list (either described using the C++ interface or the XML script) and fetches the corresponding XML files before executing each command. If one of the file isn't valid regarding its grammar defined by the corresponding schema, the operation is aborted and the error is displayed.

Third the operations are carried out and the corresponding XML output files are generated (value/constraints simulation and result from behaviour orientation search).

2.7 Data support

$YETi^3$ deals with a lot of data: three layered structured models, input/output parameters values, behaviour orientations etc. All these structured information need to be stored into files and should afterwards be easily turned into object oriented elements. To provide our framework with such functionalities, we decided to use XML as data support format. The reason of this choice, the different language features and the XML related tools are topics that are discussed in Sec.B.1 while the details about the XML schemas defined for Yeti can be found in Sec.B.2.

2.8 Conclusions

In this chapter we presented Yeti, our tool for flexible analytical and table-based relations definition and evaluation. Compared to previous closed-formed performance prediction tools, Yeti has the advantage of being very flexible and can be used either as a standalone tool or as a C++ library. Its data support defined using XML schemas guarantees that the framework will automatically detect any error made by the user during the definition of the simulation. The introduction of the notion of model reversibility also enables easy reuse of existing models in different situations than the ones they were originally meant for. Finally Yeti allows the user to script its simulations to gain a lot of time and plot the results which make it usable to easily compare models and perform input sensitivity studies.

To demonstrate the use features and how they can help to optimize VLSI design decisions, the next chapter will be entirely devoted to the presentation and discussion of several VLSI related case studies performed using Yeti.

Bibliography

[1] D. S. C. Hu, "Analytical modeling and characterization of deep-submicrometer interconnect," in *Proc. IEEE*, IEEE, Ed., vol. 89, 2001, pp. 634–664.

[2] B. S. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Trans. Comput.*, vol. 20, no. 12, pp. 1469–1479, 1971.

[3] P. Christie and D. Stroobandt, "The interpretation and application of rent's rule," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 8, no. 6, pp. 639–648, 2000.

[4] G. Sai-Halasz, "Performance trends in high-end processors," *Proceedings of the IEEE*, vol. 83, no. 1, pp. 20–36, January 1995.

[5] D. Stroobandt, *A Priori Wire Length Estimates for Digital Design.* Boston / Dordrecht / London: Kluwer Academic Publishers, 4 2001.

[6] S. Takahashi, M. Edahiro, and Y. Hayashi, "A new lsi performance prediction model for interconnection analysis of future lsis." in *ASP-DAC*, 1998, pp. 51–56.

[7] J. A. Davis, V. K. De, and J. Meindl, "A stochastic wire-length distribution for gigascale integration(gsi) - part i: Derivation and validation," *IEEE Transactions on Electron Devices*, vol. 45, no. 3, pp. 580–589, Mar 1998.

[8] W. E. Donath, "Wire length distribution for placements of computer logic," *IBM Journal of Research and Development*, vol. 25, no. 3, pp. 152–155, may 1981.

[9] R. Mangaser and K. Rose, "Facilitating interconnect-based vlsi design," in *MSE '97: Proceedings of the 1997 International Conference on Microelectronics Systems Education (MSE '97).* Washington, DC, USA: IEEE Computer Society, 1997, p. 139.

[10] J. C. Eble, V. K. De, D. S. Wills, and J. D. Meindl, "A generic system simulator (genesys) for asic technology and architecture beyond 2001," in *ASIC Conference and Exhibit proceedings*, 1996, pp. 193–196.

[11] "International technology roadmap for semiconductors 2007." [Online]. Available: http://www.itrs.net/

[12] L. Codrescu, M. D. Pant, T. M. Taha, J. Eble, D. S. Wills, and J. D. Meindl, "Exploring microprocessor architectures for gigascale integration." in *ARVLSI*, 1999, pp. 242–255.

[13] D. Sylvester and K. Keutzer, "System-level performance modeling with bacpac – berkeley advanced chip performance calculator," 1999. [Online]. Available: citeseer.ist.psu.edu/sylvester99systemlevel.html

[14] D. Liu and C. Svensson, "Power consumption estimation in cmos vlsi chips," *Solid-State Circuits, IEEE Journal of*, vol. 29, no. 6, pp. 663–670, 1994.

[15] "Bacpac - berkeley advanced chip performance calculator." [Online]. Available: http://www.eecs.umich.edu/~dennis/bacpac/

[16] A. E. Caldwell, Y. Cao, A. B. Kahng, F. Koushanfar, H. Lu, I. L. Markov, M. Oliver, D. Stroobandt, and D. Sylvester, "GTX: the MARCO GSRC technology extrapolation system," in *Design Automation Conference*, 2000, pp. 693–698. [Online]. Available: citeseer.ist.psu.edu/caldwell00gtx.html

[17] F. Chapoton, "Hyperarbres, arbres enracin
'{e}s et partitions point
'{e}es," *HOMOTOPY AND APPLICATIONS*, vol. 9, p. 193, 2007.

[Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:math/0604525

Chapter 3

Yeti: Case Studies and Applications

Abstract

In this chapter, we define two complete case studies to demonstrate the different features of Yeti including simulation scripting, input parameter sensitivity study and automatic plot generation. The first case study focuses on the reuse and extension of a model selected from the literature for the estimation of the computation performance of multi-core processors architectures. Besides the ability to easily capture models and reproduce their results, we show how we can take advantage of the model reversibility feature of Yeti to get new results from existing models in almost no time. In the second case study, we build a model from scratch to estimate the bandwidth and frequency of a communication path depending on its number of stages. We compare our results with the literature, explain some discrepancies found between key papers and propose new wire planning for optimized bandwidth. As a conclusion we present the calculation performances of Yeti and discuss its scalability with model complexity.

3.1 Introduction

While the previous chapter focused on the theoretical and implementation concepts used for the design of Yeti, the current chapter demonstrates the use, the different features and the performances of the Yeti framework on practical cases. We therefore performed two different case studies:

1. In the first case study, we illustrate Yeti ability to reproduce results of existing models and extend them using our reversibility feature. Therefore we will focus on a paper from the computer science literature describing the computation performances of multi-core processors and show how it is possible using Yeti to derive new results with very almost no model modification and time penalty.
2. The second case study demonstrates how Yeti can be used to perform model/input parameter sensitivity studies allowing the user to see the impact of some model/input variations on the model result calculated by the engine. As an illustration, we will model from scratch a communication path to estimate its maximum frequency based on the number of stages that it is composed out of. It will allow us to discuss the choice of the stage model along with its impact on delay for different wire lengths and examine how the different geometrical parameters of the gate and wire impact the resulting maximum frequency of the path.

Through these case studies, we will also illustrate the side functionalities of Yeti that make user's experience much more comfortable like automatic plotting based on the resulting XML files and simulation scripting.

3.2 Case Study 1: the Codrescu model

3.2.1 Introduction

The aim of this first case study is to demonstrate how it is possible to describe an existing model taken from the literature using Yeti formalism and how we are able to modify its orientation in almost no time to answer problems that it was originally not meant for. Therefore we have chosen a model defined by Lucian Codrescu in [1] studying the impact of the core number and type of a multi-processor architecture on computation performances of the platform and its interaction with the application parallelism[1]. Although this study is a bit old now (it was published in 1999), it is a very good example to demonstrate the different features of Yeti:

- This study links technological parameters (like wire and gate delay) to high-level platform (like the silicon surface and architecture granularity) and functional (application parallelism) related parameters. This modeling context based upon heterogeneous parameters from very different design abstraction levels makes the concept of reversible modeling introduced by Yeti very interesting to use and study.

[1] We have already briefly presented this model before in Sec.2.2.6 devoted to system-level prediction systems. The generality of Yeti allows us to express this model inside our framework emphasizing in practice its generality compared to previous analytical models.

- The paper details and references all the intermediate models used so that it is possible to reproduce the published results. This particular point is crucial since there are -unfortunately- very few papers that make all the models explicit while relying on a sufficient number of linked models.
- Analytical and table-based models are used at the same time in Codrescu's model allowing us to show how Yeti is able to mix different types of models.
- Finally the presented results are limited to the study of the dependence of the global platform computation performances with the application parallelism for different combinations of core type/number keeping the total chip area constant. Since this practical use of the models is quite limited regarding their number, there is room for performing some more experiments based on the very same models that only require to be reoriented.

Let us first explain in more details than Sec.2.2.6 the different models involved in Codrescu's study before integrating them in Yeti and trying to reproduce the paper results.

3.2.2 Codrescu model

Codrescu's model targets the representation of the computation performances of a multi-core processor architecture. Therefore different architectures are compared from a large single processor to a network of 256 simple processors: the number of processors for each architecture candidate is chosen so that the total area remains constant making their comparison fair in terms of total silicon surface. This situation is illustrated in Fig.3.1 where each architecture is represented by its processor number and issue width[2]. The basic idea behind this model was to find the most suitable candidate architectures for next generation microprocessors that could sustain Moore's law performance level. Therefore the different architectures are compared in terms of computing performance for different levels of application parallelism: in other words, the paper tries to determine in which proportion Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP) should be mixed in order to obtain the best level of performances.

The computation performances are measured by the number of operations issued per second for the complete architecture i.e. by all the processors: this is expressed by Eq.3.1 where $Perf$ is the computation performances of the platform (in Operations/s), F_{clock} is the clock frequency, IPC is the number of instructions issued per cycle and Sp is the speedup factor quantifying how much the performances benefit from the adjunction of additional parallel

[2]The issue width represents the number of functional units exploiting the instruction level parallelism to increase computation performance.

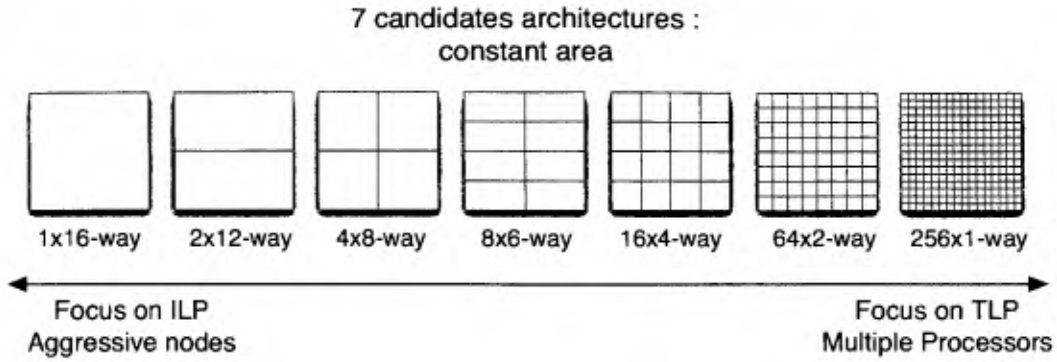

Figure 3.1: Seven multi-processor architecture candidates with a constant total silicon area are compared in terms of computation performance in the study of [1]

microprocessors given the available level of application parallelism.

$$Perf = F_{clock} * IPC * Sp \tag{3.1}$$

Each factor of Eq.3.1 is calculated based on other models which are precisely described in the following paragraphs.

IPC The number of instructions per cycle IPC for each type of processor has been estimated using two different techniques depending on their issue width:

- Processors up to 4-way were well-known at the time of the paper; their respective IPC has thus been estimated based on average IPC values of existing processors with the same issue width.
- To define realistic values for the IPC of processors over 4-way, the average efficiency of superscalar processors has been estimated (the ratio between the effective IPC and the number of functional units i.e. the instruction issue width) for a large number of processors. With typical values between 30% and 50% the geometric mean value of this efficiency equals 40%. From there, IPC values were inferred for processors with issues width larger than 4-way: these values were partly confirmed by simulations performed using SimpleScalar[2] and can be calculated from Eq.3.2 linking the IPC with IW (the issue width).

The different IPC values are gathered in Table 3.1 for each type of microprocessor.

$$IPC = 0.4 * IW \tag{3.2}$$

Processor type	Area(mm^2)	IPC
16-way	416	6.4
12-way	208	4.8
8-way	104	3.2
6-way	52	2.4
4-way	26	1.6
2-way	6.5	0.8
1-way	1.6	0.4

Table 3.1: IPC and area for the different processor types

Processor type	Wire delay (ps)	Gate delay (ps)	Clock frequency (MHz)
16-way	972	400	0.73
12-way	687	400	0.92
8-way	486	400	1.13
6-way	344	400	1.34
4-way	243	400	1.56
2-way	122	400	1.92
1-way	61	400	2.17

Table 3.2: Wire and gate delays for the different processor types

Clock Frequency The clock frequency F_{clock} of each microprocessor is based on the critical path delay CP which is supposed to be composed out of ten stages. Each stage is a gate driving a wire segment whose length depends on the area of the microprocessor. The gate delay D_{gate} and the total wire delay D_{wire} are calculated using GENESYS[3] that we mentioned earlier in this work (see Sec.2.2.5). No information is given about the technology node which the gate and wire delays are calculated for.

$$F_{clock} = \frac{1}{CP} = \frac{1}{10 * D_{stage}} = \frac{1}{10 * (D_{gate} + D_{wire})} \tag{3.3}$$

The gate delay, wire delay and resulting clock frequency (calculated using Eq.3.3) are given in Table 3.2.

Speedup Amdhal's law[4] is used to estimate the speedup provided by the use of multiple processors over a single processor. This law represented by Eq.3.4 links the performance speedup Sp with the number of parallel execution nodes $\#Nodes$ and the parallel fraction[3] of the workload Par. As a result we

[3]The parallel fraction of a workload represents the percentage of the program in size whose execution can be parallelized onto an arbitrary number of execution nodes. The remaining of the program can only be executed on a single node.

obtain the speedup comprised between 1 for single threaded programs and $\#Nodes$ for fully parallelizable applications.

$$Sp = \frac{1}{\frac{Par}{\#Nodes} + (1 - Par)} \tag{3.4}$$

3.2.3 Integration of Codrescu's model inside Yeti

In this section we demonstrate how it is possible to build a model presented in Codrescu's paper and reproduce the results using Yeti.

Methodology

Converting all the mathematical models used by Codrescu into Yeti relations leads to 6 analytical and 3 one-dimensional table-based relations. These different relations are graphed in Fig.3.2 where they are all oriented according to the models of Codrescu. As we can see the resulting behaviour has three inputs, the parallel fraction, the total area[4] and the type of processor (in red on the figure) and one output, and the computation performances of the whole architecture (in green on the figure).

To reproduce these results with Yeti, we proceed as following:

- We define the behaviour with its oriented relations inside an XML simulation file: to give an idea of its complexity, the file is 253 lines long.
- An XML simulation script file defines all the simulations that will be performed with their respective input parameters values, desired output files and plot file names. For each type of processor, we perform a single simulation and sweep the parallel fraction parameter value so that we obtain a simple plot files for each architecture candidate.

Once these files have been defined, we can launch the simulation and wait for the different plot and output values files to be generated. Once this operation is completed (it takes a few seconds depending on the number of solutions to evaluate), we can directly visualize the resulting graphs by opening the plot files into the chosen plotter (*Plot*©for Mac OS X in our case).

[4] It may also be mentioned that two additional relations are present on the graph respectively defining the node area and linking the total area with the number of nodes and their type. In the case study provided by Codrescu in his paper, all architectures are chosen so that their total silicon surface remains constant allowing fair comparison. The related area parameters will however be used in some of our following experiments with the model of Codrescu

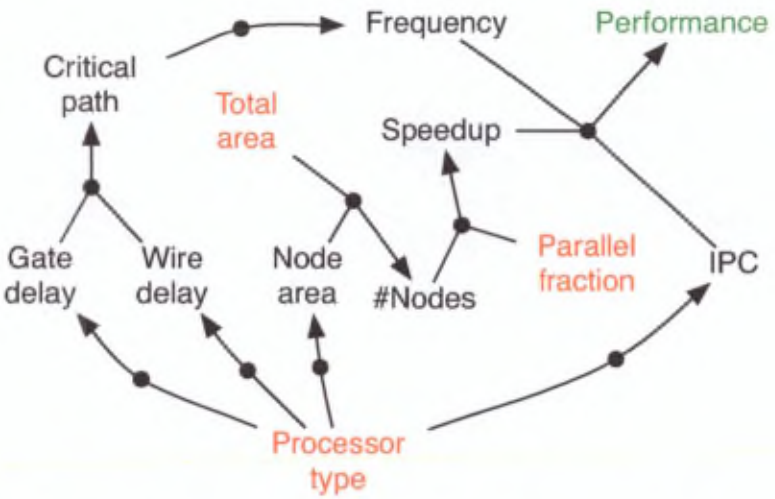

Figure 3.2: The eight relations representing Codrescu's model inside Yeti. The orientation of the relations is performed according to Codrescu: output parameters are colored in green and input parameters in red.

Experiments and results

We performed two different sweeps for the workload parallel fraction from 0 to 0.5 with a step of 0.1 and from 0.9 to 1 with a step of 0.01. In Yeti, it is pretty easy to generate the second batch of results based on the first simulation set: we just need to change the bounds and step value of the workload parallel fraction sweep tag in the input parameters values file, a matter of no time.

The two resulting curves are respectively represented in Fig.3.3 and Fig.3.4; taking back the paper results, we can see these curves are perfectly identical and our calculations perfectly match the original values from the paper. This demonstrates that Yeti is able to reproduce in a very reasonable time results taken from a paper based on a set of mixed analytical and table-based models.

On the first graph 3.3, we can see that the 16-way single-threaded mono-processor performs better than the other architectures for workload parallel fractions values smaller than 0.1; over this value, parallel architectures with smaller processors progressively reach a larger number of issued instructions per second making them a better choice. The computation performances remain of course constant for the single processor: whatever the level of parallelism, it will execute the whole code sequentially. Over 0.5, the four represented parallel architectures perform better than the mono-processor making intensive ILP less interesting than TLP.

On the second graph presented in Fig.3.4, we can observe the computation performances for very high workload parallel fraction values comprised between

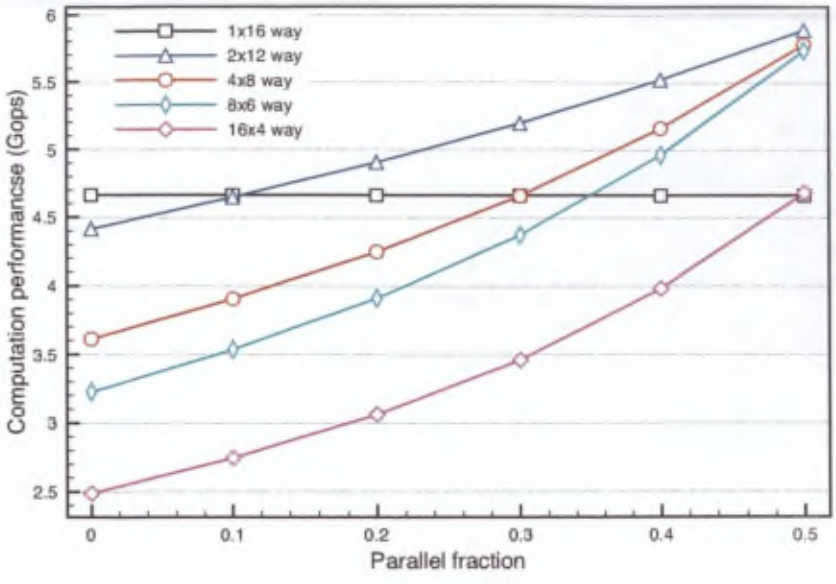

Figure 3.3: Computation performances VS workload parallel fraction ranging from 0 to 0.5 for Codrescu's model

0.9 and 1. This graph confirms the trend of highly parallel architectures becoming more and more interesting with the increasing parallel fraction; it also shows the incredible level of performances reached by the 64 and 256 core architectures which quickly outperforms other architectures for workload parallel fractions values over 0.97. It's only over 0.99 that the 256 cores architectures becomes the most interesting out of all the candidates.

In conclusion of its paper, Codrescu highlights the fact that reaching high-end computation performances is only possible at the price of highly parallel architectures coupled to applications exposing a very high degree of parallelism. It may be argued about this conclusion that with the years, the processor architectures have indeed shown a trend to embed a growing number of cores inside the same package (up to four identical cores in the current state-of-the-art of commercial GP processors [5]) but in a smaller number than the predictions made by Codrescu in his paper. The main reason for that may be the fact that general purpose processors are precisely not meant for the execution of applications with a high parallel fraction making the use of higher parallel multi-core processors based architectures less profitable than a few complex superscalar interconnected processors. Furthermore Codrescu completely ignores the impact of shared memory and interconnect on his results while architectures become more and more greedy for both these resources as the number of cores increases; this is typically why we need to make use of network-on-chips

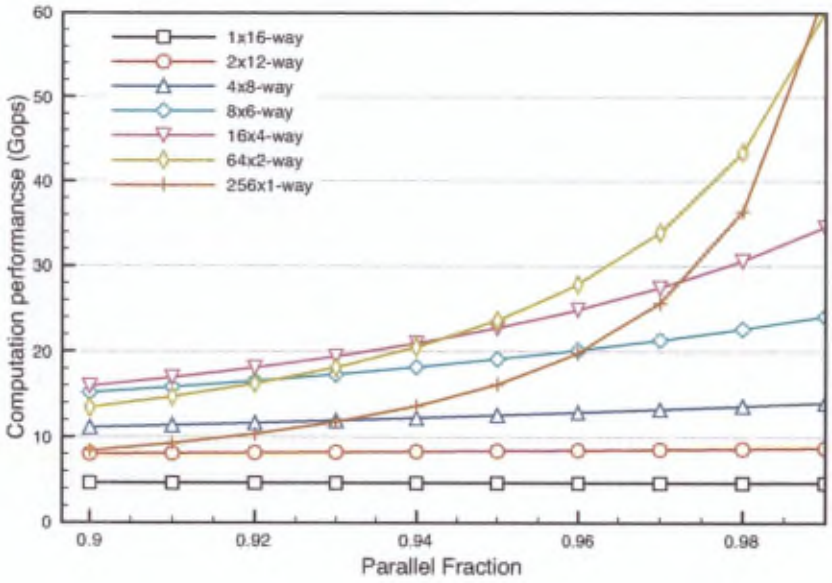

Figure 3.4: Computation performances VS workload parallel fraction ranging from 0.9 to 1 for Codrescu's model

when the number of tiles to interconnect grows. For highly parallel architectures, this additional cost in terms of memory and interconnect requirements probably makes those architectures prohibitive to design.

Now that we have shown how we have imported Codrescu's model into Yeti and found back the same results as those presented in the original paper, we will demonstrate Yeti abilities to reuse the very same model to derive new meaningful results based on two different case studies.

3.2.4 Extending Codrescu's results

In this section we will demonstrate the reversibility feature of Yeti by inverting the initial Codrescu's model and perform two different experiments:

- We first invert the model to express the computation performance as an input and the workload parallel fraction as an output parameter allowing the user to define the desired level of performance and get the minimum parallelism value required by the application to meet that constraint.
- The second experiment consists in reversing the initial model to express the total area as a function of the processor type, computation performance requirements and the workload parallel fraction. As we will see,

this study offers very interesting results when it comes to minimize the total area for a given computation performance level.

Parallelism versus computation power

The results obtained by Codrescu in his paper were meant to find the architecture exhibiting the best computation performances given different levels of workload parallelism; in other words, the aim of the model was to express the computation power value as a function of the parallelism fraction value. For a designer, it may however be convenient to select the desired level of performance (hence the number of instructions issued per second) and get in return the minimum parallelism fraction value required to sustain this level of performances for each individual architecture choice. This information could be used to check if the required level of workload parallelism seems reasonable given the targeted application.

Setting up the experiment To answer that problem, we thus need to change the orientation of the model to turn the parallel fraction Par into an output of the model and the computation performances $Perf$ into one of its input: this is illustrated in Fig.3.5. As we can see, all the relations remain the same except two relations which become oriented towards the *Speedup* and towards *Parallel fraction* parameters as respectively expressed by Eq.3.5 and Eq.3.6. We only need to manually define the new analytical rules and to change the orientation of these two relations inside Yeti to initialize a new simulation and get the results of this model inversion required to solve this new problem.

$$Sp = \frac{Perf}{F_{clock} * IPC} \tag{3.5}$$

$$Par = \frac{1 - Sp}{Sp * (\frac{1}{\#Nodes} - 1)} \tag{3.6}$$

The results are plotted for each architecture candidate proposed by Codrescu in Fig.3.6 for performances ranging from 0 to 60 Gops/s. As we could expect, the resulting graph actually represents nothing but the inverted function of the graphs resulting from Codrescu's original experiments. The only difference is that we have used a smaller step and have extended the performances values range to cover in one sole graph the whole range of values that were instead represented in Codrescu's paper on different plots. Although the graph plotted in Fig.3.6 doesn't really offer new information compared to the original results, it is more convenient for our considered problem and has been obtained in almost no time based on the previous model using Yeti.

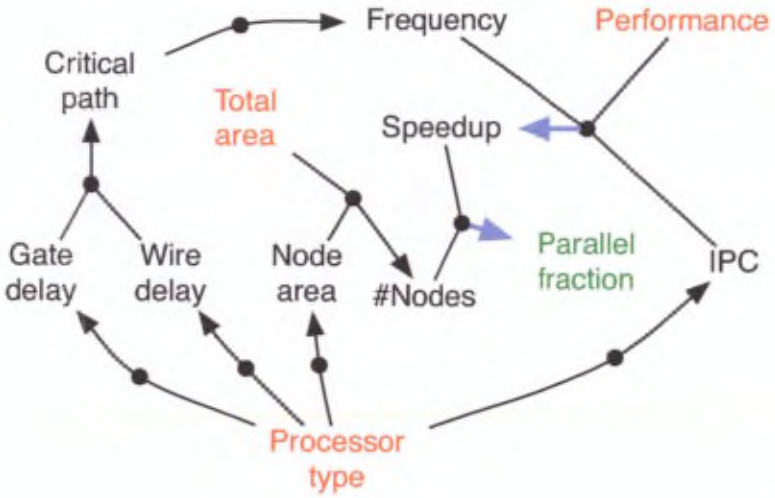

Figure 3.5: Modification of the relations (in blue) orientation to express the parallel fraction as an output and the computation performances as an input

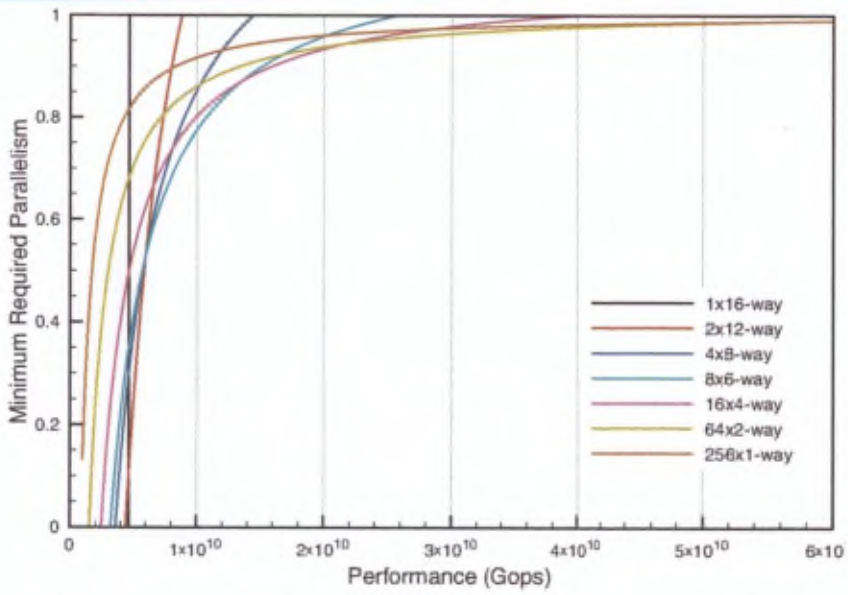

Figure 3.6: Minimum required parallelism VS computation performance for the different competing architectures

Interpretation of the results The graph plotted in Fig.3.6 represents the minimum application parallelism required to meet a given level of computation performances (number of instructions issued per second) for each of the candidate architecture. Based on this graph a user can easily choose a performance level and deduce the workload parallel fraction for which the architecture candidates can meet these requirements. In the case of 10 Gops/s for instance, all architectures are able to deliver that level of performances except the 1x16-way and the 2x12-way candidates. As the level of performances increases, more and more architectures give up and the minimum required parallelism keeps increasing to quickly approach 1 where only 64x2-way and 256x1-way architectures are able to deal with this level of performances.

If we further try to further interpret the shape of the resulting graph, we can separate each curve in three different zones as depicted in Fig.3.7:

- *Zone I* represents the part of the curve where a performance level smaller than A is always met whatever the level of parallelism. This means that even one single processor of this type has a bigger computation power than what is required by the designer: this is an opportunity for reducing the frequency of the processors to save energy.
- *Zone II* is defined by a required computation performance level ranging from A to B: this zone represents the curve in itself. As the performance level increases, the minimum required parallelism also increases until it reaches 1. It is interesting to mention that the parallelism fraction defined by this curve exactly corresponds to the effective value of the required computation performance while the points above this curve offer a higher performance level than the one required: this is why we call the Y-axis the *minimum required parallelism*.
- *Zone III* represents the part of the curve where the computation performances level cannot be met by the architecture whatever the value of the parallel fraction. This means that another architecture with more parallelism has to be selected to meet the performances.

These three zones can be easily observed on Fig.3.6 for the different curves representing the candidate architectures except for the mono-processor whose zone II is reduced to a simple vertical asymptote because its computation performances don't depend on the workload parallel fraction value.

Some important mathematical considerations We have however to mention two important things related to the mathematical inversion of the relations:

- Since we generate workload parallel fraction values for all the computation performances values ranging from 0 to 60 Gops/s for *each candidate*

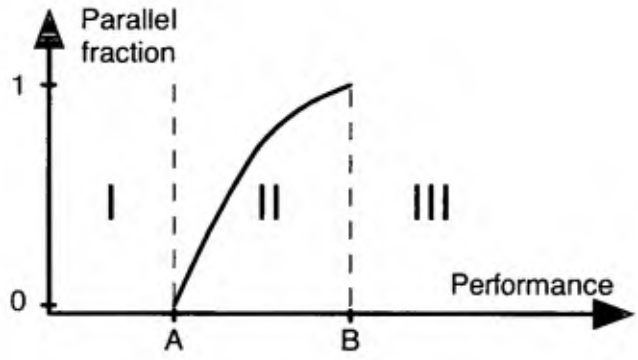

Figure 3.7: Representation of the three zones of the minimum parallelism curves represented in Fig.3.6

architecture, we sometimes get for certain curves parallel fraction smaller than 0 and bigger than 1. This is actually understandable because nothing prevents the inverted mathematical relations from generating non realistic output values for input values ranges that were originally not explored: it's up to the user to interpret the results with a critical eye and not to blindly generate results and take them for granted. Therefore we simply discarded all parallel fraction values outside the 0-1 range by limiting the Y-axis values drawn on the plot.

- For the mono-processor, the inverted Amdhal's law relation represented by Eq.3.6 leads to several non-valid solutions. Indeed when the number of nodes is equal to 1, the denominator equals 0 which leads to an infinite value: this is due the fact that the performance is independent from the parallel fraction value. In a same way, when the speedup exactly equals 1 and the numerator equals 0, the mathematical evaluation leads to an undetermined value[5]. To deal with that problem (see Sec.2.6.2 for more details), Yeti detects a zero division and tags the solution as invalid: each invalid solution is not reported in the plot file to avoid erratic and non significant values from being plotted in the resulting graph.

Although the user has to carefully interpret the results and exclude value ranges that make no physical sense, it must be understood that the mathematical considerations are not harmful at all to the good working of our framework. Indeed several implemented mechanisms invisible to the user take care of avoiding run-time mathematical errors and discarding invalid solutions by removing them from the resulting data set.

[5]Due that mathematical problem, we needed to add the plot for the single node which is just a vertical line.

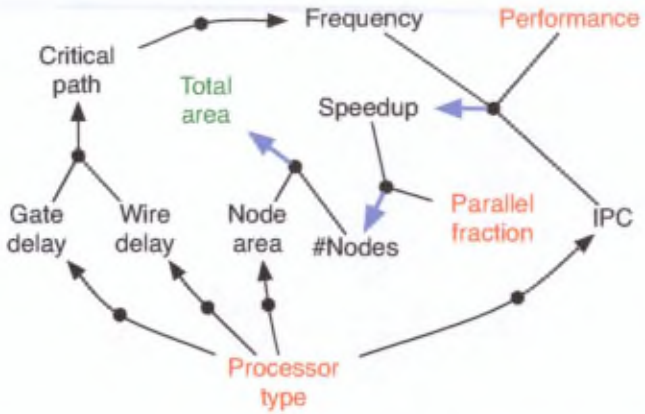

Figure 3.8: Orientation of Codrescu's relations expressing the area as a function of the processor type, workload parallel fraction and computation performances

Relaxing the area constraint

Previous experiments based on Codrescu's model elected the architecture candidates based on a strict constraint: the total area was supposed to remain constant in order to offer a fair comparison between the different candidates. Therefore the number of cores of each architecture had been selected so that multiplied by the area of the particular processor type that it instantiated, it always equalled the size of a single 16-way processor. In this case study, we remove that area constraint so that we don't impose the number of cores for each architecture: instead we would like to determine the minimum area (i.e. the minimum number of cores) required to meet a given performance constraint. This problem can again be easily solved with very few effort using the reversibility feature of Yeti.

Setting up the experiment Seen from point of view of Yeti, this problem leads to the new behaviour orientation depicted in Fig.3.8. Compared to the previous case, the parallel fraction Par and the computation performances $Perf$ are now both inputs while the only output that remains is the total area.

Again very few changes are needed in the XML input simulation files to move from the previous model orientation to our current example: Fig.3.8 tells us that only the Amdhal's law relation must be modified and expressed as a function of the number of nodes instead of the speedup which results in Eq.3.7.

$$\#Nodes = \left\lceil \frac{Sp * Par}{1 - Sp * (1 - Par)} \right\rceil \tag{3.7}$$

This equation is interesting because we can see the presence of a ceil function never mentioned in the list of the available basic operations (see Sec.A.1.2). Indeed the inversion of the Amdhal's law allows us to calculate the number of nodes instantiated from a particular processor type required to reach a precise performance level defined by the user: due to the analytical nature of the relations, nothing prevents the number of nodes from having floating point values to meet that exact level of performances. However it is obvious that a non-integer number of nodes makes no sense so that we decided to apply a ceil function on the result of the Amdhal's law inversion formula. Unfortunately Yeti didn't offer such a function among its operation set since we didn't anticipate its need so that we had to add it for the purpose of our experiment. By following the procedure described in Sec.2.6.2, it took only twenty minutes to get our new operation[6] working from code modification to successful testing.

Interpretation of the results Based on this setup, we decided to carry out several experiments to observe how the total area depends on the required performance level, the parallel fraction and the processor type. Since we have three different parameters, we represent in a single graph the total area versus the parallel fraction for a fixed performance value; each processor type is then represented by a single curve on this plot. Setting up this simulation is made much more easy thanks to the use of XML scripting enabling the definition in a dedicated file of all the separate actions to be performed to get the results plots.

Fig.3.9 represents the total area versus the workload parallelism fraction for a performance level of 0.8 Gops. As we can see all the curves representing the different processor types are horizontal meaning that the area required is independent from the parallel fraction and that the obvious choice is to take the smallest 1-way processor. Actually the required area corresponds in this case to the use of a single processor whatever the type which means that a single processor is sufficiently powerful to provide the required performance on its own for a fully sequential workload (with a parallel fraction equal to 0). When the parallel fraction increases, the single processor continues to perform both sequential and parallel parts of the workload in a sequential manner since single processor architectures offer no thread-level parallelism to benefit from a potential parallel speedup: the performances thus remains the same. Interestingly we could estimate the maximum performance level for which all the processor types only require one core to be instantiated on the platform to meet this performance requirement. Indeed this limit will be reached when the less powerful processor exactly issues as much instructions per second on

[6]Aside from the ceil function, we added by the way the floor function as it may also be useful in the future. Both operations are unary: the ceil function is represented by the ⌉ symbol while the floor function is ⌊.

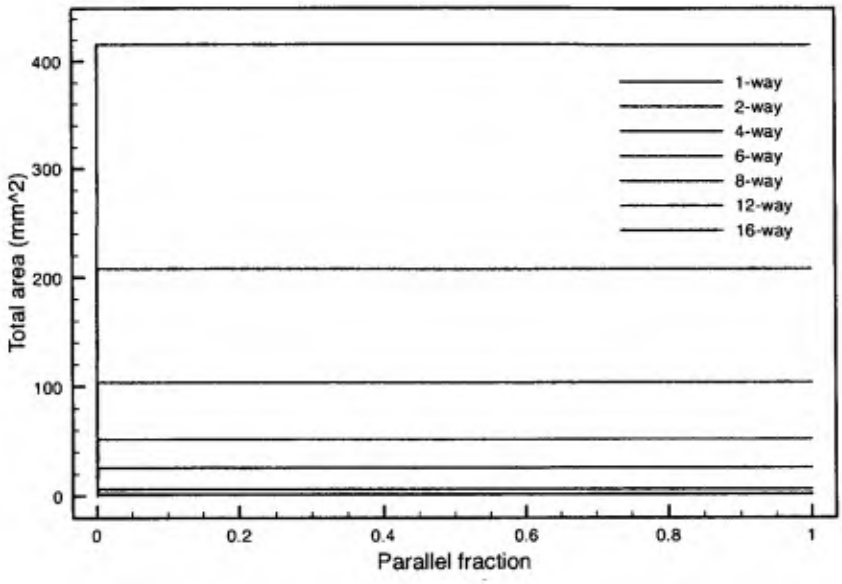

Figure 3.9: Total area VS parallel fraction for different processors architectures meeting a 0.8 Gops performance constraint

its own as what is required by the performance constraint: in other words, this comes to the performances evaluation of a single 1-way processor (speedup equal to 1) which gives according to Eq.3.1 a value of 0.868 Gops. In Fig.3.10, we performed the same experiment for a computation performance value of 0.9 Gops where we see a clear modification of the curve for the 1-way processor (only the 0-0,1 range was represented to make it more clear): this change of the curve between 0.8 Gops and 0.9 Gops is consistent with our theoretical predetermination of 0.868 Gops. The shape of the new curve, surprising at first sight, will be further explained and justified hereunder.

Fig.3.11 shows the required area to meet a 6 Gops performance constraint for the different processor types. As we can see, most of the previous horizontal lines have turned into curves showing a vertical asymptote at a certain workload parallel fraction value and quickly decreasing for higher values of this parallel fraction. As explained before, this curve shape modification can be observed as soon as the performance constraint value exceeds the number of instructions issued per second by a single X-way processor represented by $Perf_{S,X}$. Exceeding this performance threshold value entails the use of additional instances of this processor to compensate the lack of instruction level parallelism with thread level parallelism. In other words, to achieve the performance constraint set to $Perf_{Req}$ instructions issued per second, we will have one part of the code executing sequentially and the other in parallel in

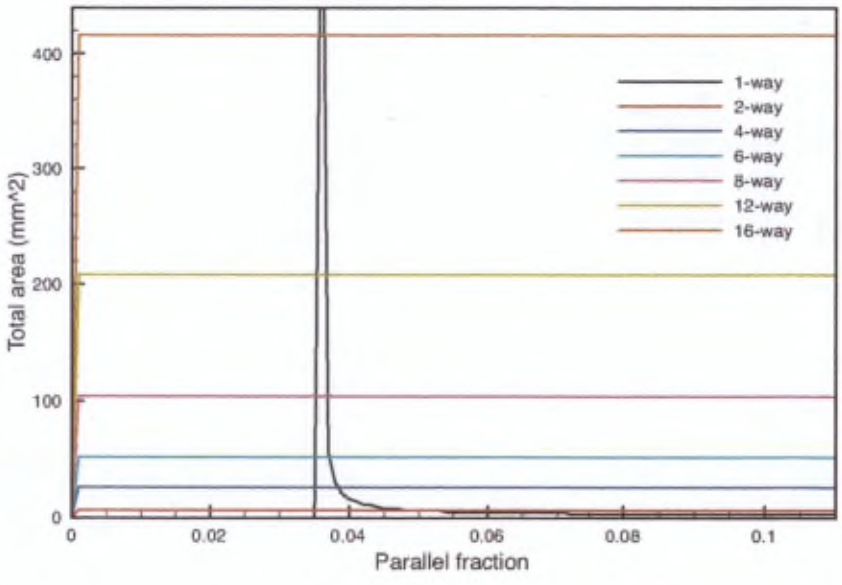

Figure 3.10: Total area VS parallel fraction for different processors architectures meeting a 0.9 Gops performance constraint

proportions described by Eq.3.8 (which is nothing more than Amdhal's law rewritten).

$$Perf_{Req} = Perf_{Seq} + Perf_{Par} = \frac{Perf_{S,X}}{1 - Par} + \frac{Perf_{S,X} * Nodes}{Par} \tag{3.8}$$

In this equation, $Perf_{Req}$ can also be seen as the number of instructions that need to be issued in one second and the two terms of the equation as the number of instructions respectively issued sequentially and in parallel. However this equation only holds if the sequential workload can be executed in less than one second in order to leave some time for the parallel workload to execute: a statement that is equivalent to say that the number of issued instructions from the parallel workload must be positive. This condition can thus be written as Eq.3.9 which leads to a limit parallel fraction Par_{Limit} expressed by Eq.3.10.

$$Perf_{Par} = Perf_{Req} - Perf_{Seq} > 0 \tag{3.9}$$

$$Par_{Limit} > 1 - \frac{Perf_{S,X}}{Perf_{Req}} \tag{3.10}$$

Below this parallel fraction value Par_{Limit}, the time required by the sequential workload to execute exceeds the total execution time required to meet the performances constraints $Perf_{Req}$: this means that this type of processor simply cannot keep up with such a high performance level value for so small workload

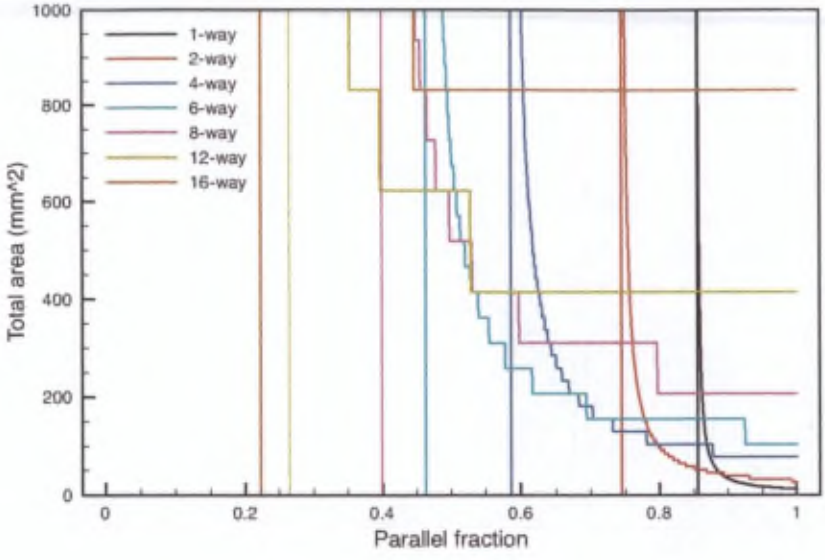

Figure 3.11: Total area VS parallel fraction for different processors architectures meeting a 6 Gops performance constraint

parallel fractions. Based on this minimum parallel fraction value, we are thus able to explain the different zones of the area curves as presented in Fig.3.12:

- *Zone I* is delimited by the range of parallel fraction values that are smaller than Par_{limit}; the required performances $Perf_{Req}$ cannot be met.
- At a parallel fraction value of Par_{limit}, we can see a vertical asymptote meaning that the area required to meet the desired performances tends to an infinite value. Indeed for a such a parallel fraction value, the sequential workload has completed its execution in the exact time that was allocated to the total execution meaning that the remaining parallel workload has no time left to execute its instructions. Mathematically, the only solution to execute a finite amount of instructions in zero time is to allocate an infinite amount of resources to parallelize their execution leading to an infinite area.
- *Zone II* is delimited by the range of parallel fraction values that are bigger than Par_{limit}; the parallel workload now has a strictly positive time left to execute its instructions. It is interesting to see that the area (hence the number of cores) required to meet the desired performances quickly decreases when we go towards higher values of the parallel fraction; this makes parallel fraction close to Par_{limit} very unfavourable.

The limit parallel fraction Par_{limit} determination method can be confirmed by

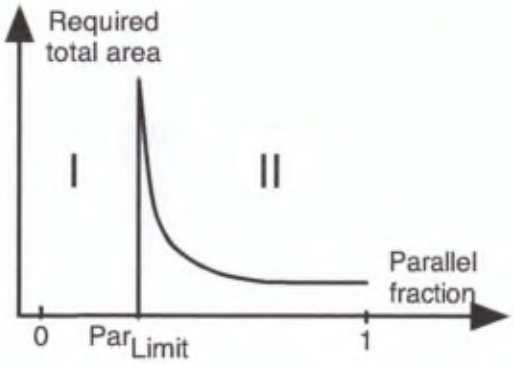

Figure 3.12: Representation of the different zones composing a curve of the area VS parallelism fraction

Processor type	1-way	2-way	4-way	6-way	8-way	12-way	16-way
Parallelism	0,855	0,744	0,584	0,464	0,397	0,264	0,221

Table 3.3: Value of the smallest workload parallel fraction required to meet computation performances requirements for different types of processors

comparing the asymptote abscissa values of the 6 Gops area graph in Fig.3.11 with Table 3.3 calculated thanks to Eq.3.10 for each processor type. This whole interpretation proves us that Yeti has efficiently and correctly enabled the inversion of model to solve this new problem. Furthermore it has only required one mathematical relation to be manually reverted and the framework did all the remaining operations including automatic plotting of the results.

Once the level of desired level of computation performances has been defined by the user, the graphs can be generated in one single run and be used to determine the architecture that minimizes the total area for a given parallel fraction. Fig.3.11 represents such an example for a 6 Gops computation performance constraint; as we can see there are many intersections between the different curves highlighting the high dependence of the parallel fraction on the total area and the interest of using such a graph to determine the best solution.

3.3 Case Study 2: stage delay modeling and applications

3.3.1 Introduction

The previous section demonstrated how it is possible to reproduce and extend results from the literature by using the reversibility feature of Yeti. In this

case study, we will instead focus on the model/input parameter sensitivity capabilities of our framework and therefore build a model from scratch, study it and show how we can optimize output parameters values thanks to Yeti. As an illustration, we have chosen to model the delay of a critical path of a chip, measure the impact of wire/gat sizing on clock frequency and optimize data bandwidth. We will also compare our results with the literature and use the model we built to explain the origin of surprising discrepancies found among well-known papers.

For this case study, we have chosen to model the delay of a critical path of a chip of area A as represented in Fig.3.14. This delay allows us to determine the maximum clock frequency and the bandwidth which are very valuable information from a designer perspective. The critical path will actually be represented by N stages of same length where each stage is composed out of a gate driving a wire to transmit the signal to the other side of this interconnect. This stage delay model actually involves many technological related parameters which is a good example to illustrate how Yeti can be used to quickly bridge the gap between system-level and technological abstraction levels (see Sec.1.2.2).

Since the critical path is composed out of several stages, we will first have to study their modeling in details. As depicted in Fig.3.13, we can represent and model a stage at three abstraction levels:

- On top of the hierarchy, we have a model defining the delay based on the electrical elements of the circuit representing the stage.
- Beneath the delay model, we can define a stage as a gate connected to wire transmitting the information to several gates at the other end. These drivers and wires can in turn be represented by electrical scheme of resistances and capacitors.
- Each electrical element of the stage can finally be modeled by technological and gate/wiring sizing parameters.

Before discussing the delay model and its impact on performances, we will first address the question of the two bottom layers which are required to feed our model with realistic values.

3.3.2 Representation of a stage

Electrical representation

Modeling a stage is not such an easy task due to the complexity of the layout (particularly in full-custom designs): signal paths have different length distributions[6] and each gate may drive from 1 to several gates (the fan-out)[7]. However we can use a simple yet realistic model to represent a stage

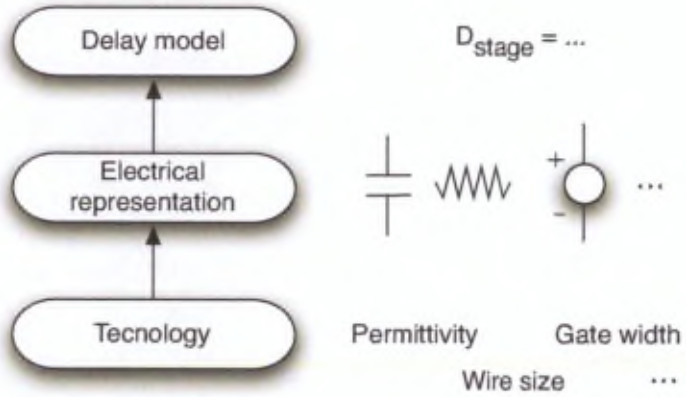

Figure 3.13: Hierarchical representation of a stage model in three layers: delay, electrical scheme and technology

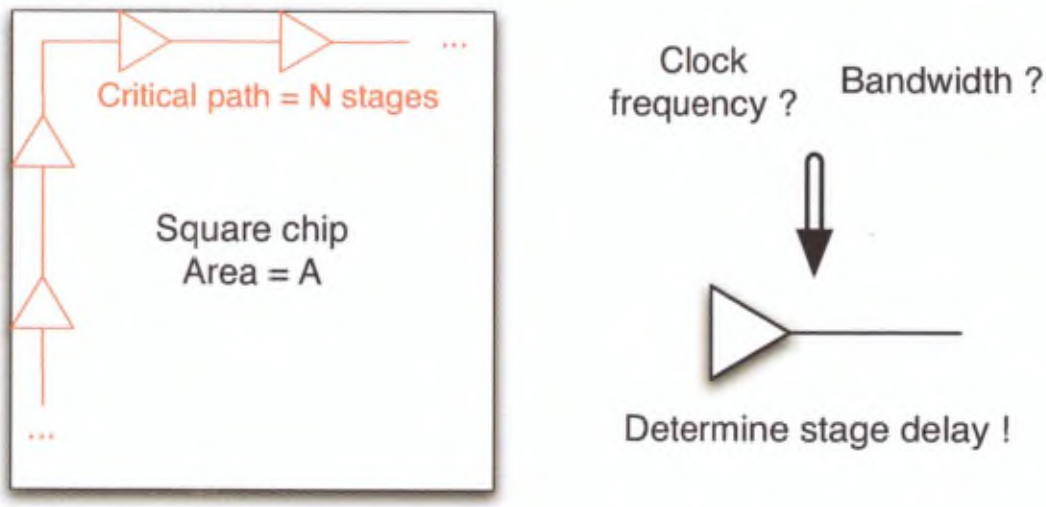

Figure 3.14: Representation of a square chip of area A and its N stages

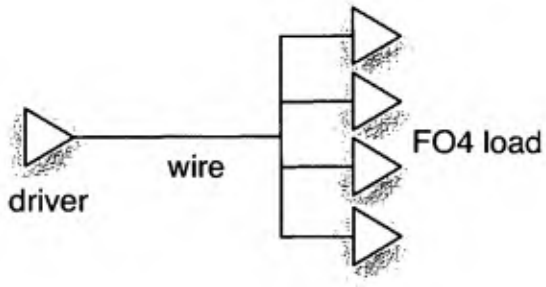

Figure 3.15: Logic representation of a stage composed out of a gate driving a 1 FO4 load through a wire

as depicted in Fig.3.15: a gate is driving a wire connected at the end to a 1 FO4[7] load[10].

To model the delay, we first have to turn this representation into an electrical model as shown in Fig.3.16 where we can see the following electrical elements:

- V_{tr} is the voltage source representing the driver switching by a voltage step
- R_{gate} is the output resistance of the driving gate
- C_{gate} is the output capacitance of the driving gate; physically this element represents the diffusion capacitance created by to the overlap of the MOS channel with the drain/source
- r_{wire} is the distributed resistance of the wire of length L_{wire}
- c_{wire} is the distributed capacitance of the wire of length L_{wire}
- C_{load} is the total load capacitance of the wire (1 FO4 in our case).

Technological representation

Now that we have defined the electrical representation of the stage, we still need to link these different elements to technological related parameters.

Gate geometry and modeling Let us first have a look at the structure of a transistor depicted in Fig.3.17. From this representation we can identify the diffusion capacitance C_{diff} resulting from the overlap between the drain/source

[7] The fan-out 4 (FO4) refers to a load of 4 minimum sized inverters that a gate has to drive. This unit is commonly used to represent gate delays because it has been shown that any CMOS gate delay is roughly the same value when expressed in FO4 number regardless of the technology node: since this metric is independent from the technology, it is very convenient to represent delays[8]. Furthermore some heuristics have shown that 1 FO4 is a good average load value for drivers in a critical path[9].

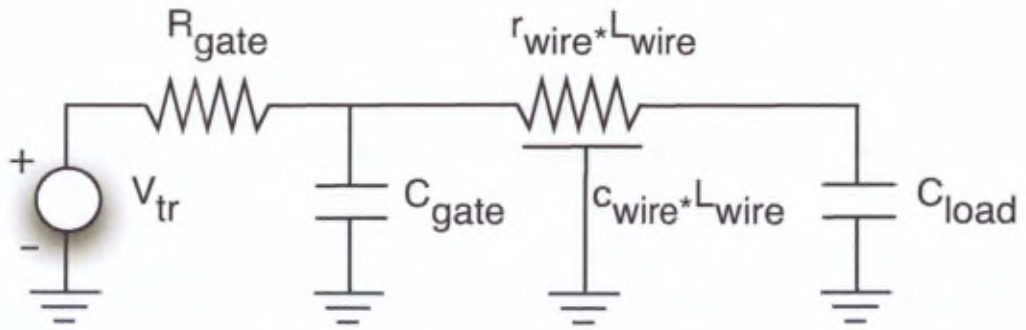

Figure 3.16: Electrical representation of a stage

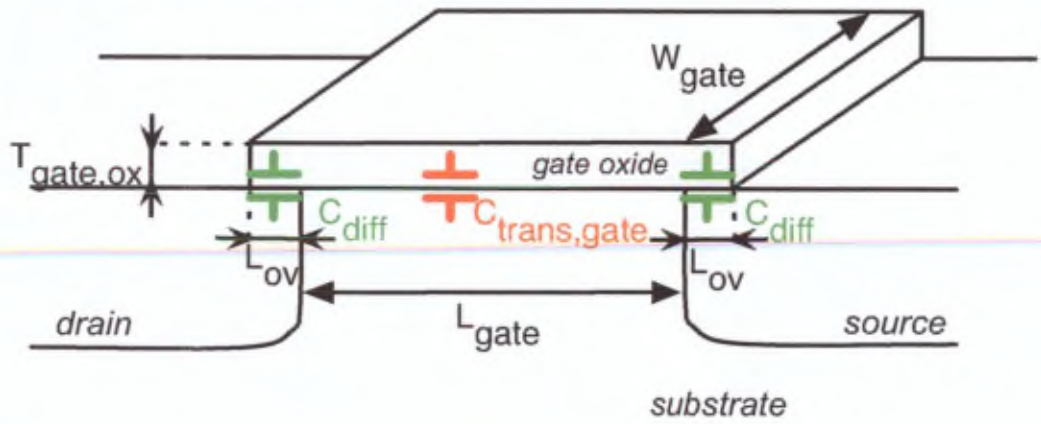

Figure 3.17: Geometrical representation of a transistor

and the transistor gate[8] capacitance $C_{trans,gate}$. Aside from the capacitances, we can also observe the different geometrical characteristics represented on this figure:

- W_{gate} is the width of the transistor gate
- L_{gate} is the effective length of the transistor gate
- L_{ov} represents the length of the overlap zone between the drain/source and the transistor gate: this overlap is a result of the lithography imperfection

[8] *Gate* is a very ambiguous term however widely used in microelectronics: it can either refer to a logic gate or to the conducting layer allowing to control the transistor channel. To avoid any misunderstanding, we will simply use the term *gate* in the first acceptance of the term and call it a *transistor gate* when we refer to the second meaning of it: we will stick to that convention in the entire text.

- $T_{gate,ox}$ is the thickness of the transistor gate oxide.

Based on these different geometrical parameters, we are able to define the values of the three elements of the electrical gate R_{gate}, C_{gate} and C_{load}.

C_{gate} is represented by Eq.3.11 where ϵ_0 is the vacuum permittivity and $\epsilon_{r,gate}$ the transistor gate relative permittivity.

$$C_{gate} = \epsilon_0 * \epsilon_{r,gate} * \frac{W_{gate} * 2 * L_{ov}}{T_{gate,ox}} \tag{3.11}$$

C_{load} is four times the transistor gate input capacitance as represented by Eq.3.12.

$$C_{load} = 4 * C_{trans,gate} = 4 * \epsilon_0 * \epsilon_{r,gate} * \frac{W_{gate} * (L_{gate} - 2 * L_{ov})}{T_{gate,ox}} \tag{3.12}$$

R_{gate}, expressed by Eq.3.13, defines the average value of the gate output resistance when it is in ON state[11]. This equation uses additional parameters like V_{dd} the supply voltage, $I_{d,sat}$ the drain current of the transistor in the saturation zone and λ (the inverse of Early voltage). $I_{d,sat}$ can be calculated using Eq.3.14 where μ_n is the electron mobility, C_{ox} the silicon oxide capacitance and V_t the threshold voltage of the transistor.

$$R_{gate} = \frac{3 * V_{dd}}{4 * I_{d,sat}} * (1 - \frac{5}{6} * \lambda * V_{dd}) \tag{3.13}$$

$$I_{d,sat} = \frac{W_{gate}}{L_{gate}} * \mu_n * C_{ox} * \frac{(V_{dd} - Vt)^2}{2} \tag{3.14}$$

Wire geometry and modeling To represent the wire and its surrounding wires, we can use the model introduced by Bohr [12] and depicted in Fig.3.18. The modeled wire is surrounded by four other wires: two parallel wires on the same layer that will create a capacitance called the side-to-side capacitance C_{side} and two orthogonal wires at the upper and lower layer that will create the vertical capacitance C_{vert}. Only plate-to-plate capacitances are taken into account in this model, fringing capacitances[9] are neglected[10]. This representation can be seen as a worst case wire organization since it assumes that the neighbour wires have the same length as the modeled wire creating the maximum parallel surface hence the highest capacitance value. This hypothesis however holds for dense interconnect schemes and is widely used in the literature[14][15][16][9].

[9] Fringing capacitances result from electrical fields propagating between conductor plates that are parallel.

[10] For more accurate wire capacitance estimation, we can use programs extracting these values from 3D wire descriptions such as FastCap[13]. They have the drawback of delivering models that are very specific to the geometry of the wire and that are less easy to manipulate than Bohr's model expression that gives a reasonable approximation of the wire capacitance: this is why we preferred the last one.

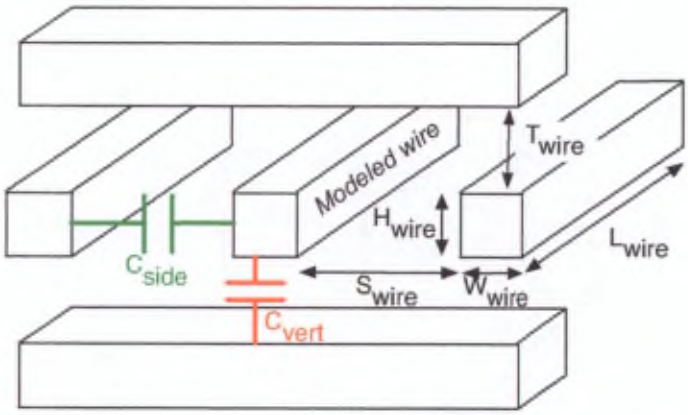

Figure 3.18: Geometrical representation of a wire and its neighbours

Different geometrical characteristics can be observed on Fig.3.18 (we suppose that all represented wires are in the same tier and thus share the same geometrical dimensions):

- L_{wire} is the length of the modeled wire
- H_{wire} is the height of the wire
- W_{wire} is the width of the wire
- S_{wire} is the spacing between two neighbour wires, the sum of the spacing and the width is often referred to as the *pitch*
- T_{wire} is the height separating two successive layers

Based on these parameters, we are now able to determine the total wire capacitance and resistance.

C_{wire}, represented in Eq.3.15, is the contribution of the four surrounding capacitances.

$$C_{wire} = 2 * (C_{side} + C_{vert}) \tag{3.15}$$

The side-to-side capacitance can be calculated by Eq.3.16 where ϵ_{wire} is the wire relative permittivity optimized to be as small as possible in order to minimize the impact of capacitive effects.

$$C_{side} = \epsilon_{wire} * \frac{H_{wire} * L_{wire}}{S_{wire}} \tag{3.16}$$

The vertical capacitance C_{vert} is expressed by Eq.3.17

$$C_{vert} = \epsilon_{wire} * \frac{W_{wire} * L_{wire}}{T_{wire}} \tag{3.17}$$

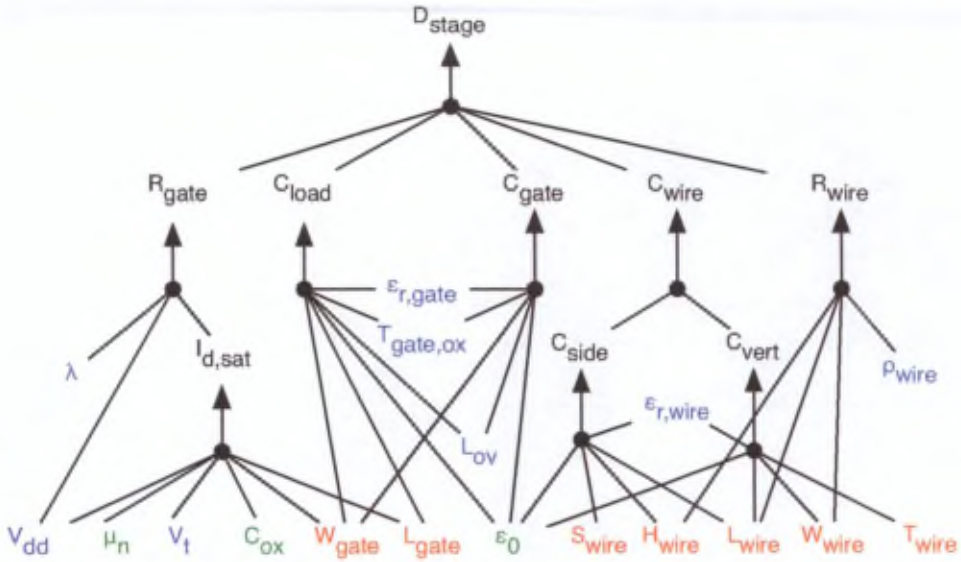

Figure 3.19: Yeti behaviour corresponding to the modeling of a stage delay: green parameters are physical constant, blue are silicon process related and red are geometrical parameters fixed by the designer

The total wire resistance R_{wire} is given by Eq.3.18 where ρ_{wire} is the wire copper resistivity.

$$R_{wire} = r_{wire} * L_{wire} = \rho_{wire} * \frac{L_{wire}}{W_{wire} * H_{wire}} \tag{3.18}$$

3.3.3 Modeling the stage delay into Yeti

Now that we have successively defined the stage model at the logic, electrical and technological level, we end up with input parameters whose value can be easily extracted from the literature to feed our model. These different input parameters are defined in the Yeti behaviour presented in Fig.3.19 with the respective orientation of the different relations involved. Inputs are classified into three different types: green inputs are constant physical values, blue inputs define technological related parameters that are fixed by the chosen process and finally red inputs are parameters whose value (mostly related to wire and gate geometry) can be adjusted by the designer once the technology has been determined. On top of all the previous relations, we have defined the delay relation using all the electrical elements: we will now study this delay model in details.

Model sensitivity study

Motivation One could probably wonder at first sight why it is so important to evaluate the impact of different proposed models on the output parameters values. If we take a look in the literature at models used to estimate the value of stage delay, we will find a lot of results but it is often difficult to compare them due to different input parameters values and assumptions. Their impact of the chosen model on the presented results is therefore rarely estimated making papers that predict the same output very difficult to compare. As an example, huge discrepancies in wire delay estimation can be observed: while Sylvester[10] claim that a 1 mm long local wire in 100nm would have a 340 ps delay, Meindl[14] gave a prediction of 30 ps for the very same type of wire. Comparing the underlying models is thus very interesting and important to understand where such differences could come from.

To illustrate model sensitivity capabilities of Yeti, we propose an original study that consists in taking five models taken from significant contributions of the literature about stage delay modeling and comparing these five models for similar inputs. These different models were established for the 180nm tehcnology node, a few generations from today top-notch technologies. The selected papers were written from 2000 to 2002, a period where the problem of growing interconnect delays compared to fast decreasing gate delays was taken very seriously. A lot of very interesting papers from recognized authors and researchers were published at that time, making their estimations and underlying models worth comparing.

The five stage models The five different electrical models for the estimation of stage delays are the following: Rabaey[17] (see Eq.3.19), Horowitz[9] (see Eq.3.20), FED model[18] (see Eq.3.21), Meindl[14] (see Eq.3.22) and Sakurai[19] (see Eq.3.23). All these models yield for the 180 nm technology node and come with no referenced assumption nor limitation for the input parameter value range: they only rely on different approximations to estimate the very same delay value.

$$D_{Rabaey} = R_{gate} * C_{wire} + \frac{C_{wire} * R_{wire}}{2} \tag{3.19}$$

$$D_{Hor} = R_{gate} * (C_{gate} + C_{wire} + C_{load}) + R_{wire} * (C_{load} + \frac{C_{wire}}{2}) \tag{3.20}$$

$$\begin{aligned} D_{FED} = \; & 1.00724 * R_{gate} * C_{wire} + 1.00426 * R_{gate} * C_{load} + \\ & 1.12524 * C_{wire} * R_{wire} + 1.04468 * R_{wire} * C_{load} \end{aligned} \tag{3.21}$$

Parameter	$\epsilon_0[\frac{F}{m}]$	$C_ox[\frac{F}{m^2}]$	$\mu_n[\frac{m^2}{Vs}]$
Value	8.854e-12	39e-6	1

Table 3.4: Physical constant input parameters values used in the stage delay model

Parameter	$\lambda[V^{-1}]$	$V_{dd}[V]$	$V_t[V]$	$L_{ov}[m]$	$\rho_{wire}[\mu\Omega.cm]$	$\epsilon_{r,gate}$	$\epsilon_{r,wire}$
Value	1e-4	2.0	0.6	5e-9	2.2	2.3	5.0

Table 3.5: Technological related input parameters values used in the stage delay model

$$D_{Meindl} = R_{wire} * C_{wire} + 0.69 * R_{gate} * C_{wire} + 0.69 * C_{load} * (R_{wire} + R_{gate}) \quad (3.22)$$

$$D_{Sakurai} = 0.377 * R_{wire} * C_{wire} + 0.693 * (R_{gate} * C_{load} + R_{gate} * C_{wire} + R_{wire} * C_{load}) \quad (3.23)$$

Before being able to estimate the model, we first need to give a value to each input parameter of the behaviour represented in Fig.3.19. These values are represented in table 3.4 for the physical constants, in table 3.5 for the technology related parameters and finally in table 3.7 for the designer defined parameters. For this study, we decided to use the most common technological parameters values that could be found: we took all the data from the ITRS roadmap (missing information were filled in with plausible values).

Results and interpretation As a first experiment, we used Yeti to compare the different models for a wire length ranging from 0.1 mm to 1 cm: the result is depicted in Fig.3.19. At first sight, we can see that there are significant differences in the overall delay values except for Rabaey and Horowitz models that are very close to each other and only differ by a wire length independent term. To quantify the deviation between these different models, we have calculated two different metrics represented in Fig.3.21.

The green curve of Fig.3.21 represents the maximum relative error (the difference between the maximum and the minimum predicted delay value divided by the maximum value) between the different models for the different wire

Parameter [nm]	S_{wire}	H_{wire}	W_{wire}	T_{wire}	W_{gate}	L_{gate}	$T_{gate,ox}$
Value	180	180	180	360	180	180	10

Table 3.6: Design related input parameters values used in the stage delay model

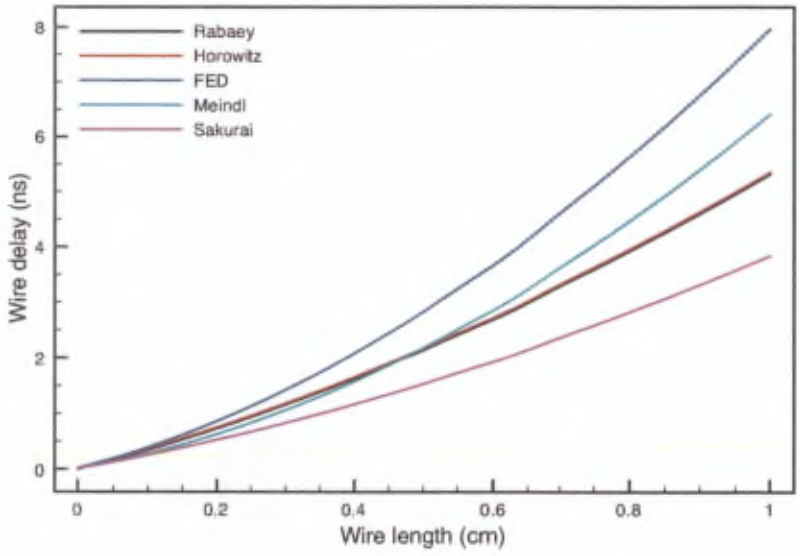

Figure 3.20: Comparison of five models estimating stage delay for a 0.1mm to 1cm wire length range

lengths. This first metrics represents *for a particular wire length* the worst error that we could do by selecting the two most diverging models.

The second metrics (red curve on Fig.3.21) defines the *relative delay value dispersion* for different wire lengths. Usually a model is compared to a reference (for instance an experiment performed to measure the values of the modeled process) enabling the comparison: in our case we compare models together so that there is no reference to compare with. Therefore this second metrics will estimate the dispersion of the different models by calculating for each particular wire length the mean value of the absolute difference between each pair of predicted delay values and by dividing it by the mean value of the predicted delay values. Since all the pairs of values are tested, it prevents this dispersion value from reaching very high values when one single point highly differs from all the others.

Looking at the resulting graph, we can see a certain correlation between the dispersion and the maximum relative error that both have very high values for small lengths, decrease quickly and then increase slowly. For very small wire lengths, the terms with a quadratic length dependence become negligible making linearly dependent and independent terms become dominating. This can be observed on Fig.3.22 (which is the same as Fig.3.20 with a zoom on smaller wire length values) where we can see that both Meindl/Sakurai and Abou/Horowitz model curves superimpose on the graph because they have the

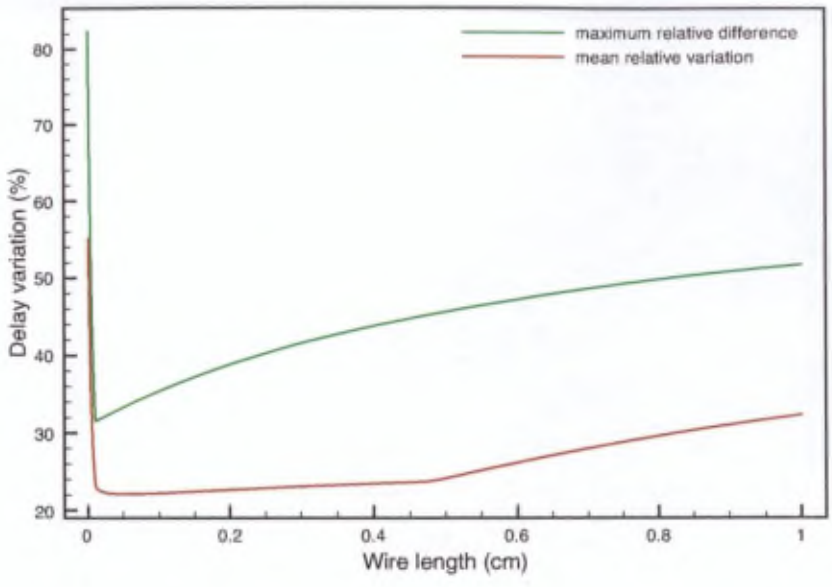

Figure 3.21: Dispersion and maximum error for the five models estimating stage delay

same coefficient for the term linearly dependent of the wire length. Rabaey model however gives very different delay estimations since it ignores the effect of the load capacitance which becomes important at such small lengths: this explains why the error is huge for small wire length values. Around 0.1 mm, Rabaey intersects the other models making the relative difference much smaller between the different models and also leading to smaller dispersions values. The dispersion value then increases until 0.475 cm where we see an inflection point resulting from another model curve intersection on graph 3.20.

For higher values of the wire lengths, we see that all model curves show a quadratic behaviour because the $C_{wire} * R_{wire}$ begins to dominate due to its quadratic dependence with the length: since all models use different coefficients for this term, the delay curves diverge more and more and both dispersion and maximum relative error values continue to grow.

If we summarize our observations, we can see that the five compared models show significant differences: the mean value of the maximum relative error equals 44,6% while the mean dispersion equals 25,9%. This difference between the compared models partly explains the tenfold difference for the predicted wire delay presented as illustration example of this section. It must however be made clear that none of these models can be considered as more accurate than the others since there are no experimental measures that the resulting prediction could be compared to. In the next part of this chapter, we will

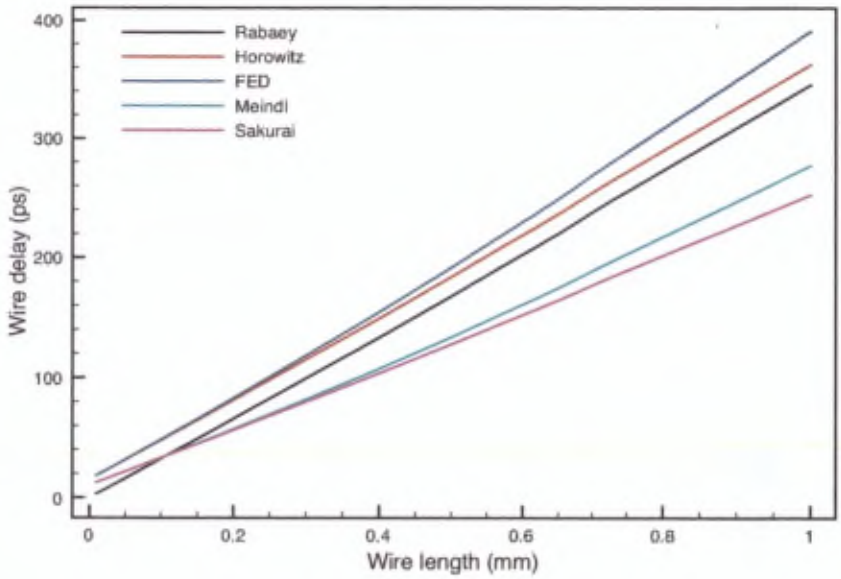

Figure 3.22: Comparison of five models estimating stage delay for a 0.01mm to 1mm wire length range

use Horowitz model delay prediction not because this model is more or less accurate but simply because it takes many interactions between the different electrical components of the stage model into account.

Now that we have examined how the model influences the stage delay output value, let us evaluate quantitatively the impact of input parameter variations on this delay.

Input sensitivity analysis

To measure the sensitivity of a model output to input variations, we can use the *constraints evaluation* feature of Yeti (see Sec.2.5.1): we define upper and lower bounds for the input values and observe the resulting values for the outputs.

We performed an experiment to measure the ±25% input sensitivity of the major geometrical parameters of the stage model: each of these inputs have been set one after the other to a lower and upper bound of respectively 75% and 125% of the nominal value. The results of this experiment are summarized in Table3.8. The nominal values of these input parameters are presented in Table3.7 while other input parameters keep their previous values of Table 3.5.

Input parameter [nm]	S_{wire}	H_{wire}	W_{wire}	T_{wire}	W_{gate}	L_{gate}	L_{wire}
Nominal value	360	360	360	720	900	360	1e+6

Table 3.7: Design related input parameters values used for the ±25% input sensitivity study

Parameter	Bound	Model				
		Rabaey	Horowitz	Abou-Seido	Meindl	Sakurai
T_{wire}	Lower	11,1%	10,2%	10,1%	10,2%	10,2%
	Upper	-6,7%	-6,1%	-6,2%	-6,1%	-6,1%
S_{wire}	Lower	22,2%	20,3%	20,3%	20,3%	20,3%
	Upper	-13,3%	-12,2%	-12,2%	-12,2%	-12,2%
H_{wire}	Lower	-16,4%	-15,0%	-14,8%	-14,7%	-15,0%
	Upper	16,4%	15,0%	14,8%	14,7%	15,0%
W_{wire}	Lower	-8,1%	-7,4%	-7,1%	-7,0%	-7,3%
	Upper	8,2%	7,4%	7,3%	7,2%	7,4%
W_{gate}	Lower	33,1%	30,2%	29,9%	29,8%	30,2%
	Upper	-19,8%	-18,1%	-17,9%	-17,9%	-18,1%
L_{gate}	Lower	-24,8%	-26,4%	-26,2%	-26,1%	-26,4%
	Upper	24,8%	27,5%	27,2%	27,1%	27,5%

Table 3.8: Results of the input sensitivity study of gate geometry related parameters performed for the stage delay model

Analyzing the results from this input sensitivity leads to some interesting conclusions. First we can see that the impact of an input variation on the stage delay can be very different from one parameter to another. While some parameters lead to 6% delay variation, other make this value exceed 30%: averaged over all the input parameters, a 25% value variation leads to a 16,4% stage delay variation. In this case, the output variation is thus smaller than the input parameter value variation that created it but remains close to it: this means that input variations have a significant impact on output values. Combining a simultaneous 25% variation of all these inputs parameters dramatically changes the resulting stage delay value: between the maximum and the minimum, the delay is multiplied by a factor 7.06.

Secondly it can be observed that the output value variations may be symmetrical with a positive or negative 25% input value variation (case of H_{wire}) or not (case of S_{wire}). This can be confirmed if we draw for these two parameters the stage delay versus an input parameter sweep within the -25%/+25% range around the nominal value. The first graph (Fig.3.23) shows the delay versus the wire height: as we can see the dependence is linear for all the models and the angular coefficient of the line doesn't change much from one model to another. The second graph (Fig.3.24) representing the wire spacing shows instead an obvious non-linearity: this results from the wire capacitance which is the sum of the side-to-side and the vertical capacitance. When the wire

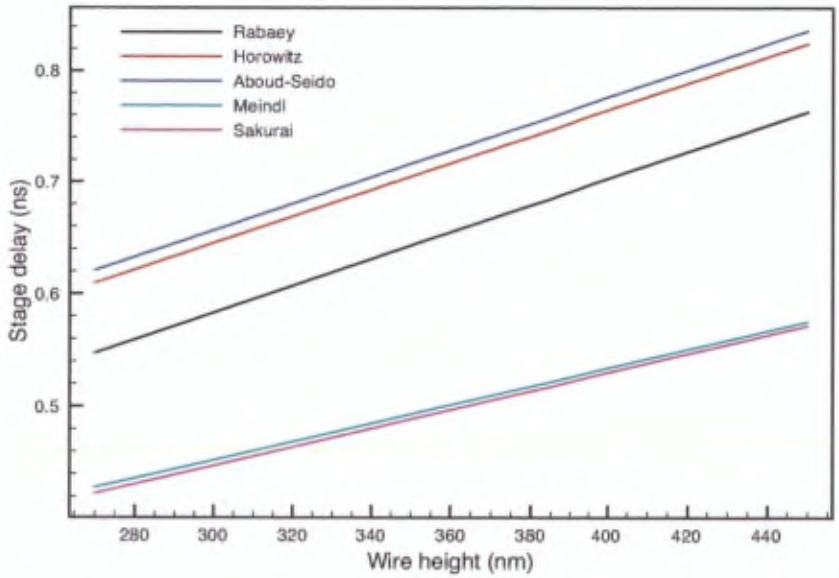

Figure 3.23: Stage delay VS wire height for a -25%/+25% range around its nominal value

spacing increases, the side-to-side decreases as an hyperbolic function while the vertical capacitance remains constant: this leads to a $A + \frac{B}{S_{wire}}$ function form that can be observed on the graph.

As a conclusion, we can see that the stage delay is highly sensitive to both the model and value variations of the numerous input parameters. If we combine the -25%/+25% with the model sensitivity, we even reach a multiplying factor of 10.31 for the difference which is very close to the tenfold difference that we mentioned in our first introducing example showing the wire delay prediction discrepancy. The same order of magnitude for the predicted and initial difference demonstrates that the combination of both model and input uncertainty is enough to explain this very surprising discrepancy at first sight. We thus recommend to be careful with papers delivering results relying on non-explicit models without their assumptions, with no restriction on the input parameters values range and some missing input parameters values. These considerations and more are discussed in one of our paper dedicated to model classification and comparison[20].

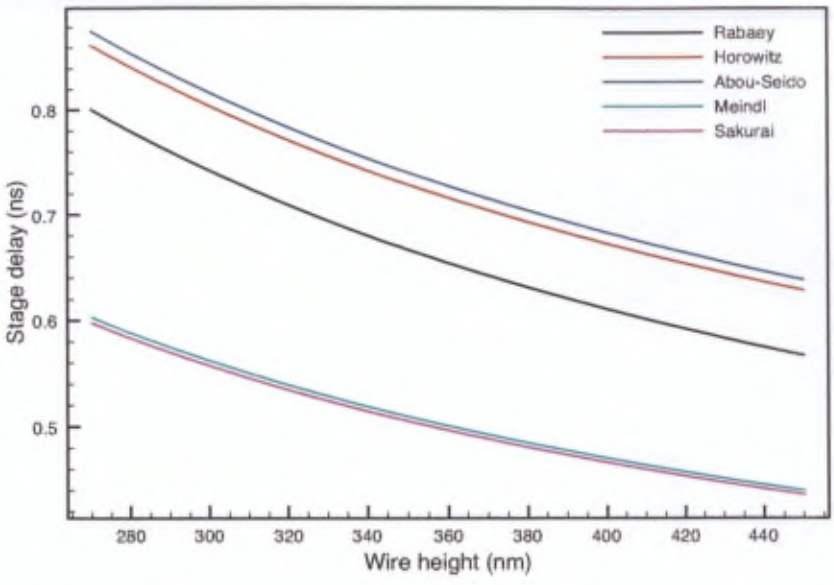

Figure 3.24: Stage delay VS wire spacing for a -25%/+25% range around its nominal value

3.3.4 Experiments with the stage delay model

Introduction

After discussing stage delay models and selecting one, we will now apply this model to the study of a chip frequency and bandwidth and perform experiments to draw original conclusions about:

- The optimum number of driving gates in a critical path
- An improved and more realistic definition of the wire bandwidth and its optimization
- The objective impact of Miller effect on delay uncertainty

In order to perform these different experiments, we first need to add some models to move from a single stage representation to a whole chip representation as it was presented in Fig.3.14. The different relations that were added to our previous Yeti model (Fig.3.19) are represented in Fig.3.25 with their respective orientations. To represent a global communication link we use the corner-to-corner length L_{path} of the chip of area $Area$ (Eq.3.24) and divided it into N stages of equal length L_{stage} (Eq.3.25). Each stage delay is evaluated using our previously discussed stage representation (Eq.3.20) so that the resulting path delay is simply obtained by multiplying the stage delay by the

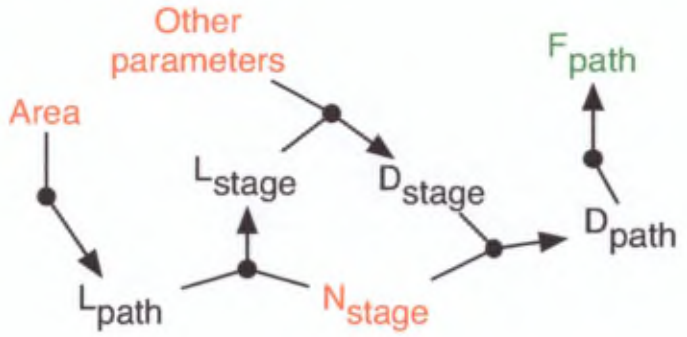

Figure 3.25: Relations representing a chip maximum frequency based on a stage delay model

number of stages (Eq.3.26). The path frequency F_{path} is obtained by inverting the path delay[11] as expressed by Eq.3.27.

$$L_{path} = 2 * \sqrt{Area} \tag{3.24}$$

$$L_{stage} = \frac{L_{path}}{N_{stage}} \tag{3.25}$$

$$D_{path} = N_{stages} * D_{stage} \tag{3.26}$$

$$F_{path} = \frac{1}{D_{path}} \tag{3.27}$$

In the previous section we compared five different stage models using close to minimum sized values for the wire geometrical parameters: this choice was appropriate for a relative model evaluation. However if we want to draw results that can be quantitatively compared with the literature, we need to use a more realistic wire sizing scheme. Different strategies are commonly used for wire sizing depending on the length of the wires: [16] defines four different sizings depending on the routing hierarchy (local, semi-global, half-fat wiring and fat-wiring) for the 50nm technology node. We used this wire planning and adapted it to 180nm by selecting only the two extreme wire sizing policies:

- *Local wires* connect gates in very small regions (50k gates blocks[12]) leading

[11]The different stage delays that we discussed in Sec.3.3.3 are 0% → 50% delays meaning that they represent the period of time required for the voltage on the load to reach half the step voltage of the source. This is however not sufficient for the output gate to switch so that we still need to wait an extra time to have the output voltage stabilized at the right value. Inverting this delay will thus not directly lead to the frequency but to a value proportional to it. Since we try to optimize the frequency (or any metric including it in all our further experiments), this will not change our results: only the absolute frequency value would require a correction before being exploitable.

[12]This 50k gate block size refers to FSB[21](Functional System Blocks) that are simple blocks encapsulating basic functions whose complexity is not believed to grow with technology. More complex systems should use more of these interconnected blocks rather than increasing their size so that the assumption of a 50k gate complexity for local wires makes sense[22].

Parameter [nm]	S_{wire}	H_{wire}	W_{wire}	T_{wire}
Value	180	540	360	216

Table 3.9: Input parameters for the local wire sizing in 180 nm

Parameter [nm]	S_{wire}	H_{wire}	W_{wire}	T_{wire}
Value	1300	1500	700	800

Table 3.10: Input parameters for the global wire sizing in 180 nm

to maximum wire lengths around the millimeter. These wires are allowed to scale with technology because the wire delay increases linearly with technology while the length decreases linearly at the same time so that the wire delay remains constant from one technology to another[23]. For the 180nm node, we used the geometrical parameters described in Table 3.9 that were derived from [16].

- *Fat wires* connect regions across the whole chip and are used for global communication. Since these wires connect regions at the scale of the chip size, their length roughly remains the same over time since the chip area is pretty constant from one technology to another. Global wires length thus don't scale with technology contrarily to local wires: that's why they are referred to as *fat wires*. These wires are usually between a few and a few tenths millimeters long. The geometrical parameters values we used for our following experiments are given in Table 3.10.

In the remaining of this chapter we will use these two types of interconnect sizing as an input to our model to respectively derive results about local and global wires.

Let us now examine the first case study: the frequency of a path versus the number of stages.

Number of stages for optimal frequency

Context Given a certain chip size or wire length, it may be interesting to evaluate how the number of stages impacts the frequency of a path: in the case of global wires, this problem is often referred to as OBIS[24] (Optimal Buffer Insertion and Sizing) and can be defined as follows. For a long global wire, we know that the delay is proportional to the square of its length[13]: splitting a wire into two equal parts seems thus very profitable as it divides the delay by four. By doing so we however created another stage hence we need to add a new driving gate that also adds a delay in counterpart: if this gate delay is

[13]This square relation between wire length and delay results from the $C_{wire} * R_{wire}$ term of stage delay equation 3.20.

smaller than the wire delay gain obtained by wire splitting, this new solution is better than the previous, otherwise not. While the gate delay is constant, the marginal delay gain resulting from an additional wire splitting decreases with the number of stages (due to the square length dependency of wire delay): this entails that the overall delay gain will become negative at some point leading to an optimum number of stages (or repeaters) to minimize the total path delay.

Driver width dependency As a first experiment, we would like to compare our model with the literature to see if our predictions fit existing data. Therefore we have calculated the maximum frequency versus the number of stages for a global wire of a $750mm^2$ square chip (as suggested by [16]) which gives according to Eq.3.24 a length of $54,8mm$. We chose 1X sized inverters and used the global wire planning scheme as it was described above for all the geometrical wire parameters. The resulting graph is plotted in Fig.3.26 where we can immediately see several interesting things.

First of all, the curve exhibits, as one would have expected, a maximum value for a path composed out of 11 stages: this value can be compared with [16] predicting a 14 value instead. These values are quite similar and the difference could easily be explained by all the implicit assumptions (wire planning policy, technological parameters besides the feature size) that are made and the missing models (particularly the stage delay model). Such a difference is thus very acceptable and shows that our model is suitable and sufficiently realistic to perform further experiments for optimum repeater insertion.

Secondly we can also notice that, although there is an optimum, the curve is very flat so that the optimum frequency is only 1.2% better than a single gate driven wire. This small gain is however perfectly understandable due to the 1X sized driver (used for the sake of comparison with [16]): let us now see what happens when these repeaters are sized up.

Fig.3.27 represents the path frequency for different size of repeaters from 1 to 50. Since frequency obviously increases with repeater size, it would have been be difficult to compare and represent the different curves on the same scale. Rather than displaying the absolute frequency values, we have, for each single curve, divide all the points by the value of the first point corresponding to a single repeater driving the path. This normalization process entails that each curve will start at the same value of 1 making their comparison easier and immediate for higher values of the number of stages. As we can see on Fig.3.27, the larger the repeaters are, the higher percentage of frequency we can gain from optimal repeater insertion. This gain grows much faster than the repeater sizing up: this can be explained by looking at the form of the delay expression. Indeed the delay for a wire segment can be separated into a sum of capacitances multiplied by R_{gate} and another one multiplied by R_{wire}:

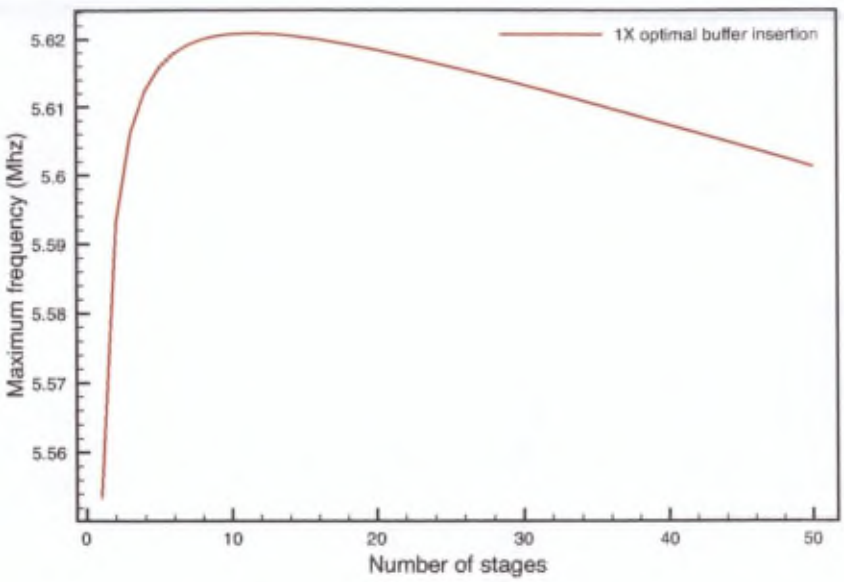

Figure 3.26: Maximum frequency of a 54.8 mm global wire for different values of the number of stages

while R_{wire} only depends on the wire sizing, R_{gate} depends on the saturated drain current (hence the gate width). For small sized repeaters, R_{gate} is much bigger than R_{wire} so that delay is roughly proportional to the length: there is not much to gain from optimal repeater insertion. On the contrary, when repeaters are sized up, R_{gate} decreases so that it becomes closer to R_{wire} value: the delay becomes quadratically proportional to the wire length which increases the benefit that we get from splitting long wires into smaller segments. This trend explains why the relative frequency gain increases with repeater size and the slope becomes steeper around this maximum. Choosing a 50X gate sizing makes the relative frequency gain even reach 33% making it worse to spend some effort on optimizing the number of stages.

Wire length dependency Now that we have observed the influence of the gate width, it could be interesting to measure the impact of wire length on optimal repeater insertion curves. Therefore we have represented in Fig.3.28 the normalized frequency versus the number of stages for different lengths of a global wire ranging from 1mm to 10cm. This experiment was performed for 1X sized repeaters driving the wires. As we can see from this figure, the optimum number of stages decreases as the total length of the path decreases while the slope of the curve becomes steeper and steeper past that point. This phenomenon is in fact quite simple to understand: when the wires become

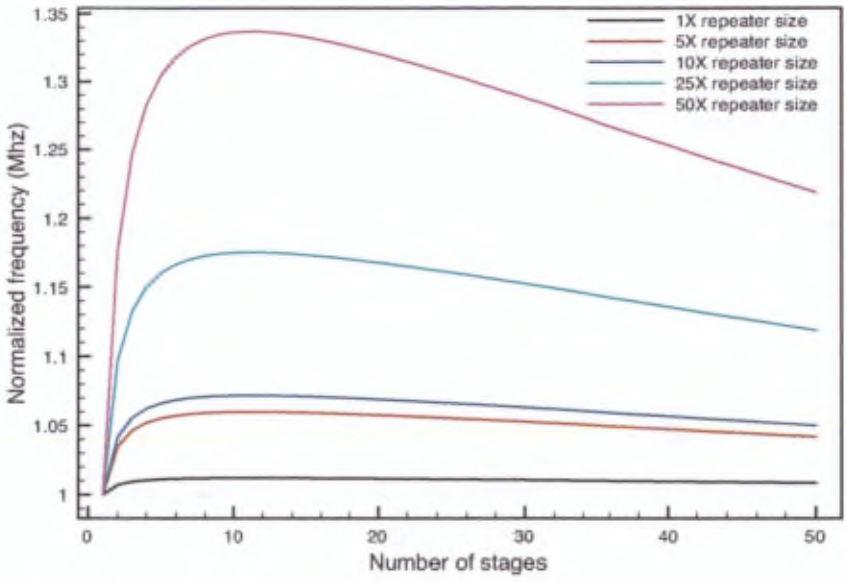

Figure 3.27: Comparison of the normalized frequency of 54.8mm long global wires for different repeater sizes

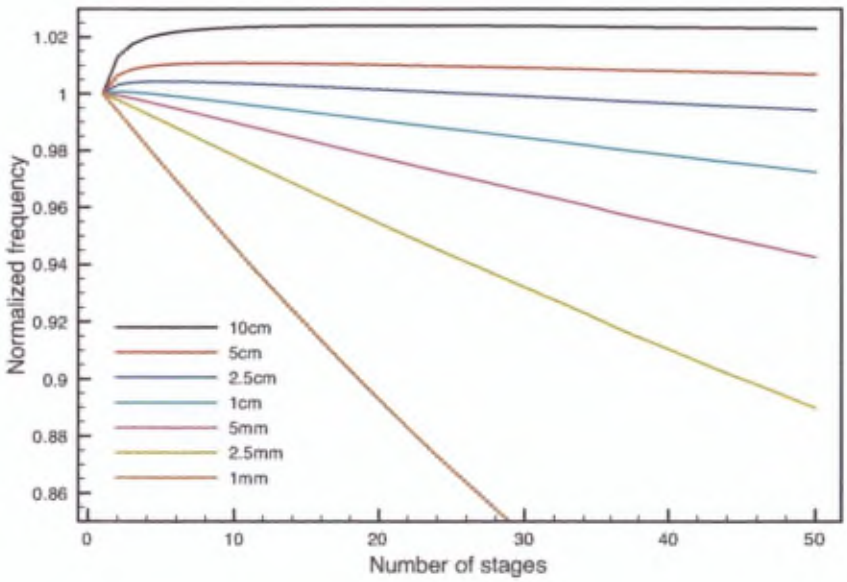

Figure 3.28: Comparison of the normalized frequency for global wires with different length (1X size repeaters)

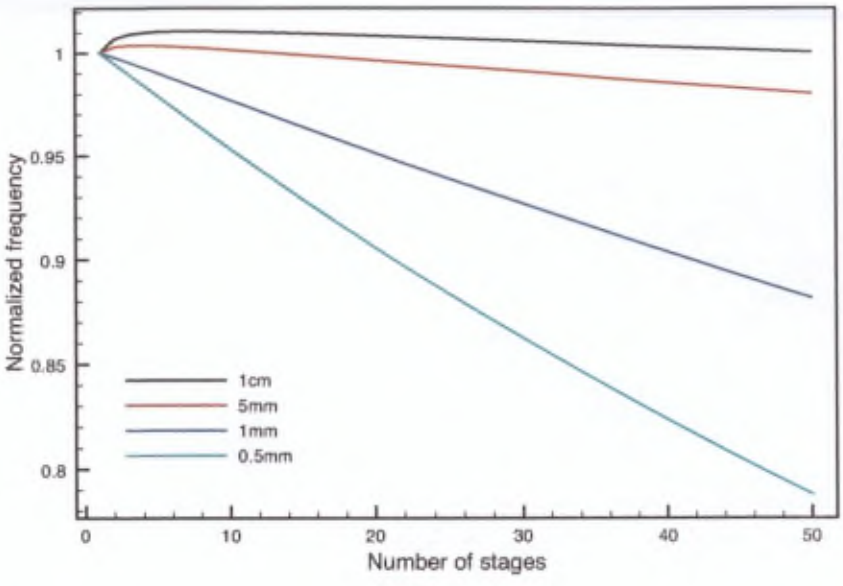

Figure 3.29: Comparison of the normalized frequency for local wires with different length (1X size repeaters)

shorter, their delay quickly decreases so that gate delay increases relatively. This makes the gain from wire splitting smaller and smaller compared to the gate delay remaining the same: the optimum number of stages thus decreases. This goes on to the point where the insertion of a single additional repeater even degrades the total delay hence decreases the frequency making repeater insertion harmful: this can be observed for global wire length of 5mm and below on Fig.3.28.

This phenomenon of decreasing normalized frequency with shrinking wire lengths also gives us a very interesting clue about the proper length of these fat wires: for lengths beyond 2.5cm, the curves show that repeater insertion is not appropriate. Obviously, the sizing policy of fat wires makes them not suited at all for such small lengths. Indeed if we perform the same experiment for local wires we can observe (Fig.3.29) the same phenomenon but for smaller wire lengths: this difference comes from the local wire sizing much more appropriate for smaller lengths.

Bandwidth study

Bandwidth optimization is often an important concern since data intensive applications (like video encoding) imply a heavy data exchange between different processing units at the scale of global communication[25]. Increasing the

frequency of a communication link is the usual way to improve the amount of data that flow on it. However performing this optimization blindly without any consideration for other parameters could lead to absurd solutions. As an example, increasing the distance between neighbour wires decreases the side-to-side wire capacitance hence increases the wire frequency *but* also the average interconnect pitch lowering the number of links that can be aligned side-to-side on a given silicon area. The sole optimization of wire frequency would always lead to increase the spacing between wires missing the opportunity to rather increase bandwidth by using more but slower wires. That's why we used a much more fair metrics for bandwidth optimization able to describe the compromise between wire frequency F_{wire} and its pitch $Pitch_{wire}$ as described by Eq.3.28.

$$BW = \frac{F_{wire}}{Pitch_{wire}} = \frac{F_{wire}}{S_{wire} + W_{wire}} \tag{3.28}$$

Impact of wire spacing on bandwidth Let us now see how the bandwidth optimization based on this new metrics impacts the wire dimensioning as it is defined by the global wire planning. Therefore we first decided to measure the influence of the spacing on our bandwidth metrics for a 5cm long global wire composed out of 10 stages driven by 1X gates. The resulting graph is represented in Fig.3.30 for a wire spacing ranging from 0.18μm to 6μm in the case of the nominal wire width (700nm). The represented curve has a maximum for a wire spacing value of 1100nm around which the bandwidth metrics value quickly drops.

For small wire spacing values, the side-to-side capacitance increases quickly and contributes more and more to the total wire capacitance value: this makes the frequency of the communication link decrease much faster than the pitch does hence decreases the bandwidth.
For large spacing values, the side-to-side capacitance only decreases very slowly contributing less and less to decrease the total wire capacitance while the pitch grows linearly with the spacing: the frequency grows much slower than the pitch does and the bandwidth decreases.

Interestingly the nominal wire spacing of 1300nm is quite close to the wire spacing value of maximum bandwidth of 1100nm with a small difference of 0.8% in terms of bandwidth: this means that the wire planning for fat wires has been dimensioned so that a good compromise is found bewteen wire frequency and pitch.

Impact of wire width on bandwidth As a second experiment, we wanted to study the second parameter that has a direct influence on the pitch hence the bandwidth metric: the wire width. We plotted the bandwidth versus the wire width for the nominal wire spacing value of 1300nm in the case of a global wire. As we can see from Fig.3.31 the curve has a maximum for a much

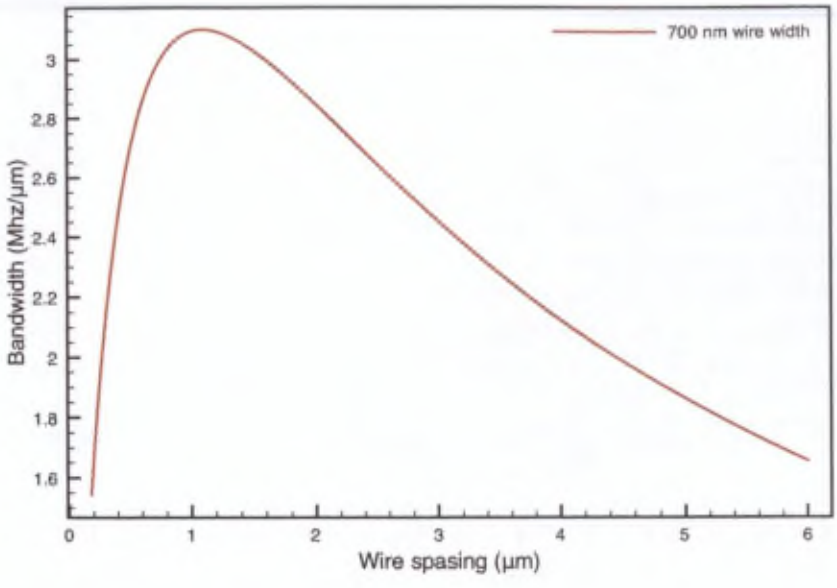

Figure 3.30: Bandwidth evolution for different wire spacing values and for a nominal wire width of 700nm

smaller value of the width than the nominal value of 700nm: by decreasing its value to 180nm, we have a 100% gain for the bandwidth per pitch value. Indeed when we start decreasing the wire width starting from 700nm while keeping the wire spacing constant, the vertical wire capacitance value shrinks making the total wire capacitance drop while the resistance increases due to a smaller wire section: since the wire resistance is quite small compared to the gate resistance, the delay decreases (see Eq.3.20). This effect coupled with the width shrinking favours the bandwidth per pitch that quickly increases for smaller widths until the wire resistance becomes too small and degrades the delay sufficiently to outweight the width reduction leading to the optimum width value.

Such a small wire width optimal value could seem surprising but for larger repeater sizes we can clearly see on Fig.3.32 that the optimal wire width value actually grows with the repeater size: this directly results from the gate resistance decreasing. We can also notice that scaling up the gate width makes the dimensioning of optimal bandwidth more and more profitable because the relative contribution of the wire capacitance to the delay becomes more important due to the smaller gate resistance.

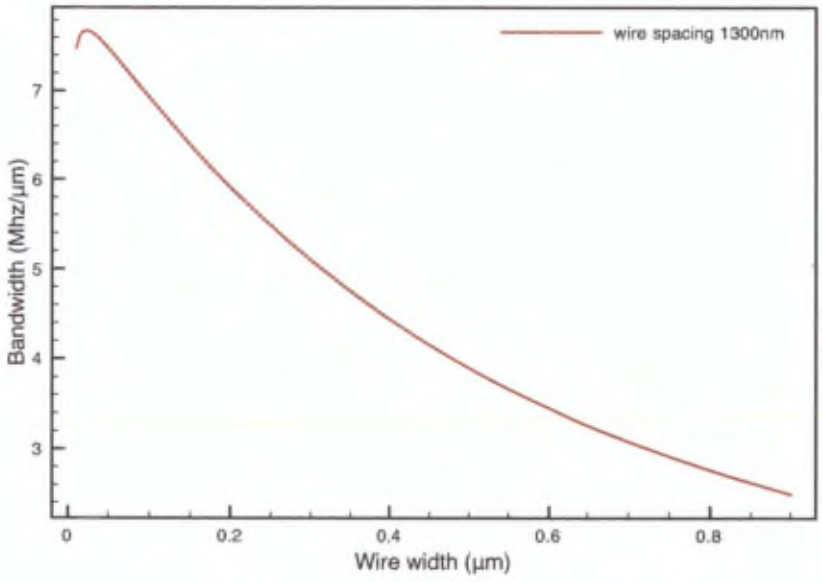

Figure 3.31: Bandwidth versus wire width for a nominal wire spacing of 1300nm

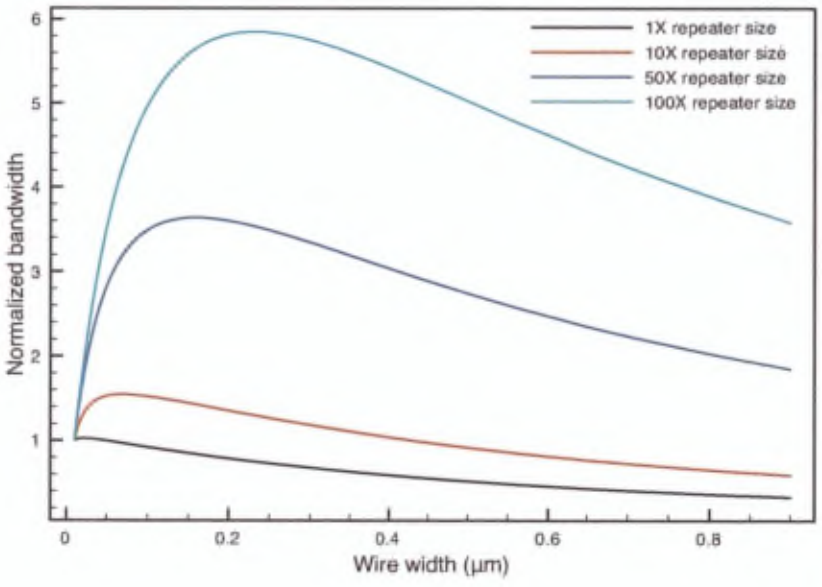

Figure 3.32: Normalized bandwidth versus wire width for different repeater sizes

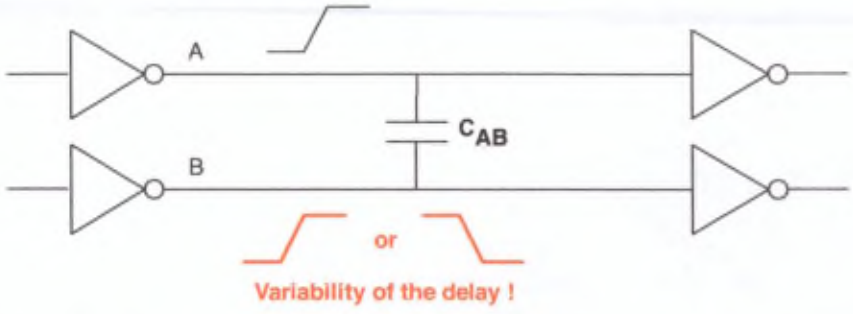

Figure 3.33: Illustration of the coupling capacitance variability resulting from the Miller effect between two neighbour wires

Impact of miller effect on delay

Context Timing closure has become a very important task in nowadays VLSI design particularly for high-end process with smaller and smaller feature sizes[26]. It's only after the place and route step that we are only able to measure accurately wire delay and observe its impact on frequency: if previous estimations were incorrect, it might well jeopardize the initially planned timing and require re-design. One of the biggest source of possible timing violation is the *Miller effect* introducing high variability on the delay: many authors are worried about this problem as it may lead to serious critical path delay penalties[27][14]. In this section, we will study Miller effect and compare its effect on delay to other possible parameters to see how it contributes to the overall delay variation.

Miller effect modeling Miller effect results from the electrical field interaction occurring between neighbour switching wires and is more important at the global communication scale. Fig.3.33 illustrates this phenomenon for two neighbour wires A and B: when A switches in one direction, its delay is lowered (increased) when B simultaneously switches in the same (opposite) direction.

This effect resulting from the interaction of the electrical field of the both wires is modeled by a side-to-side wire capacitance C_{AB}. To represent this variation of capacitance due to the neighbour wire switching, it is common to multiply the side-to-side capacitance C_{side} by a factor of correction FC_{Miller} that is lower (bigger) than 1 for a neighbour wire switching in the same (opposite) direction as the victim wire. In our Yeti representation we thus modify the model for C_{wire} in order to take Miller effect into account as shown in Eq.3.29 is.

$$C_{wire} = 2 * (FC_{Miller} * C_{side} + C_{vert}) \tag{3.29}$$

The value of the correction factor varies from one author to another: while [28]

Technological parameters	FC_{Miller}	Relative stage delay	
		Lower value	Upper value
±5%	1	−6.8%	21.1%
±5%	0.5 → 1.5	−24.6%	55.2%
±10%	1	−13.4%	48.1%
±10%	0.5 → 1.5	−30%	76.4%
nominal	0.5 → 1.5	−19.1%	19.1%

Table 3.11: Comparison of the miller effect and a ±5%/±10% technological input variation impact on the delay variability

uses a $[0, 2]$ range, [29] identifies situations where this interval even extends to $[-1, 3]$. However in our work, we will use a more reasonable correction factor range of $[0.5, 1.5]$ as proposed in [30].

Experiments To objectively quantify the impact of Miller effect on delay variation, we decided to compare it relatively to other possible sources of variations: technological parameters. Since manufacturing processes cannot be 100% accurate (due to lithography, material properties change over the chip from one production batch to another), dispersion is always present[31] so that it must be taken into account exactly like Miller effect. We chose the following technology related parameters: the relative gate oxide permittivity, the relative wire insulation permittivity, the gate threshold voltage and the gate supply voltage. For each of these parameters we performed a ±5% and ±10% input value sensitivity study that we compared and combined with Miller effect to measure the impact on delay: this can be estimated easily using bounds instead of scalar values in Yeti. The results of this study are provided in Table 3.11 for a 5cm long global wire using the wire planning described earlier.

First of all, we can see that the miller effect has symmetric effect on the delay as expressed on last line of this table: this value of 19.1% is far from being negligible and has a strong influence on delay variability. However, looking at the technological parameters only we can see that we encounter higher delay variations even for the ±5%. This shows clearly that if the Miller effect has a strong effect on delay prediction, so do other parameters whose variation with process is unavoidable: most of the time, however, they are ignored in papers. Combining the ±10% technological parameter variation with the Miller effect we even reach more than 100% total variation around the nominal delay value which is tremendous.

Secondly it is interesting to observe that the technological parameters, contrarily to the miller effect, have a non-symmetrical impact on delay variation: the upper value is always bigger than twice the lower value. This makes process variations even more annoying because only the upper value variation is

	Behaviour performances	Basic operations performances
File generation ON	10.25 kB/s	184,56 kBOp/s
File generation OFF	88.0 kB/s	1.58 MBOp/s

Table 3.12: Performances of Yeti expressed in behaviours per second (B/s) and in basic operations per second (BO/s)

a dimensioning criterion used for the maximum frequency evaluation.
In summary we showed using this example that, using the bound estimation feature of Nessie, it is possible to estimate the impact on the output of a model for an input sensitivity analysis of several combined parameters.

3.4 Yeti performances

Now that we have examined the different features offered by Yeti during our two previous case studies, we will provide the reader with some information about the computation performances of Yeti.

To evaluate the performances of Yeti, we have compiled and run Yeti under a Mac OS X Intel Core 2 Duo 2.0 GHz computer. We used our previous example of Codrescu (see Sec.3.2.2) composed out of 8 relations (3 table-based and 5 analytical relations) and performed 10^5 successive evaluations of the behaviour. Since a behaviour evaluation time depends on its complexity, a much more fair measure would consist in expressing the performances in terms of number of basic operations (addition, multiplication and so on) calculated per second which is independent of the behaviour complexity and therefore represents a better measure of the absolute computation performances of Yeti. Our current model contains 18 basic operations once we summed them up from the 8 relations (table-based relations were counted for one single basic operation which won't overestimate the performances). Each simulation was performed several times to calculate an average value of the calculation time over a sufficient number of simulations to get rid of too high variations on the resulting value due to other threads executed simultaneously by the operating system. The results of these measures are given in Table 3.12 with XML output file generation successively disabled and enabled.

The first striking observation that can be made from the results of this table is the huge difference of almost a tenfold factor between the performances with and without XML output file generation. When XML output file generation is enabled, Yeti spends most of its time writing data on the hard drive which considerably slows the Yeti evaluations down. In our case, the generated output file corresponding to 10^5 evaluations of the same behaviour and the displaying of the two input and six output parameters lead to 42Mo file which is already quite heavy. XML is very easy to read for a human but the overhead introduced

by tags compared to rough data is far from being negligible. If the user is only interested in the model evaluation capability to directly exploit results in its own program, we advise him to disable the output file generation to gain a lot of computation time.

If we disable the file output generation, Yeti performs 1.58 Mops/s which is quite satisfying and means that the overhead compared to rough compiled operations is reasonable and makes Yeti suitable for large campaign simulations. Furthermore the way the relations are evaluated in Yeti (basic operations and relations redundant evaluations avoidance) guarantees that the computational complexity grows linearly with the addition of relations and basic operations making Yeti still efficient for large behaviours. Concerning the C++ obkect structure building time from the XML initialization file, it took 30ms for the complete behaviour of 8 relations which is almost negligible compared to the duration required to evaluate thousands of behaviours.

3.5 Conclusions

Through the two case studies we have presented in this chapter the different features of Yeti regarding scripting, input/model sensitivity analysis, automatic plot generation. Based on this set of abilities, we have extended the model of Codrescu to derive new results with almost no time penalty once the initial model was entered into the framework. A second study has demonstrated how it is possible to build models from scratch and proposed several new results applied to the optimization of bandwidth and multiple stages maximum frequency.

If Yeti is probably quite powerful when it comes to deal when analytical models, it has the drawback of requiring complex XML files to be entered manually by the user which might be difficult at the beginning. However once the files have been defined, drawing automatically new plots between any pair of parameters of the models, changing the orientation of a model to get new results or changing input value ranges is always a few characters typing away from the expected result. When gaining experience in using Yeti combined to an XML editor, building up a model and initializing the simulation becomes faster and faster. Anyway offering model specification flexibility and very syntaxic and semantics verification will at some point require incompressible effort to put in.

Bibliography

[1] L. Codrescu, M. D. Pant, T. M. Taha, J. Eble, D. S. Wills, and J. D. Meindl, "Exploring microprocessor architectures for gigascale integration." in *ARVLSI*, 1999, pp. 242–255.

[2] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set, Tech. Rep. CS-TR-1996-1308, 1996. [Online]. Available: citeseer.ist.psu.edu/burger96evaluating.html

[3] J. C. Eble, V. K. De, D. S. Wills, and J. D. Meindl, "A generic system simulator (genesys) for asic technology and architecture beyond 2001," in *ASIC Conference and Exhibit proceedings*, 1996, pp. 193–196.

[4] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1967, pp. 483–485.

[5] Intel, "Intel core2 quad processor overview."

[6] D. Stroobandt, *A Priori Wire Length Estimates for Digital Design*. Boston / Dordrecht / London: Kluwer Academic Publishers, 4 2001.

[7] P. Zarkesh-Ha, J. Davis, W. Loh, and J. Meindl, "Stochastic interconnect network fan-out distribution using rent's rule," *Interconnect Technology Conference, 1998. Proceedings of the IEEE 1998 International*, pp. 184–186, Jun 1998.

[8] W. D., "Revisiting the fo4 metric," *Real World Technologies*, p. 9, 2002.

[9] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr 2001.

[10] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron," in *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 1998, pp. 203–211.

[11] J. Brockman, "Cse 462 - vlsi design - net delay," Course notes of VLSI DESIGN - University of Notre Dame, Indiana, 2004.

[12] M. Bohr, "Interconnect scaling-the real limiter to high performance ulsi," *Electron Devices Meeting, 1995., International*, pp. 241–244, Dec 1995.

[13] K. Nabors and J. White, "Fastcap: a multipole accelerated 3-d capacitance extraction program," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions onComputer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 11, pp. 1447–1459, 1991.

[14] J. D. Meindl, J. A. Davis, P. Zarkesh-Ha, C. S. Patel, K. P. Martin, and P. A. Kohl, "Interconnect opportunities for gigascale integration," *IBM Journal of Research and Development*, vol. 46, no. 2-3, pp. 245–264, 2002.

[15] J. Cong and D. Z. Pan, "Interconnect estimation and planning for deep submicron designs," in *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM, 1999, pp. 507–510.

[16] D. Sylvester and K. Keutzer, "Getting to the bottom of deep submicron ii: A global wiring paradigm," 1999. [Online]. Available: citeseer.ist.psu.edu/sylvester99getting.html

[17] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits (3rd Edition)*. Prentice Hall, 2003, no. 702.

[18] A. Abou-Seido, B. Nowak, and C. Chu, "Fitted elmore delay: a simple and accurate interconnect delay model," *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 422–427, 2002.

[19] T. Sakurai, "Interconnection from design perspective," in *Advanced Metallization Conference conference proceedings*, 2000, pp. 53–58.

[20] A. Vander Biest, F. Robert, D. Verkest, and S. Vernalde, "A taxonomy for technology extrapolation," in *NORCHIP Conference, Oulu (Finland), 21-22 novembre 2005*, 2005.

[21] "Us patent 6338158 - custom ic hardware modeling using standard ics for use in ic design validation," 2002.

[22] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan 2002.

[23] K. Saraswat and F. Mohammadi, "Effect of scaling of interconnections on the time delay of vlsi circuits," *IEEE Transactions on Electron Devices*, vol. 29, no. 4, pp. 645 – 650, Apr 1982.

[24] C.-P. Chen, Y.-P. Chen, and D. F. Wong, "Optimal wire-sizing formula under the elmore delay model," in *DAC '96: Proceedings of the 33rd annual conference on Design automation*. New York, NY, USA: ACM, 1996, pp. 487–490.

[25] D. Milojevic, L. Montperrus, and D. Verkest, "Power dissipation of the network-on-chip in multi-processor system-on-chip dedicated for video coding applications," *Journal of Signal Processing Systems*, p. 15, June 2008.

[26] L. Scheffer, "Timing closure today," in *ASP-DAC proceedings*. Cadence, 2001.

[27] D. Sylvester and K. Keutzer, "Impact of small process geometries on microarchitectures in systems on a chip," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 467–489, Apr 2001.

[28] Y. Cao, C. Hu, X. C. Huang, A. Kahng, S. Muddu, D. Stroobandt, and D. Sylvester, "Effects of global interconnect optimizations on performance estimation of deep submicron design," in *Proc. IEEE/ACM Intl. Conference on Computer-Aided Design*, November 2000, pp. 56–61. [Online]. Available: http://www.gigascale.org/pubs/269.html

[29] A. B. Kahng, S. Muddu, and E. Sarto, "On switch factor based analysis of coupled RC interconnects," in *Design Automation Conference*, 2000, pp. 79–84. [Online]. Available: citeseer.ist.psu.edu/kahng00switch.html

[30] D. Haris and M. Bushnell, "332:578 course - chapter 7: Crosstalk," Course notes of Deep Submicron VLSI course - Harvey Mudd College and Rutgers University, Apr 2005.

[31] R.Preston, *Managing variability on sub-100 nm designs*, 2004.

Chapter 4

State of the Art on Performance Prediction Tools and Methods

Abstract

This chapter defines the state-of-the-art for tools and methods dedicated to VLSI systems performance estimation based on a wide review of the literature. We start by redefining the basic vocabulary to remove the ambiguity usually associated to some embedded design terms. Based on the description of 19 reviewed tools, we propose five comparison criteria (functionality description, platform description, mapping methods, estimated performance criteria and design space exploration capabilities) to extract their common methodologies and concepts. Comparing those different criteria for each tool allows us to present the advantages and limitations of the state-of-the-art performance estimation frameworks and establish the requirements for a new more generalist and flexible tool.

4.1 Introduction

Looking at the literature, the scope of chip performance estimation methods and tools ranges from simplistic and very compact models to completely integrated flows also enabling synthesis. Such a variety is not easy to handle in a single chapter and that's why we separated the state of the art in two different parts in order to simplify the classification of the tools and systems:

- Closed-formed modeling based performance estimation tools and systems were reviewed in the chapter related to Yeti (see Sec.2.2).
- Methodologies and systems that rely on the explicit separation of the functionality and the platform running it in order to perform system performance evaluation will be reviewed and discussed in this chapter.

Even with this separation, we still face a tremendous amount of tools to deal with because performance estimation is a process carried out at very different steps of the design flow for various aspects of a system. If all these frameworks may sometimes seem very different from each other, they however rely on common concepts, methodologies and representations: that's why they deserve a specific chapter to be discussed and compared.

Classification of the papers

Since performance estimation covers a wide area of the literature, claiming to exhaustively review each single relevant paper would be difficult to achieve or alternatively require an unbelievable amount of time. We tried instead to select out the most innovative, comprehensive and fundamental publications to be able to identify common concepts and methodologies. To help the reader finding its way among the different papers that we are going to review, we decided to divide them into several categories based on their main concern:

- *Functionallty-centered tools and behavioral languages*[1] address the representation and definition of the system functionality which is always the starting point of a design: its knowledge is mandatory in a context of performance estimation.
- *HW/SW codesign tools* cover many different aspects (specification, simulation, synthesis etc.) with a focus on platform heterogeneity modeling. These tools are usually meant for synthesis and try to find a direct path from functional specification down to implementation.
- *Y-chart* based tools explicitly model the functionality, the platform and their mapping and therefore make a very interesting substrate for performance exploration. Most of the time they put the spell on the methodology rather than implementation.
- *Design space exploration tools* address the problem of exploring -or at least providing exploration help- for a wide space of possible solutions in order to extract the most interesting ones.
- *UML* is a modeling language coming from the software engineering field and has inspired many authors for the development of performance evaluation and functionality specification tools. Therefore we have chosen to devote a section to UML-based tools and methodologies.

[1] Languages describing how the system *behaves* (in other words its functionality)

- *Software performance estimation tools* enable the execution time prediction of an application running on a given microprocessor. Since it's mostly all about benchmarking/profiling for a given architecture, this choice doesn't satisfy our care for generality and we will thus discard it[2].

However some of the publications we are going to discuss simultaneously fall in several of the categories we previously defined and were therefore classified based on their predominated aspect. Before starting the review of the different tools, we will spend some time in defining clearly some ambiguous concepts used in the world of embedded design.

4.2 A bit of vocabulary

If we first have a look at some very generalist papers about VLSI performance prediction systems, we may well read some bad news for the coming state of the art to establish[2]:

> "There are probably as many descriptions of system-level design as there are system designers and codesign researchers."

Unfortunately the more papers you'll read on the subject, the more you'll agree with that statement. To avoid any further ambiguity in the usage of terms that will be often met in this chapter, we will spend some time to define different concepts and the way we will use them through the rest of this text.

Let us start with some basic vocabulary that will be at the centre of performance prediction tools and systems:

- *System* refers to the virtual and material sum of all the designed entities i.e. the functionality and the platform.
- *Functionality* defines *what* the system is supposed to do, in other words how it will respond to its environment by generating outputs based on input stimuli. The *functional specification* covers all the constraints that the system need to fulfill in order to reproduce the behaviour defined by the designer.
- *Platform* defines a set of material blocks that will provide the functionality with different usable services and associate implementation (i.e. non-fucntional) costs to those services (like execution time, required silicon area, electrical power consumed etc.). That's only when the platform has been defined that the functionality can be back-annotated with those costs to make sure that sure that the *non-functional* constraints are satisfied.

[2]The reader interested in software performance evaluation is however strongly advised to have a look at one of the most exhaustive and astonishing survey[1] we came across during our literature survey. If it's a little old now (2001) but it presents a very complete and critic review of related frameworks and tools.

- The *mapping* operation consists in establishing the adequacy between the functionality and the platform. This will be done by defining spatially (the allocation) and temporally (the scheduling) how each functional operation will be associated with a platform element.
- An *Abstraction level* refers to the description granularity of the functionality and the platform at a given design step. It establishes the different primitive blocks they will be composed out of as well as the timing granularity. The more we move down in the abstraction levels, the more detailed and accurate the system description will be.

Aside from the previous concepts, the magic words *hardware* and *sofwtare* come very often in front of the scene when looking at the literature. However depending on the people using them and the context, their meaning can be very different and sometimes misleading. In the next paragraphs we will expose our critical analysis of their different definitions and set up the vocabulary that we will use in the remainder of this chapter.

Programmer's perspective

> "Computers cannot do any useful work without instructions from software; thus a combination of software and hardware (the computer) is necessary to do any computerized work."[3]

The high-level language programing community uses the term *software* as a reference to the sequence of instructions defining the functionality of a computer-based system. Software is a pure virtual specification while hardware is a synonym of computer (i.e. a generic microprocessor-based machine able to run any behaviour that can be specified by a programming language). Hardware is never exposed to the programmer: all the information related to the microprocessor type and instruction set are encapsulated into the compiler that performs the mapping between the software (functionality) and the hardware (the platform). Performance only refers to execution time resulting from the hardware computation power, the programing skills of the software programer and the compiler used.

Embedded design perspective

> "Embedded controllers for reactive real-time applications are implemented as mixed software-hardware systems. These controllers utilize Micro-processors, Micro-controllers and Digital Signal Processors but are neither used nor perceived as computers."[4]

The embedded system community gives a completely different meaning to the previous terms: software refers to a microprocessor/microcontroller design

style while hardware means FPGA/ASIC design style. Compared to computer programming where the material support is fixed and entirely defined, embedded system designers have to specify the platform as well as the functionality using specific tools and languages. One of the major task called HW/SW partitioning consists in dividing the functional specification into smaller blocks called tasks and to assign them to the different HW/SW components of the chip. By moving one task from software to hardware or inversely, the designer is thus able to compare different performance compromises (in terms of time, power consumption etc.). The additional effort to tailor the platform to the functionality comes with a global improvement of the resulting general performance (smaller execution times, higher throughputs, lower power consumption etc.).

Y-chart perspective

> "The Y-chart is an iterative design methodology that allows concurrent development of hardware and software."[5]

Hardware and software may also appear once and then in the context of Y-chart description: its use is *in our opinion* very inappropriate and incredibly misleading. In the quotation at the beginning of this paragraph, *hardware* is used for *platform* while *software* actually refers to the *functionality*. If this misuse of language is quite understandable, it entails a certain ambiguity with the previous definition of the hardware and software terms[3]. Indeed the Y-chart defines at multiple abstraction levels how the functionality is mapped onto the platform.

The *functionality* defines *what* the system is supposed to do whatever the specification language chosen by the designer. This functionality is totally independent from any material or silicon consideration: this means no cost can be associated with it, only can it be simulated to make sure that it reproduces the behaviour the designer had in mind.

The *platform* is a physical support that provides the different services that the functionality requires and associates costs with it. However this is not sufficient to get our system running: we need to define where (the allocation) and when (the scheduling) each subpart of the functionality will be executed on the platform. That's only when the functionality is mapped onto the platform that the notion of performances makes sense, not earlier.

[3] It is interesting to note that, in the programer's view definition of terms, software however corresponds to functionality and hardware to the platform

Hardware Description Language

A particular category of languages called HDL (*Hardware Description Language*) are used for hardware design style: however their name deserves some clarification since it could be misleading. If HDL are used as an entry-point of FPGA- or ASIC-based systems, it should be understood that they don't actually *describe* the hardware. On the contrary, HDL offer a way to specify a hardware functionality independently from the platform and its structure by providing data types, explicit concurrency and specific notions of times that object-oriented languages don't. Because these concepts are made explicit, it is easy to make a one-to-one parallel between functional processes and platform components: that's why it is commonly said that HDL allow the user to also describe the structure of a system and not only its behaviour.

In the remainder of this work and unless we explicitly mention the contrary, we will adopt the following vocabulary convention:

- Hardware and software will be used as previously defined in the context of embedded system design
- Functionality, platform, system and mapping will be used as explained at the beginning of this section
- Behaviour and functionality are most of the time used to refer to the same concept: we will consider them as perfect synonym
- Architecture and application are often used at the highest levels of abstraction to refer to the platform and the functionality. We will adopt this vocabulary when the papers we are reviewing use these terms themselves.

Now that these definitions have been established we are now ready to examine the state-of-the-art tools in performance prediction.

4.3 Literature survey

4.3.1 Behavioural languages

Ptolemy II

Ptolemy[6] is a JAVA-based framework developed to deal with heterogeneity in embedded system modeling: it has become quite popular since its release and widely used in the research field.

Ptolemy specifically targets the specification and co-simulation of heterogeneous models: it does not address any system performance estimation issue, nor does it explicitly describe the platform. It rather focuses on a behavioral representation of the system seen as a structure of components called *actors*.

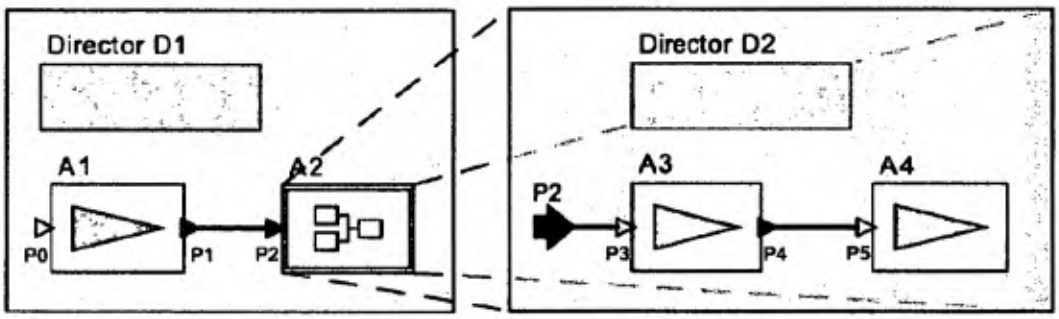

Figure 4.1: Ptolemy hierarchical representation of a system using nested actors (A_i) communicating through ports (P_j)[6]

The "*hierarchical heterogeneity*" results from the structure adopted in Ptolemy: actors are either composite (internally defined as a structure of several actors) or atomic (emphasizing the fact they are indivisible hence not composed out of other actors). Actors communicate with each others using *ports* that encapsulate the communication mechanism (FIFO, rendez-vous, messages etc.). Fig.4.1 depicts a simple 2-level hierarchical model with composite and atomic actors with their respective ports.

A model of computation (MoC[4]) defines the execution semantics of a structure composed out of actors: in other words it specifies the order of execution of the different actors and how communication between them will take place. MoC's are implemented using five specific *domains*:

1. Communicating Sequential Processes (CSP) use the mechanism of rendez-vous points to communicate between actors.
2. Continuous Time (CT) domain defines the behaviour of processes based on mathematical functions depending explicitly on time.
3. Discrete Event (DE) processes use discrete events chronologically ordered and executed on a continuous timeline.
4. Process Network domain (PN) define actors communicating with each other using FIFO's with a non-blocking writing/blocking reading mechanism.
5. Synchronous dataflow (SDF) use a token producer/consumer paradigm to represent the synchronous execution of actors and their communication.

Accordingly to the paper, the introduction of domains nearly suppresses the

[4]A very comprehensive and complete classification of these MoC's can be found in [7]. It also provides the reader with some guidelines to choose among the numerous available MoC's based on the respective functional aspects that they capture.

risk of *emergent behaviours*[5] and therefore allows the user to hierarchically build models without having to worry about some possible modifications of the behaviour.

El Greco (CoCentric System Studio)

El Greco[8] (which has recently been turned into Synopsys *CoCentric System Studio*[9]) is an environment for fast modeling, specification and execution of complex heterogeneous systems. It is intended to be used for algorithm exploration and for functional verification. El Greco has many strong similarities with Ptolemy regarding its hierarchical component modeling and co-execution capabilities. Models are built by the user either in a top-down or bottom-up flavour. Four types of hierarchical models are available:

1. *Data flow graphs* are graphs whose nodes may contain instances of other models.
2. *Or-models* are made out of mutually exclusive states with transitions between these states. States may be hierarchical (meaning they contain other encapsulated models) or atomic (with their behvaiour defined using C++ code).
3. *And-models* -contrarily to the previous type of model- are able to represent tasks whose execution can take in parallel.
4. *Gated models* contain 2-children nodes with mutually exclusive execution triggered by a condition.

The first model type is meant to be used for control-free behaviours while the three last models are best at representing control-dominated behaviours. However, contrarily to Ptolemy which interprets the models to simulate their behaviour, El Greco is able to compile them, leading to faster execution. To enable the co-simulation of the different models, El Greco uses a dataflow specification based on cyclo-static graphs (CSDF[6]) and a control specification strongly inspired form Esterel control semantics (see Sec.4.3.1).

[5] Emergent behaviours correspond to unwanted behaviours that the user had not foreseen resulting from the interaction and co-execution of the communication of different heterogeneous models.

[6] CSDF are a generalized version of SDF already briefly discussed in the section devoted to Ptolemy (see Sec.4.3.1). While SDF actors consume a fixed number of input tokens number for each firing condition, CSDF actors use a number of input tokens cyclicly selected out of a list over time. This flexibility enhances application parallelism exposition but comes at the price of a more complex static scheduling step. The interested reader is advised to refer to the very comprehensible [10].

ESTEREL and other synchronous languages-based tools

Esterel[11] is a language belonging to the synchronous[7] reactive[8] language family relying on mathematical techniques to specify and verify the system behaviour. This ability to mathematically prove some system properties is called *formal verification* and has been a driver for the industrial adoption of those languages designed for safety critical applications (like in avionics).

Esterel is mostly control-oriented relying on a mechanism of concurrently-running and preemptive threads with a synchronization based on a single clock[9]. This language has been converted into a commercial tool called *Esterel Studio*[14] providing the user with a complete design flow around Esterel from specification, simulation and formal verification to System-C/C code generation and VHDL/Verilog code generation.

Apart from ESTEREL, we can also mention other synchronous reactive languages like LUSTRE[15] and SIGNAL[16] that are more suited to capture data-flow oriented behaviours. Both languages gave birth to integrated tools respectively called SCADE and SILDEX.

SystemC

SystemC is a hardware description language (see Sec.4.2) that can be viewed in a certain extent as a successor for VHDL and Verilog. Compared to the latter, it introduces a new abstraction level called TLM[17] to replace RTL allowing the specification of the system beahaviour with less effort while keeping it executable.

SystemC isn't a new language at all in the sense that it extends C++ with a complete and open source library. The underlying idea that has lead to the development of SystemC is reuse: indeed C/C++ is a well-known software programing language and remains, despite of its age, widely used. That's why several attempts to preserve its syntax while making it closer to the hardware have been made: among them we can cite HardwareC[18] and SpecC[19] which focused on the extension of C rather than C++.

[7] Synchronous languages[12] are based on the synchronous concurrency model where processes are able to perform computation and exchange information in zero-time. Languages like VHDL fall not in the SDF category because they need to introduce delta-cycles to resolve the simultaneous execution of concurrent operations in order to emulate synchronism. It is important to understand that this zero-time execution and communication capability only affects simulation where the real value of the execution time associated with an operation is not necessarily known and only operation precedence makes sense.

[8] Reactive systems[13] are defined as systems reacting to stimuli present in their environment by producing the adequate outputs within a certain amount of time. Examples of reactive systems can be found in avionics and mechanical control systems.

[9] The exact mechanism of communication and synchronization goes well beyond the scope of our discussion. For more details the reader is advised to refer to [11]

SystemC takes the approach to preserve C++ in its whole and implements a library that is adding all the features related to hardware that it lacks. Since then, SystemC has become really popular in the academic field and is promised to a bright industrial future thanks to the adoption of SystemC TLM 2.0.

The major extensions to C++ introduced by SystemC are the following:

- *Data types* for hardware description need to be more specific than C++ classical types. SystemC therefore offers the support for logic and bit-wise types. Beyond dedicated operations that can be carried out on them (like logical AND, OR etc.), the list of values they can take is adapted to represent hardware possible driver state (like high impedance output, undefined value, weak driving etc.).

- *Concurrency* is crucial to represent the inherent parallelism of hardware that a sequential language like C++ is natively unable to capture. SystemC precisely fills that gap by defining a class named *module* whom all user-defined classes representing a hardware piece derive from. Inside this class, the user is able to define concurrent functions (containing sequential operations) using processes or threads (whose execution can be interrupted). Similarly to VHDL, SystemC also defines a sensitivity list that holds a list of signals whose change will trigger the process. Finally modules can be instantiated at will and composed hierarchically.

- *Communication* between modules is achieved through a mechanism relying on *ports* and *channels*. Ports are interfaces enabling a module to exchange information with another module and define the direction of the data flowing through it. Channels offer a very elegant way to abstract communication by implementing different underlying communication mechanisms (FIFO's based, semaphores etc.) that remain transparent at the module level.

- *Time* is a notion completely missing in C++ where operations only have to take place sequentially to comply with the functional specification. SystemC provides the user with different time concepts (notion of clock-/cycle, introduction of delay in an operation etc.) that are mandatory when trying to model a hardware block.

With its bit-wise data types, hierarchical building of concurrent processes and clock-based time paradigm, SystemC is able to extend C++ to capture hardware description. Furthermore it can be combined to already existing C++ code describing an application so that HW/SW co-design becomes possible within the same language.

4.3.2 HW/SW codesign tools

POLIS

POLIS[4] is a framework targeting the specification, simulation, verification and synthesis of embedded systems. It offers a unified Hardware/Software specification approach allowing the user to explore the effect of different partitioning choices on execution time. Rather than re-implementing already existing tools, POLIS philosophy is to reuse as much as possible free/academic tools and wrap them up into an integrated design flow by filling the gaps. This tool, referenced from many papers about HW/SW codesign, has been widely used by various authors and therefore deserves some attention.

POLIS uses a common HW/SW representation based on an extension of finite state machines called *Co-design Finite State Machines*(CFSM)[10]. By using such a "neutral" specification for both the hardware and the software, it becomes easy to represent a system in its whole regardless of its particular HW/SW partitioning.

The different steps from specification to implementation are the following:

- *High Level Language Specification* consists in automatically turning an initial specification written by the designer in a high-level language into a CFSM model. It is interesting to note that POLIS offers no automatic partitioning since "these decisions are based heavily on design experience and are very difficult to automate"[4].
- *Formal verification* takes place once the system model has been converted into CFSM. POLIS is then able to translate that model in FSM which is turn used as an entry point for an external verification system.
- *System co-simulation* is used to provide the user with feedback on its previous design choices. Simulation can be done by using Ptolemy or by turning the CFSM model into VHDL that can afterwards be exported into commercial simulation tools.
- *Hardware synthesis* is done using an external tool named SIS[21] that performs logic optimizations and provides a netlist of logic gates for the chosen target library.
- *Software synthesis* consists in two different steps. After optimizing the behaviour in a processor-independent representation, it translates the result into standard C that can be used afterwards in a processor-dependent compiler.

[10]Like FSM, CSFM[20] turn a set of inputs into a set of outputs using only a finite number of internal states but differ regarding the communication paradigm. Instead of using a synchronous communication mechanism, CFSM define asynchronous, finite, non-zero transition times representing the delay of execution. This transition time is of course unknown at the very first steps of the design but its value is refined through the different steps of the POLIS flow.

Thanks to the use of external tools integrated inside a seamless design flow, POLIS is one of the first complete methodology for HW/SW co-simulation and co-synthesis. However the price to pay for invoking external tools is the constant need for transforming the initial specification into tool-specific input formats: besides the overhead, the designer needs to make sure that the functionality remains preserved.

AADL

AADL[22] which stands for *Arhcitecture Analysis and Design Language* is a textual language developed for the combined description of an architecture and its application forming together a *system*. First meant to be only used in the avionics industry, AADL scope was finally proven to be broader than this latter field: it is currently under a SAE[11] standardization process for the specification of real-time systems.

In AADL, a system is a structure of different instantiated components defined by a hierarchy of different objects as depicted in Fig.4.2:

- A *category* defines the nature of a component. AADL includes 11 different categories divided in three subclasses: one application-related category (process, subprogram, thread, thread group, data), one platform-related category (processor, memory, device, bus) and finally the hybrid category including application- and platform-related components (called system).
- A *type* is associated with one particular category and describes the interface of a component in a way VHDL would do. The inputs and outputs signals of components are connected by the mean of *ports* allowing communication.
- An *implementation* is associated with a type and defines the inner structure and properties of a component (like an *architecture* in VHDL) using a syntax and a semantics that is peculiar to the category it belongs to. The interesting point in here is that several implementations may be associated with a same type representing different possible choices for a same component.
- An *instance* of a given implementation/type/category combination defines a component. Components may also contain inner sub-components (again like VHDL) defining in turn a hierarchical structure.

Based on a system of components described by the user, AADL is able to perform a time-based simulation and also verify aspects related to the security and reliability of the behaviour. A tool called *ADeS* developed as a plug-in

[11] The SAE (Society Of Automotive Engineers) is an association of industrial companies organizing conferences and developing standards in the field of automotive and avionics.

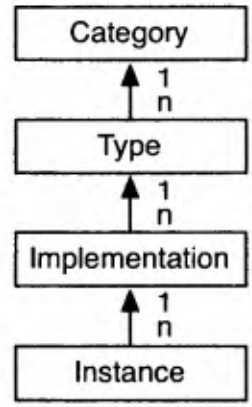

Figure 4.2: Object hierarchy inside AADL using a 1 to n composition relation

for the Eclipse environment allows the user to edit AADL code, visualize the resulting system and simulate it.

Chinook

Chinook[23] is a tool for automated co-synthesis of HW/SW mixed systems built from off-the-shelf components. It mainly focuses on control-dominated applications and provides a straight-forward path from specification to the netlist of interconnected components and compiled/synthesized code for the microprocessor/hardware units. Compared to other synthesis tools, Chinook facilitates design space exploration thanks to a high-level of automation and decreases implementation time and downtimes due to manual intervention which allows the designer to easily try and compare different implementation alternatives.

If Chinook is quite old now, it is one of the first tools introducing simultaneously the principles of a single functional specification written in a unique language, automatic interface and communication synthesis with timing constraints and an automated path down to a working HW/SW heterogeneous architecture: that's why it deserves some attention.

The main steps are the following:

- The *specification* entirely in Verilog consists in a functional part and a structured/interface-based description of the architecture. Additionally the designer has to provide a library of the different available components (microprocessors, FPGA's, peripherals) containing timing and I/O infor-

mation. The partitioning of the different processes is done manually[12] and each process can therefore be assigned to a particular computational unit by tagging it in the Verilog code (untagged processes imply an hardware implementation). Finally the designer is able to add timing constraints (response time, latency, rate, etc.) that will be further used during synthesis.

- The Verilog processes are synthesized for the FPGA's and compiled for the micro-processors using vendor specific tools. Chinook assumes that timing information can be drawn from these operations and fed back to the rest of the design process.
- A static and non-prememptive scheduling is performed to determine the execution order of processes on the different computational units with respect to the different timing constraints.
- Interface and communication synthesis exploits the information present in the component library (I/O and timing diagrams) to synthesize and generate the software interfaces and glue logic required for communication.

Aside from the synthesis, simulations can be carried out at various levels: behavioural level[13] using a Verilog simulator, co-simulation of the different HW/SW blocks using C models emulating the microprocessors and assembly-level simulations for debugging purpose.

4.3.3 Y-chart related tools

MESCAL

MESCAL[5] (the *Modern Embedded Systems, Compilers, Architectures and Languages* project) is a methodology for fast design space exploration of architectures targeting network-processing applications. More precisely it is meant to provide a correct-by-construct path from the functional specification to its mapping onto a network of ASIP's.

MESCAL chooses a 3-hierarchical levels Y-chart based approach for the system modeling as it is depicted in Fig.4.3. The functionality is defined independently from the platform and a mapping is performed afterwards. Based on the results of the performance analysis, the designer is free to iterate that process either by modifying the application, the architecture or the mapping strategy. The MESCAL methodology defines three levels of abstraction corresponding to

[12] Like POLIS, Chinook assumes that the designer is in better position to determine the mapping of the tasks on the computational units than automated tools would do.

[13] Behavioural level, used in the context of HDL, means a coding style closer to a high-level language than RTL and is here not at all a synonym of functionality.

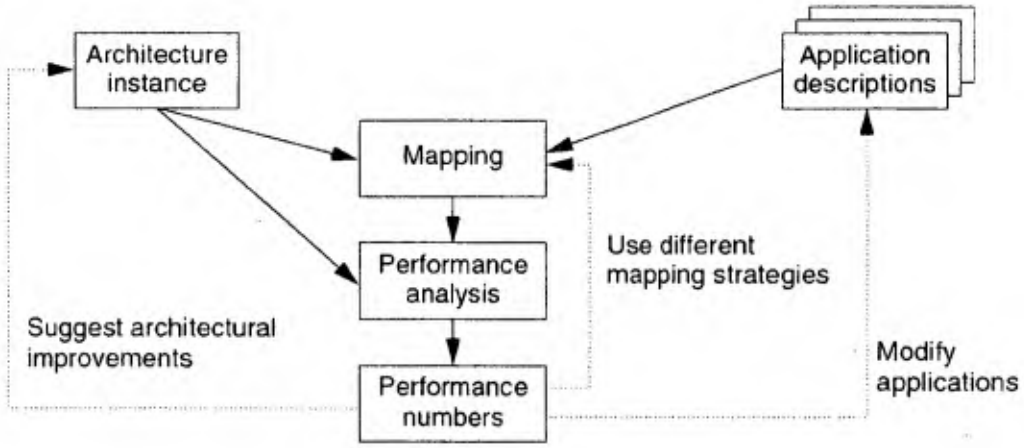

Figure 4.3: The Y chart methodology used within the MESCAL framework [5]

the levels of "hardware parallelism" that network-centric applications can take advantage of:

- The *architecture level* represents the architecture as a set of communicating ASIP's and accounts for the so-called processing-element parallelism.
- The *micro-architecture level* represents the main internal blocks of an ASIP where instruction-level parallelism can be studied
- The *bit-level* level represents the different computation units of the ASIP at a bit-level and is suited for exploring the bit-level parallelism of an algorithm for instance.

Since these levels are quite different from each other, the MESCAL methodology defines a multiple view methodology to represent the platform. Each view corresponding to a particular abstraction-level defines the appropriate semantics for the description of the platform at this particular level which makes further automated code-generation work way easier.

To perform design space exploration during the refinement of the initial architecture throughout the three different levels, MESCAL uses the TEEPEE framework. TEEPEE is nothing more than an extension of the previously discussed Ptolemy framework (see Sec.4.3.1) for the modeling of architectures. It provides *actors* support for the modeling of processing elements, memory and on-chip communication (limited to network-on-chip). MESCAL reformulates the mapping problem in an interesting way by considering an application as a set of components whose behaviour is defined by their particular MoC and by representing the architecture as a structure of virtual machines implementing these MoC's. The mapping problem then becomes a "matching" problem

between the processing elements and the application components that can be solved as a *communication networking* problem[14].

In summary MESCAL provides a framework for manual design space exploration and automatic code generation. However the authors admit that a joint refinement of the application and architecture throughout the successive, isolated and well-defined abstraction levels does not allow the designer to optimize the performances as much as a global multi-level optimization would do.

ARTEMIS

ARTEMIS[24] is a workbench focusing on the modeling and performance evaluation of System-on-chip multiprocessors architectures particularly for multimedia applications. The tool is divided into a system-level modeling environment called SESAME and a part devoted to the calibration of these models.

Following the Y-chart approach, SESAME decomposes the system model into the application model (describing the functional behaviour of the system) and the architecture model (enabling the execution of the functionality) capturing the performance constraints. Thanks to an explicit mapping of the application onto the architecture, performances can then evaluated.

- The *application* is described using Kahn Process Networks[25] (KPN) which is a MoC consisting in a network of processes communicating via unbounded FIFO channels with a blocking read/non-blocking write mechanism. KPN are either built manually based on user-defined XML files either automatically using the Compaan tool by extracting them from a code written in a subset of MATLAB. Executable code written in C/C++ is contained in KPN nodes.
- The *architecture* is defined in SystemC at a transaction level and is built out of a several blocks picked up in a library of components coming with performance annotations.
- The *mapping* consists in several successive steps that lead to the complete scheduling of the application on the architecture. First the application behaviour is simulated independently from any platform-related concern: during execution KPN process nodes record all the events relative to the communication and computation to establish a complete execution trace. After this step, the application is mapped onto a network of virtual processors (with execution capability and FIFO channels for communication): since there is a one-to-one relation between KPN process nodes and those virtual processors, this operation is pretty straightforward. The type of event scheduling can be chosen between FCFS, round-robin

[14] Very few details are given about this mapping method but we thought it interesting to put the spell on this *supply and demand* representation of the architecture and the application.

or user-defined policy. Finally the description of the virtual processors is gradually refined down to the actual platform where the total cost of the platform can be evaluated. To optimize the mapping method and enable design space exploration, SESAME uses a mathematical model based on the computation/communication demands of the application and architectural computation/communciation performances and power consumption. This model is explored using multi-objective methods to find approximative Pareto-optimal mapping solutions.

Since SESAME uses very high-level descriptions of both the application and the architecture, a satisfying level of accuracy is difficult to reach. Therefore some performance calibration can be done to profile the application on a configurable processor by refining the description to the micro-code level (the intersted reader should refer to the paper for more details).

SPADE

SPADE (System level Performance Analysis and Design space Exploration)[26] is a methodology for performance estimation of the mapping of an application onto an heterogeneous architecture at very high levels of abstraction. SPADE follows the Y chart approach and models the application, the architecture and its mapping separately so that their impact on performances can be estimated individually. The whole methodology relies on traces capturing the communication and computation order from the application to map them on an architecture. SPADE offers behavioural simulation capability but doesn't handle synthesis.

Compared to most of the other tools, SPADE individually models the application, the architecture and the mapping:

- The application is based on the KPN model of computation that has been chosen because they are able to efficiently expose the parallelism of an application described as processes exchanging information through channels. SPADE offers an API allowing the user to manually specify the application model in the C language: using a read/write function that reads/writes to/from the KPN channel and an execute function that triggers the execution of the related process. When the execution of a KPN model is simulated, each of these three functions triggers the recording of a new event entry inside an event vector to build the complete trace of the communication and computation behaviour. This operation results in the generation of the workload of the application that is going to be mapped onto the architecture.
- The architecture is described using a structured description of instantiated components taken from a predefined library. Each block able to perform computation (called a processing resource) is composed out of

a *trace driven execution unit* (TDEU) and an interface. The TDEU interprets the trace entries and offer a way to execute the associated processes. Each TDEU comes with a list of symbolic instructions defining the processes that can be executed by this particular component and their associated latencies are defined by lower-level models or estimated thanks to the designer experience. The interface enables to connect the TDEU to a communication block containing the protocol used to exchange data (shared buses and point-to-point connections are supported). Although a library of predefined components is already provided, the user is free to add its own blocks.

- The mapping needs to be specified manually by the user in order to evaluate how well the workload generated by the application will suit the architecture. Each process is mapped onto a TDEU in a many-to-one fashion meaning that several processes can share the same functional unit over time: this involves an explicit scheduling. Each process port is mapped onto an I/O port in a one-to-one manner which is more restrictive but doesn't require any scheduling.

Once these three elements have been defined, the simulation can take place to deliver the performances to the user in terms of execution time and TDEU/-communication usage. Traces are generated on-the-fly from the application to drive the simulation allowing all the estimation to be carried out in one single step.

SPADE is one of the most interesting tool that explicitly separates mapping concerns from the architecture/application modeling but leaves all the allocation/scheduling up to the user.

METROPOLIS

Metropolis[27] defines a metamodel with precise semantics capturing all the elements relevant in the context of embedded system modeling including the application, the architecture, their refinement, their mapping and the various abstractions used throughout the different design steps. This metamodel is believed to be a solution to the design discrepancies, specification alteration and bugs that occur during a design process due to communication problems between the different design teams. Rather than providing an extensive set of tools, design activity is centered around Metropolis metamodels and API's can be used to interface the framework with external tools while automatically turning the metamodel into the required input file format.

- The *application* is represented as a set of concurrent processes with their own execution thread holding sequential code. To exchange information, these processes use communication ports connected to a medium specifying the way data will be exchanged (the communication protocol).

Besides this communication/computation separate modeling, the meta-model also needs to specify the execution semantics. Therefore some instructions inside the threads are associated with events (execution start or end of a piece of code) that are combined into event vectors defining a behaviour. Since the latter is non-determinstic due to its non-ordered event based definition, the designer is able to add constraints on the precedence of processes and respective execution order.

- The architecture takes into account two different aspects: the functionality that they implement and its efficiency i.e. the cost associated to it.

  First the architecture is described as a network of processes like in the application except that this network represents the physical structure of the design rather than its behaviour. Each process contains several services implemented as threads (depending on the type of functional block it models) that can be accessed by the application in order to satisfy its needs.

  Second events associated to these threads trigger the execution of a certain number of *quantity managers* evaluating the cost of using this particular service. The type of each quantity can be chosen among a library or defined-by-the-user. Interestingly time is here seen as one quantity type among others and doesn't have a special role compared to other tools.

- The mapping step consists in merging an application network and an architecture network into what is called a *mapping network*. Basically this is done by synchronizing the events of the application and the architecture while applying allocation constraints provided by the user: the resulting mapping network can be seen as the implementation of a certain service.

  However different algorithms mapped on different architectures may be able to provide the same service for different costs: this is defined as a *platform*. The designer is able to choose among those different implementations the cost combination that best suits its needs and fulfill the requirements. Several platforms providing different services can then be recursively assembled into a more complex platform providing a richer service. This hierarchical building of services enables in turn bottom-up design flow and facilitates in a large extent the communication between different teams or companies which can exchange platforms implementing services.

Besides these mapping functionalities, Metropolis also offers verification, simulation and synthesis possibilities.

4.3.4 Design space exploration tools

MILAN

The MILAN (Model based Integrated simuLAtioN) framework[28] aims at simplifying SoC optimization and design space exploration by providing the user with a unified environment for the interoperation of heterogeneous simulators. Compared to other tools, MILAN is a component-centric tool where the compromise between simulation speed and accuracy is made explicit and is determined by the user needs. It relies on two different sub-tools called HiPerE (High-level Performance Estimator) and GenM (Generic Model) respectively used for the SoC performance estimation (limited to energy and latency) and the flexible representation of the SoC structure and properties.

GenM is a generic model for capturing SoC related information about the platform, the application and its mapping:

- A SoC platform is composed out several instances of processors, reconfigurable logic units, memories and interconnects. Additionally the user has to specify all the possible operating *states*[15] for each processor and reconfigurable unit along with their possible supply voltage (if it can be adjusted at run-time).
- An application is described using a data flow graph representing a network of communicating tasks (with their amount of data input/output)
- Mapping information (called *performance parameters*) include the energy and time for executing a given task on a particular microprocessor or reconfigurable unit but also the processing unit that each single task of the application will be assigned to.

MILAN methodology for performance estimation basically consists in a 2-hierarchical levels simulation process:

- The designer uses the graphical user interface to specify the architecture, application and mapping model based on GenM. The combination of these possibilities defines the design space that will be explored.
- The first performance evaluation takes place using HiPerE. Since the allocation has been manually specified in the mapping model along with the performance parameters, the performance evaluation task simply consists in a scheduling using a FCFS policy in this case. The total energy and latency is computed by integrating all the performance parameters over time: a report is then generated with the activity (including the idle time ratio) of all components to make bottlenecks identification easier.

[15] Each component is described by a collection of states representing all the possible configurations that this component may be in at a certain time. Additional information (again in terms of energy and latency) for transitions between states have also to be provided.

This performance estimation is carried out for all the possible designs: solutions with the most interesting time/energy compromises can then be selected for more accurate performance estimation.

- To replace the rough performance estimations of each component defined by the user at the first step, several techniques can be used to achieve more accurate results (complexity analysis, trace analysis, cycle-accurate simulations etc.). Based on the description of a task (in C/C++ or JAVA) and the knowledge of the components composing an architecture, MILAN is able to transform the task code so that it fits a chosen simulation tool input requirement. Once the simulation results are known, the previous performance evaluation can be performed again based on these new and more accurate values of performance parameters.

If MILAN is not particularly innovative regarding the design space exploration methodology, it outshines any other tool when it comes to model dynamically reconfigurable SoC thanks to its interesting state-based modeling.

A design estimation and exploration environment

The framework presented in [29] allows the user to perform design space exploration in a much easier and more intuitive way than classical tools do. Furthermore heterogeneous *predicators*[16] can be used together in the same environment sparing a lot of exploration time hence allowing the designer to explore more solutions.

The framework relies on the concept of *domain*[17] containing information about a specific area of expertise related to design (for instance the ASIC architecture, the technology, the cell library etc.). Each domain comes with a lot of information like identifiers (descriptive words related to that domain), associated domains, numerical parameters, associated tools and executable scripts.

Orthogonal domains can be gathered into what is called a *design context* containing all the useful information about the exploration that the designer wants to perform. Contexts can either be defined manually or automatically in an interactive way using the built-in search engine. The user enters a description in an textual form (for instance "FFT 1024" or "ASIC maxPower:=5mW 250nm FFT") and the search engine selects domains out of all the available ones to propose relevant contexts for the user to choose among. Once this is done, he's able to either refine the context by adding information suggested by the already selected domains or execute *commands* (like area, execution

[16] An estimator is defined in this paper as any kind of method that is able to predict a particular performance-related metric. This includes commercial performance estimation tools, databases of performance measurements, models regardless of their complexity and nature, etc.

[17] The notion of *domain* is totally independent and different from the one defined in ROSETTA (see Sec.4.3.4).

time, complexity etc.) to get a measure of the performances. The numbers are automatically extracted from estimators invoked by the script embedded inside the domain.

This framework comes as a graphical user interface communicating with the tool core by sending HTML requests based on client-server paradigm which makes it very easy to put the tool online and share user-defined domains.

The SoC Architecture Explorer

This *SoC Architecture Explorer* tool[30] is designed for fast design space exploration of SoC architectures and the evaluation of compromises between area and execution time. Combining all the design choice parameters values can however quickly lead to an overwhelming number of possibilities almost impossible to explore in realistic time. This tool tries to cope with that issue by proposing a 2-phases scheduling approach combined with a smart exploration of the architectural choices.

The application and the architecture are modeled as following:

- The *application* is decomposed into communicating processes and described using data flow graphs where channels represent the data exchanged between the processing nodes.
- The *architecture* model is actually a set of parameters determining its structure: the number of functional blocks, the bus width, their operation frequency, the communication buffer size. These parameters along with the mapping parameters (assigning a given process or a channel to a functional unit or bus) define the design space that will be further explored.

The performances estimation and architecture exploration takes place in several phases.

First, the application is profiled to perform a data flow analysis and estimate the data transfers between the different communication channels. After that a pre-scheduling operation is carried out based on the previous communication profile to determine the order of execution of the different processes of the application. This results in the System-Level Execution Order Graph (SL-EOG) built once an for all.

Second, an architecture is selected by determining a value for all the parameters describing it: therefore trees representing orthogonal choices are drawn. A n-deep tree represents n successive decisions to be taken so that leaves correspond to a single solution. Tree pruning methods are used to explore them and select a solution that could satisfy user-defined bounds on execution time and area. Based on that architectural choice, a post-scheduling operation starting from the SL-EOG is performed in order to refine the execution order of the processes

matching the architectural constraints. This results in an Architecture-Level Execution Dependency Graph (AL-EDG) that can be analyzed to get a more accurate estimation of the execution time and then fed back to the architectural choice phase to determine the next solution to explore.

This method seems quite interesting since it reduces the number of solutions to compare by using a smart design design space exploration policy. Results are presented that show a 2700 factor reduction in exploration time between exhaustive and custom approaches in the case of on audio/video decoding system while getting close to the Pareto-optimal solutions (more results can be found in the related paper).

EPICURE

EPICUE[31] is a methodology for the specification, verification, performance exploration and synthesis of reconfigurable systems. The main restriction concerns the architecture based on a processor connected to a reconfigurable unit trough a generic interface called ICURE. Besides enabling communication between previous units, ICURE manages all the reconfiguration activity at runtime and comes with an API for the microprocessor allowing the programer to easily trigger context switching of the reconfigurable unit. For specification and verification, EPICURE relies on Esterel studio (see Sec. 4.3.1) that translates afterwards that specification into C. Rather than describing in more details EPCURE, we will put the spell on the embedded exploration tool called Design Trotter.

Design trotter is a JAVA-based tool meant for architecture performance exploration in the context of multimedia applications. The design space exploration is divided into two successive phases: the *system* and the *relative* estimation steps[32]. While the first is used to reduce the design space size to select the most interesting promising architectures, the latter provides the user with refined relative performance estimations of the chosen architectures.

System evaluation The specification, written as a C application[18], is first refined into a *Hierarchical Control Data Flow Graph* (HCDFG). This is a MoC deriving from the data flow graphs and extending it to control and hierarchy as depicted in Fig.4.4. Each HDCFG contains others HDCFG's or CDFG's. CDFG's are composed out of conditional nodes (representing *if, for* statements) and DFG's. A DFG is a structure built of non-conditional nodes exchanging data.

The architecture is defined by a User Abstract Rules file (UAR) defining the

[18]The C used is not fully ANSI compliant but only makes some restrictions on the subset of instructions and imposes a certain writing style.

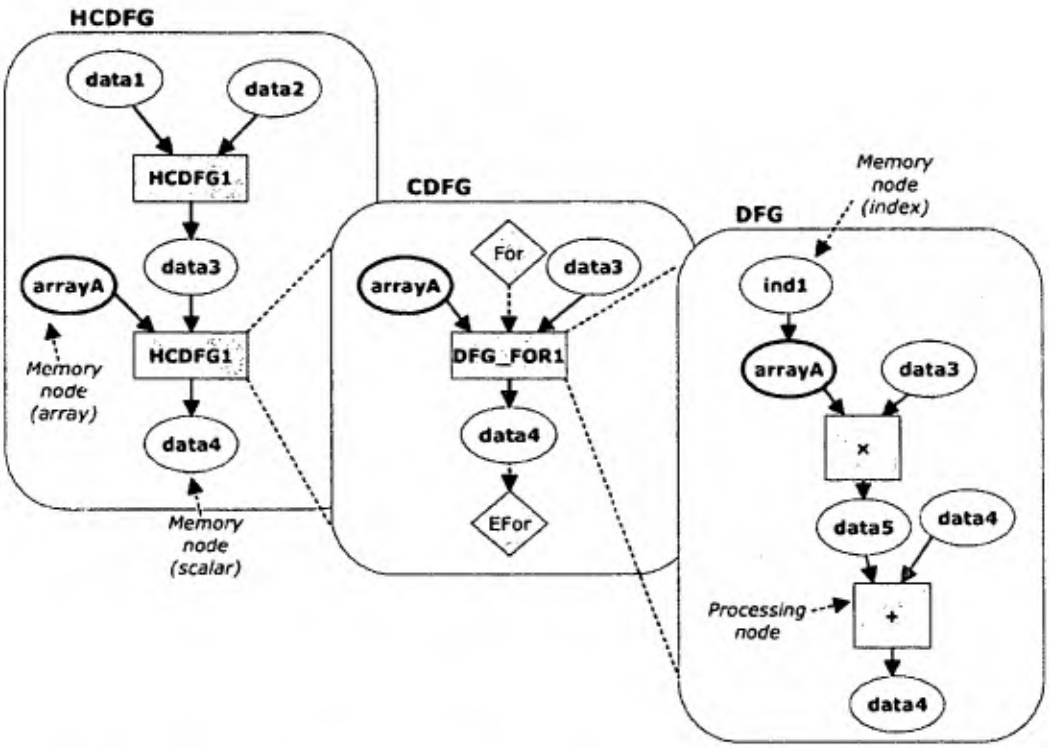

Figure 4.4: The application description in Design Trotter based on hierarchical HCDFG's, CDFG's and DFG's[31]

different resources for computation and the memory hierarchy with their respective costs in terms of relative execution cycles.

Once these files have been defined, the system estimation can take place by first characterizing the application by its computation, memory and control needs using the previously HCDFG. With that information, a list-based scheduling is performed in order to find different compromises between resources requirements and the number of execution cycles which results in cost profiles. Since the number of solutions to explore is very large, heuristic methods are used to guide the choice of the initial number of each resource type.

Relative estimation The previous cost profiles defining a particular scheduling for a given time constraint are used as a starting point for the relative exploration step. Since this estimation is more accurate, it requires a more precise description of the architecture: therefore a hierarchical structure made of functional elements (memory or computation related) is defined. Then the estimation phase takes over, preceded by a *projection* phase (matching between the memory/computation needs of the application and the existing architectural resources) and a *composition* step (taking into account the resources implementing the scheduling).

ROSETTA

ROSETTA[33] is a language meant for describing the constraints and requirements of very heterogeneous systems. Compared to other languages, ROSETTA does not focus on the sole functional aspects but also captures all non-functional aspects that are relevant in the modeled system. Furthermore it is not synthesizable and only partially executable (a certain subset of the language supports execution). However if ROSETTA is very difficult to compare with other languages[19], it obviously deserves some attention regarding its very distinctive approach.
Modeling all relevant aspects (especially the non-functional ones) of an heterogeneous system requires to extend the description syntax and semantics of classical languages. Therefore ROSETTA introduces new concepts:

- *Facets* represent the different functional and non-functional aspects that a designer would like to capture in its design. For instance, they can be related to the power consumption, the delay constraint, the functionality etc. Facets also implement an interface of input and output variables enabling inter-facets information exchange.

[19] In [7], the numerous specification languages reviewed are classified into 3 categories plus one extra category created especially for ROSETTA.

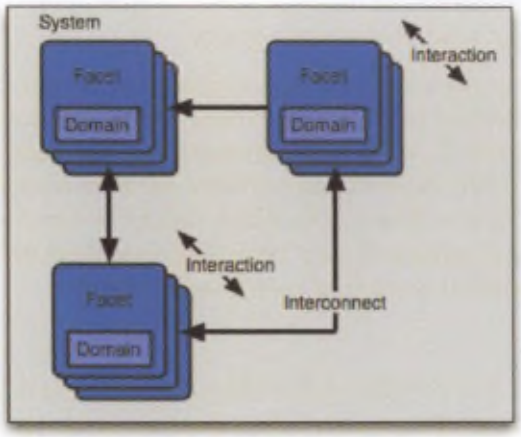

Figure 4.5: ROSETTA representation of a system with interconnected components (facets are defined within a given domain)[33]

- *Domains* define the semantics required to specify a facet. For instance, a state-machine based behaviour will probably be described and integrated differently than a maximum power consumption constraint would be: that's why domains are so important to describe heterogeneous systems. Thus a facet is always defined within a given domain chosen among a ROSETTA predefined domain library (logic, discrete-time, continuous-time, finite-state, constraints).
- *Interactions* define how information are exchanged between different facets that do not necessarily rely on the same semantics. For instance, interactions enable to make synchronous reactive and state-based models coexist and interoperate.
- *Components* are a collection of facets defining all the conditions that a model element should comply with in order to be valid.

The above concepts are illustrated in Fig.4.5 representing a system described in ROSETTA. It is organized as a structure of different components, each one composed out of different facets whose semantics is defined within a certain domain. The different components communicate information using interactions.

The system modeling process basically consists in selecting the different domains the designer he's interested in, defining the facets associated with the previous domains, assembling them into components and instantiating them in turn to form a structure representing the system. ROSETTA comes with an editor and a parser checking the syntax of the user description. Since the language is for non-executable specification purpose only, it is not synthesizable and is meant to be refined manually in other languages. However to be able

to validate the result of further simulations against ROSETTA specification, the latter allows the user to automatically generate vector tests.

4.3.5 UML

UML[34] is a modeling language coming from the software engineering domain: it is able to model a system in a general way by capturing its architectural and behavioural aspects in a graphical way using *diagrams*. UML is particularly well suited for object-oriented analysis and design but can also used in business and organization structure modeling. A few years ago, many people came up with idea of using UML to model VLSI systems both at the functional and platform side. Since the problem is addressed by a more software-oriented community than EDA designers, the solution is quite distinctive and therefore deserves some attention. Compared to more standard functional description languages where code is entered by the user in a textual way, the graphical modeling in UML makes it probably -from the user point of view- less specific and easier to learn. Furthermore UML is a now an amazingly widespread modeling language implying a lot of reusable knowledge and specific tools.

MARTE

MARTE[35] stands for Modeling and Analysis of Real-Time and Embedded Systems and is a UML profile[20] focusing on embedded systems modeling. This profile extends the previous SPT profile (Scheduling, Performance and Time)[36] and will become a future standard approved by the OMG group[21] which probably ensures a certain support for the future.

MARTE relies on two different main sub-profiles:

- *SPT* is a profile used to verify extra-functional properties (like response times, throughput, deadline, etc.) by performing *Scheduling* and *Performance* analysis (more details can be found in [36]). These properties are quite related to timing issues but MARTE developers plan to extend it to other ones (like memory size, power consumption).
- *QoS & FT* (Quality of Service & Fault Tolerance) UML Profile introduces the notion of Quality of Services in an interesting way. Any virtual resource can be used through a *server* providing access to it while *clients* are the entities sending usage requests to the server. The server can provide a certain *offered QoS* and the client needs a *required QoS* to satisfy

[20] A UML profile is a mechanism used to extend the semantics of UML to a particular domain (finance, network representation, hardware modeling etc.). By defining higher-level models with stronger representation capabilities for that specific domain, modeling becomes faster and easier for the designer.

[21] The Object Management Group group is an international non-profit computer industry consortium that strives for the development of modeling standards for the design, execution and maintenance of software. Most of their activity revolves around CORBA and UML modeling.

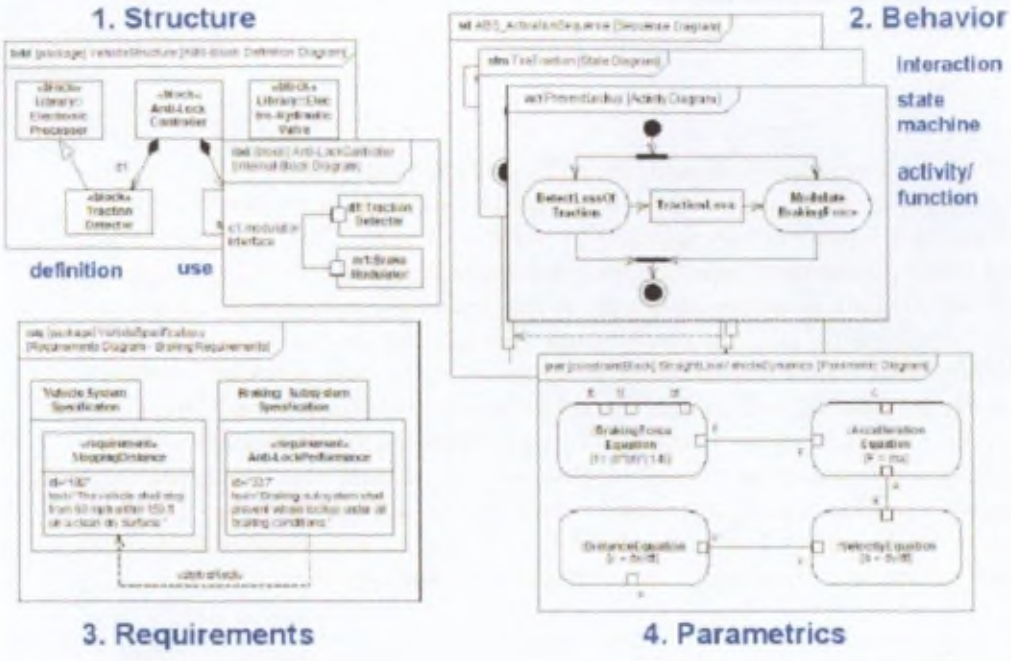

Figure 4.6: SysML system description based on four aspects: the structure, the behaviour, the requirements and the parametric aspect[38]

the *QoS contract*. This definition of the QoS enables analytical analysis to determine if the provided services can satisfy the QoS required by the client.

Besides these profiles, MARTE also defines the QAM (Quantitative Analysis Modeling) domain separating the concerns of embedded system modeling into four different parts: a workload, behaviour, allocation and platform view accounting for both the functionality, the platform and their interaction.

SysML

SysML[37] is an extension of UML for the analysis, specification, design and verification of System-on-chip's. Unlike MARTE that defines very specific profiles with rich semantics, SysML takes a more general approach. As depicted in Fig.4.6, SysML relies on four different major aspects:

- The *structure* aspect describes how the system will be structured both at a functional and platform point of view. SysML defines the *assembly*

concept that allows the user to describe in a hierarchical way any element of the system (hardware or software part, digital or analog etc.) Ports are used to interconnect point-to-point modules and enable data exchange between them. *Block definition* diagrams are used to describe how blocks are interconnected to each other while *inner-block* diagrams describe their respective content creating a hierarchy. Since those profiles don't offer a rich semantic, their high abstraction level may provide the user with very flexible modeling abilities.

- The *behavior* aspect extends four standard UML diagrams to specify the different facets of the system behaviour. The *use case* diagram describes the functional services provided by the block, the *activity* diagram represent the data/control exchange between the different blocks of the system, the *sequence* diagram represents the interaction between the different blocks and finally the *state* diagram describes the different states of the blocks, transitions between them and conditions triggering the latter.
- The *requirements* diagram describes in a textual way all the functional constraints that the system has to fulfill in order to be valid.
- The *parametrics* diagram specifies quantitative constraints on the values of some parameters (delay, power consumption etc.).

To summarize, SysML is a very flexible UML-based SoC modeling tool thanks to its high level of abstraction and hierarchical description of both the functionality and the platform. However SysML pays the price for its generality by requiring a larger design effort and being potentially more ambiguous about the resulting description due to its weak semantics.

KOSKI

KOSKI[39] is a framework for the specification, validation and synthesis of multi-processor SoC's based on UML functional/platform description. Compared to other reviewed methodologies, KOSKI trades some model flexibility and generality for setting up a seamless and efficient design flow from specification down to FPGA prototypying. The philosophy of KOSKI is to derive the best *implementation* (the platform in our acceptation of the term) that meets the specifications using internal optimization processes; therefore a methodology consisting in several well identified phases:

- During the *specification and requirements*, the designer specifies the requirements for the platform and the application by defining the maximum total cost and the way to compute it (the *cost function*). This cost function is a user-defined mathematical combination of several factors (the area, the power consumption, the latency and the throughput) that enables an objective comparison of two different platforms.

- The *UML design* phase consists in defining by the mean of UML models the functionality of the system (application model) that can then be verified using functional simulations. The application description uses the *Khan Process Networks*[25] model of computation to represent the communication of the different sub-tasks. Finally the different possible architectures possibilities are also defined.

- The *UML interface* is a layer extracting simplified and abstracted models from the application and the different architectures that will be used during the automatic architecture phase.

- The *architecture exploration* phase aims at finding the best platform for the specified application: in other words identifying the platform that minimizes the previously defined cost function. To limit the exploration times, this phase is divided into two different sub-phases:

  - The *static exploration* methods evaluates roughly the cost function of different architectures by using an allocation/scheduling method based on a simplified representation of the application. The aim is to find good candidates for the the dynamic exploration phase. It is also important to understand that no functional aspect needs to be verified during this step since it has already been done earlier: the *functional accuracy* can thus be very limited while the focus is now put on non-functional properties of the system.
  - The *dynamic exploration* phase performs a cycle-accurate exploration based on a detailed allocation/scheduling. During allocation, tasks are moved from one processing element to another as long as it doesn't violate the previously established task inter-communication dependence. Thus little change can be done to the previously established mapping: dynamic exploration can thus be seen as a local and deeper optimization step on a limited number of solutions while the static exploration phase is a global optimization step to select good solution out of the numerous possible architectural choices.

- The physical implementation phase takes place after the exploration phase and automatically generates code for the defined target processor(s).

KOSKI comes as a GUI allowing the user to simulate, optimize and synthesize the platform for a given application. Finally KOSKI provides the designer with different possible design scenarios (i.e. a predefined sequence of tools calls) ranging from a single-processor design where no exploration needs to be carried out to complex multi-processors platforms where both static and dynamic exploration methods are required.

4.4 Analysis and classification of literature

The previous section presented several academic and some industrial tools centered around the topic of performance estimation at system-level. Because they are very different from each other and pursue different goals, it is difficult to compare them directly. Therefore we have chosen different criteria that we use to compare different frameworks and extract the common concepts that they share.

The chosen criteria are the following:

- The functionality description
- The platform description
- The allocation and scheduling policies
- The performance criteria
- The design space exploration capabilities

Functionality description

The functional specification is probably the first thing that is defined when starting a design. It captures the behaviour of the system ie the service that it is supposed to provide the final user with: it is thus crucial to examine its description for performance estimation. The functionality is defined in table 4.1 for each previously reviewed tool by its specification language or a model of computation. Different interesting conclusions can be drawn from the interpretation of this table.

First, it is obvious that, regarding the multiplicity of the functional descriptions used by the tools, there is not one model of computation outshining all the others. Depending on the aspects that need to be captured (communication, control/data dominance, memorization requirements etc.), the designer should be able to use the MoC that corresponds the best to its needs. Representing this modeling heterogeneity is the approach chosen by Ptolemy II (but also El Greco in a smaller extent) and explains its success among the embedded design community. However the common point among all these models of computation is that they all try to explicitly expose in a way or another the parallelism of the captured behaviour and its communication needs. This is really important since it allows the designer (and automated synthesis tools) to take advantage of the intrinsic platform parallelism and communication structure.

Second, these models of computation are not necessarily used as an entry point for design: the user has thus to translate manually its favorite language to the format required by the tool which can be quite a burden if not automated. Only a few academic tools (like Design Trotter and Polis) include a parser enabling

Tool	Behavioural specification
Ptolemy II	Hierarchical structure of heterogeneous MoC's (CSP, CT, DE, PN, SDF)
El Greco	Hierarchical structure of heterogeneous MoC's (DFG, Or-model, And-model, gated model)
Esterel	Synchronous reactive language based on concurrent running processes with focus on control
SystemC	Hierarchical structure of concurrent processes containing sequential-based threads
POLIS	Codesign Finite State Machine (CFSM) description
AADL	Hierarchical structure of nested processes
Chinook	Verilog written at behavioural-level[22]
MARTE	A strong semantics UML-profile enabling the specification of heterogeneous MoC's
SysML	A weak semantics UML-profile for the recursive and hierarchical description of a functionality
KOSKI	Functionality based on UML models describing Kahn Process Networks
Mescal	Based on Ptolmey's Discrete Event model of computation
ARTEMIS	Kahn Process Networks
METROPOLIS	Network of concurrent processes with inner sequential C code
SPADE	Kahn Process Networks
MILAN	Data Flow Graphs
A DSE environment	Abstract textual definition of the functionality relying on various embedded models
The SOC explorer	Data Flow Graphs
Design Trotter	Hierarchical Control Data Flow Graphs
ROSETTA	Different domains of functional description (DE, CT, FSM) composed hierarchically

Table 4.1: Functionality specification of the state-of-the-art tools

conversion from a familiar language to a particular MoC while commercial tools put a lot of efforts in automating as much as possible these tasks since they try to reduce time-to-market and manual intervention.

Finally we can also notice that some tools and languages (AADL, SySML, Ptolemy and SystemC) take into account the hierarchical nature of the functionality and thus go for a recursive modeling. This specific aspect will be dealt later on when we will discuss the allocation/scheduling criterion.

Platform Description

As already mentioned, the platform refers to the physical support that will be used to execute the functionality. The platform is also intimately related to the notion of cost: it's only when the functionality uses the different services provided by this platform that performances (execution time, silicon area, energy etc.) can be quantitatively estimated. We thus focus our classification on topological description and physical cost of the platform: table 4.2 summarizes the main related features of the reviewed tools.

From our comparison table, we can see that almost each tool defines its own description of the platform. Indeed unlike functionality whose characteristics are captured inside well-known and -identified models of computation, the platform has no standard description format. This entails a large variety and heterogeneity in the resulting models used to represent these platforms:

- All behavioural tools represented in the table entries have been filled up with *Not relevant* mentions. That's because they do not explicitly represent the platform using a dedicated semantics although they are able to capture the behaviour of the platform.
- Almost all the tools model the platform in a way or another: most of them go for for a structured network of components and only a few ones enable a parameter-based description (like *the SoC explorer* and *the DSE environment*). However the instantiated components are usually chosen inside small and predefined libraries that only include most of the time the computational parts of the architecture. Some tools represent explicitly either the memory or the interconnect part but very few consider the three aspects simultaneously.
- Since platforms can be defined at very high levels of abstraction, the inner components may themselves contain sub-components. If we already came across this recursive-style description in behavioural modeling, it makes even more sense to describe an architecture hierarchically since the components have physical bounds (due to the divide-and-conquer policy used by automated synthesis tools) that make them easier to separate from each other.

Tool	Platform specification
Ptolemy II	Not relevant
El Greco	Not relevant
Esterel	Not relevant
SystemC	Not relevant
POLIS	Only gate libraries are defined for platform synthesis
AADL	Hierarchical structure of components (memory, computation and interconnect)
Chinook	Components interface described in Verilog
MARTE	Specific UML-profile
SysML	UML-profile defining a hierarchical structure of components
KOSKI	UML model of the platform
Mescal	Based on TEEPEE extending Ptolemy with actors modeling the behaviour of processing, communication and memory elements
ARTEMIS	System-C model for components with performance annotation
METROPOLIS	Recursively defined network of processes representing the topology of the architecture with attached *quantity managers*
SPADE	Structure of components (communication and functional units) instantiated from a library
MILAN	Structured description of components (processors, interconnects, reconfigurable units and memories) with annotated execution time and energy consumption
A DSE environment	Abstract textual definition of the platform relying on various embedded models
The SOC explorer	Set of parameters defining the platform (bus frequency, functional unit frequency, bus bit width, buffer size)
Design Trotter	Structured description of memories and functional units
ROSETTA	Non-hierarchical structure of interconnected domain-dependent components

Table 4.2: Platform definition used by the state-of-the-art tools

Based on this platform hierarchy, we can divide the reviewed tools into two different categories: limited hierarchy and recursive hierarchy.
At one hand, limited hierarchy refers to the fact that only a limited amount of well-identified abstraction levels can be used to represent the platform. This approach is commonly chosen by synthesis-centric tools that need intermediate platform representations to get their work done properly: most of the time the lower levels remain hidden to the user and are built automatically by the tool as a netlist of components.
At the other hand, recursive hierarchy is adopted by performance estimation centric tools that define the platform as an unlimited recursive structure of components (like SysML, AADL and METROPOLIS). If this last approach emphasizes the intrinsic hierarchy of a platform by representing it as unfolding trees, its recursive nature involves a shared -thus weaker- semantics for all levels which entails a greater modeling effort.

- As already explained, the platform is just a physical support enabling the execution of the functionality by providing it with several different services: several of the already reviewed tools take advantage of this concept:
  - MARTE introduces the notion of quality of service by establishing the cost and performances of a given platform component: the physical resource is seen as a *server* providing a service to the *client* (the application) for a certain cost.
  - METROPOLIS uses a separate network of processes to describe the functionality and the platform: each platform-related process enables the execution of a given functional-related process. Furthermore quantity managers are attached to the use of the resources so that their usage cost can be evaluated.
  - SPADE defines components with inner execution units, each one coming with a list of symbolic functional instructions that can be treated by this particular execution unit.

  This demand/offer approach is essential when it comes to clearly separate the functionality from the platform and associate costs to the use of a component. Furthermore drawing such a clear and explicit link between functionality and platform makes the work of the mapping tools much easier.

Allocation and scheduling

Once the functionality and platform have been modeled, we need to define *where* and *when* each operation will be executed on the architecture. These two steps respectively called allocation[23] and scheduling are required in order

[23]In the case of the reviewed HW/SW codesign tools, the different tasks forming the functionality are manually dispatched among the HW and SW components. This step is called task allocation and is just

Tool	Behaviour specification
Ptolemy II	Not relevant
El Greco	Not relevant
Esterel	Not relevant
SystemC	Not relevant
POLIS	Manual partitioning, automatic scheduling
AADL	Manual allocation
Chinook	Manual partitioning (verilog processes are tagged by the user), automatic scheduling
MARTE	Behaviour/platform interaction manually described in a UML model
SysML	Allocation specified manually
KOSKI	Automatic allocation of tasks on processing elements and scheduling
Mescal	Manual allocation of the application on the platform support (Service/demand problem)
ARTEMIS	Automatic allocation and scheduling (FCFS, round-robin or user-defined)
METROPOLIS	Manual allocation, automatic scheduling (functional and architectural networks are synchronized to form the mapping network)
SPADE	Manual allocation, manual scheduling
MILAN	Manual allocation and FCFS scheduling
A DSE environment	Script dependent
The SOC explorer	Manual allocation, automatic scheduling in two phases
Design Trotter	Automatic mapping, list-based scheduling
ROSETTA	Not relevant

Table 4.3: Allocation and scheduling methods used by the state-of-the-art tools

to make the adequacy between functionality and platform. The policy used to carry them out will determine in a large extent the performance of the resulting system (resource minimization, execution time maximization, static power minimization etc.). Taking these methods into account is thus crucial in performance estimation: table 4.3 presents the different methods as precisely as possible given the information available in the papers.

Looking at this table, we can see that almost all the tools deal with allocation and scheduling in a way or another. While some of them only provide the support for the mapping representation (for instance AADL and SysML), others completely automate the process (like Artemis and Koski). Scheduling is also much more automated than allocation mostly because the tools often assume

a particular case of allocation

that the designer is in best position to take that type of decision. However, the results of scheduling highly depends on the initial allocation step: for instance it's obvious that allocating heavy computational tasks on slow functional units will lead to poor execution times whatever the chosen scheduling policy. Unfortunately tools that automate both allocation and scheduling usually perform them separately failing to optimize the problem in its whole.

It is also interesting to note that almost all allocation/scheduling methods have fixed policies that are most of the time focused on timing-related variables optimization. In the context of embedded systems where performance is a matter of multi-criteria optimization, tools should provide the user with different policies to choose among depending on the most predominant aspects.

Finally we can also note that SPADE, MESCAL, METROPOLIS and MARTE offer easier allocation by adopting a demand/service approach for functionality/platfrom representation. However none of them offers automatic allocation whereas it could be automated in a very elegant and proper way.

Performance criteria

All the reviewed tools allow the designer to make performance-related measures thanks to simulation or the use of models. However tehre are many differences in the nature and number of these performance criteria[24]: table 4.4 presents for each of them the different types of performance criteria that result from their use.

As we can see most of the performance evaluation revolves around time (usually execution time) probably because it's so important to measure for real-time applications. Chip area (or some measurements of the architectural resources) and power consumption are far less represented than time but deserve some attention since timing constraints are not the only constraints that need to be met in a design. However when these last two performance criteria are considered inside a tool, it's often as a constraint set by the user rather than a result evaluated by the tool for a set of design choices. These constraints are often used for internal optimization purpose only and the user is not able to explicitly compare compromises between all the performance criteria unless he performs repeatedly different estimations while changing the constraints values.

Finally only two of the reviewed frameworks make user-defined performance criteria possible. METROPOLIS embeds that feature inside its event-based

[24]Performance commonly refers to execution time and time-related measurements which is a bit restrictive in the field of embedded systems where many other aspects need to be taken account. To extend the meaning of performance, we have borrowed the term *criterion* from multi-criteria decision analysis: not only does it relate to the fact that performance cover various aspects but it also emphasizes the fact that it's a parameter value the designer is interested in when making a design choice.

Tool	Behaviour specification
Ptolemy II	Execution time
El Greco	Execution time
Esterel	No
SystemC	Time-related measurements
POLIS	Execution time
AADL	Time-related measurements
Chinook	Time-related measurements (latency, throughput)
MARTE	Time-related measurements (response times, throughput, deadline verification)
SysML	Execution time and power consumption
KOSKI	Constraints on latency, power consumption, chip area and throughput can be set by the user and tried to be met by KOSKI during synthesis
Mescal	Execution time
ARTEMIS	Execution time and power consideration are considered during optimization
METROPOLIS	Customizable quantity managers allow the user to define performance criteria (by default only time and energy are included)
SPADE	Time-related measurements
MILAN	Latency and energy
A DSE environment	Domain-dependent (area, time and power consumption are provided as examples)
The SOC explorer	Execution time and area
Design Trotter	Execution time and number of resources (an indirect measure of area)
ROSETTA	The *constraint domain* allows the user to put constraints on the value of any desired performance criterion

Table 4.4: Performance criteria estimated by each state-of-the-art tools

simulation core: quantity managers can be attached to a process whose execution will trigger the latter evaluation. Additionally the user is also able to define custom quantity managers (by integrating a piece of code calculating the value of the evaluated performance parameter) that will automatically be evaluated each time the associated process is triggered. ROSETTA also sets up a way to define non-functional constraints on parameters for each component: however the tool is meant for specification only and doesn't provide any way to estimate them based on the description of the functionality/platform.

Design Space Exploration

Apart from modeling the application, the platform and their interaction, the designer needs to be able to evaluate and easily compare different design choices: therefore many performance estimation tools claim to provide him with design space exploration. Design space exploration, as we intend it, consists in exploring different design choices and results in compromises between the performance criteria. The entire design space is determined by the different design *degrees of freedom* and associated values while the exploration method ranges from exhaustive design space sweep to complex heuristics to only explore specific parts of the design flow. The designer use those methods to have an overview of the impact of choices on performances: he thus has to explicitly define performance criteria, degrees of freedom and the exploration method. The comparison of these different tools regarding the design space exploration method is summarized in table 4.5. First of all, we can see that some tools don't offer any support for design space exploration whereas others have "manual exploration capabilities". However, there is no real difference between these two types: indeed the term "manual" refers to the fact that the designer is able to change some design parameter (regarding the functionality, the platform or possibly the mapping method), make the estimation tool run again and compare the result to the previous ones. Basically any tool generates a set of outputs depending on its inputs: it's obvious that changing the latter values will certainly modify the output values. This leads to the conclusion that any tool is able to perform "manual exploration" of the design space, the only difference being that some papers explicitly claim to support it while other's don't: that's why we made the distinction. We are totally aware that the previous distinction could seem a bit absurd but it shows how misused and overrated the "design space exploration" expression often is.

Aside from the previous tools which don't really provide any special support for design space exploration, we have a few ones that allow the user to define several design choices to compare and explore them automatically afterwards. MILAN, the SoC explorer and KOSKI all work on a same principle of optimization: they define a cost function and try to minimize it for different architectural competitors. This leaves us with ARTEMIS and Design Trotter

Tool	Behaviour specification
Ptolemy II	No
El Greco	No
Esterel	No
SystemC	No
POLIS	Manual exploration (tasks can be swapped from SW to HW or inversely)
AADL	No
Chinook	Manual exploration (tasks can be swapped from SW to HW or inversely by tagging the Verilog code) and scheduling optimization based on heuristics methods
MARTE	Manual exploration (platform and functionality can be changed)
SysML	No
KOSKI	Automatic selection among several architecture candidates based on the estimation of a user-defined cost function (area, power consumption, latency and throughput).
Mescal	Manual exploration (platform and functionality can be changed)
ARTEMIS	Automatic exploration of the mapping methodology based on the computation/communicaiton performances and power consumption resulting in Pareto-optimal curves
METROPOLIS	Manual exploration (platform and functionality can be changed)
SPADE	Manual exploration (platform and functionality can be changed)
MILAN	Exhaustive exploration of all the possible architectures
A DSE environment	Manual
The SOC explorer	Automatic exploration of the architectural parameters trying to reduce the number of explored solutions
Design Trotter	Internal heuristics used to explore different compromises between the execution time and number of resources and discarding sub-optimal solutions
ROSETTA	No

Table 4.5: Design space exploration and optimization methods of the state-of-the-art tools

Tool	No
Ptolemy II	No
El Greco	No
Esterel	Yes (using ESTEREL STUDIO)
SystemC	Yes (using commercial tools to turn SystemC into synthesizable RTL)
POLIS	Yes (Hardware: translation to VHDL, Software: translation to standard C followed by the use of vendor specific tools (FPGA/ASIC synthesis tool, compiling tool)
AADL	No
Chinook	Yes (targeting high multi-processor integrated with off-the-shelf HW components + interface synthesis)
MARTE	No (currently under work)
SysML	No
KOSKI	Yes. Synthesis towards multi-processor architectures SoC's
Mescal	Yes (only towards ASIP's)
ARTEMIS	No
METROPOLIS	Yes (using dedicated external tools)
SPADE	No
MILAN	No
A DSE environment	No
The SOC explorer	No
Design Trotter	No
ROSETTA	No

Table 4.6: Synthesis capability for each state-of-the-art tools

which are the only ones falling into our "design space exploration" definition by resulting in different compromises for the user to finally choose among.

Finally we can also mention that none of the tools is able to simultaneously play with design variables coming from more than one of the three different degrees of freedom proposed in the Y-chart approach (mapping, functionality and architecture). This last point is particularly annoying since it doesn't allow the designer to automatically study how different algorithms match a set of architectures for instance.

Synthesis capability

Aside from the five previous criteria, we provide the interested reader with an additional classification of the different reviewed tools based on the synthesis capability and presented in Table 4.6. As we can see most of the tools don't provide the user with synthesis capability, primarily because we put the focus

on performance prediction when we selected the different tools among the literature. However it is interesting to see that the most abstracted tools offering the most flexible modeling approach have no synthesis capability (like Ptolemy, SySML and MILAN for instance). The reason for that is simple: the closer we need to approach the implementation level, the more details we need to know about the architecture. Dealing with too many implementation details will however lead to a much longer time to estimate one single solution which makes complex design space exploration more complicated: that's why dedicated DSE rather go for more abstracted and less detailed representations making synthesis more difficult.

To cope with that problem, the only way out is to proceed in two steps (like KOSKI) by performing a first wide design space exploration based on rough models of the components and then proceed to local exploration and optimization in a second step. This allows tools to keep synthesis capability while still exploring widely the design space. However HW/SW systems synthesis remains a problem because of the interfacing between these heterogeneous components: KOSKI and Chinook solve that by restricting the number and variety of components that the user can select to build the architecture.

4.5 Conclusions

In this chapter we have reviewed 19 different tools/langugages focusing on VLSI system performance evaluation and design space exploration. We extracted the common concepts and methodologies behind those tools and proposed an original classification to compare them based on five different criteria: the functional description, the platform description, the allocation/scheduling strategy, the multi-criteria nature of performance estimation and finally the design space exploration capabilities. Although these tools all provide the user with some estimation performance related features, their respective concern revolves around different topics ranging from simple specification to complete and automated top-down implementation. In their particular domain, they offer most of the time very interesting ideas so that we may wonder if it is really worth spending time to redevelop another framework for pure performance estimation.

In fact trying to outshine these tools in their specific domain of interest would probably be a mistake: PTOLEMY is unbeatable when it comes to represent a behaviour using heterogeneous modeling, POLIS is one of the rare tools providing a seamless and complete flow, METROPOLIS allows the user to specify custom performance criteria etc. Moreover some of the languages are getting standardized -or at least widely adopted- while other have required a lot of work and are still under development: duplication of the effort is surely not what we are seeking. Instead we could bring together all the different

interesting ideas and concepts behind each of these tools with our own into one unique framework dedicated to pure performance evaluation. For each of the discussed criteria, we would like to improve the following points:

- Functionality and platform are most of the time represented separately using predefined description languages and offering only a limited hierarchy. Modeling them recursively and jointly in a demand/supply style would allow the user to specify costs for the utilization of the platform services and facilitates the further work of the allocation/scheduling operation.
- Allocation and scheduling are performed separately in each tool, are not always automated and policies cannot be customized or rarely selected among a restricted list of choice. Since these mapping methods are crucial in estimating accurately the system performances, they need additional support than what usual tools do provide the user with.
- Performance criteria are often limited to time-related variables only which is too restrictive in the context of embedded system design.
- Design space exploration is probably one of the weakest point of all the reviewed tools. They fail in simultaneously representing the different degrees of freedom coming from the functionality, the platform and the mapping methods which restricts the design space and doesn't enable joint optimization. Furthermore design space exploration is most of the time performed internally inside the tool and does not provide the user with the different compromises based on the performance criteria. When it does, the exploration policy is always fixed and cannot be manually determined.

By coping with all these limitations, we will try to take performance estimation centered tools a step further in terms of generality and flexibility: this is precisely the topic of the next chapter describing our own tool called Nessie.

List of acronyms

ASIC	Application-Specific Integrated Circuit
CFSM	Codesign Finite State Machine
CSDF	Cyclo-Static Data Flow
CSP	Communicating Sequential Processes
CT	Continuous Time
DE	Discrete Event
FIFO	First In First Out
FCFS	First Come First Served
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

HCDFG	Hierarchical Control Data Flow Graph
HW	Hardware
KPN	Kahn Process Network
MoC	Model of Computation
PN	Process Networks
SDF	Synchronous Data Flow
SoC	System-on-Chip
SW	Software
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

Bibliography

[1] J. Russell, "Literature survey: Software performance estimation," University of Texas at Austin, Tech. Rep., 2001.

[2] J. Platin and E. Stoy, "Aspects on system-level design," in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, 1999, pp. 209–210.

[3] J. D. Yule, *The Concise Encyclopedia of Science and Technolog*, McGraw-Hill Professional Publishing, Ed. McGraw-Hill, 2005.

[4] F. Balarin and P. D. Giusto, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[5] A. Mihal, C. Kulkarni, M. Moskewicz, M. Tsai, N. Shah, S. Weber, Y. Jin, K. Keutzer, C. Sauer, K. Vissers, and S. Malik, "Developing architectural platforms: A disciplined approach," 2002. [Online]. Available: citeseer.ist.psu.edu/mihal02developing.html

[6] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong, "Taming heterogeneity?the ptolemy approach," *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 127–144, January 2003. [Online]. Available: http://www.gigascale.org/pubs/393.html

[7] I. Panagopoulos, "Models, specification languages and their interrelationship models, specification languages and their interrelationship for system level design," HPCL,The George Washington University, Tech. Rep., 2002. [Online]. Available: http://hpc.gwu.edu/%7Ehpc/iptools/pub.htm

[8] J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in el greco," in *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*. New York, NY, USA: ACM, 2000, pp. 142–146.

[9] (2007). [Online]. Available: http://www.synopsys.com/products/designware/system_studio/system_studio.html

[10] T. Parks, J. Pino, and E. Lee, "A comparison of synchronous and cyclostatic dataflow," 1995. [Online]. Available: citeseer.ist.psu.edu/parks95comparison.html

[11] F. Boussinot and R. de Simone, "The esterel language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=97299

[12] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proc. of the IEEE, Special issue on embedded systems*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[13] G. Berry, "The foundations of esterel," in *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000, pp. 425–454.

[14] (2007). [Online]. Available: http://www.esterel-eda.com/products/index.html

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[16] P. L. Guernic, T. Gautier, M. L. Borgne, and C. Lemaire, "Programming real-time applications with signal," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1321–1336, Sept. 1991.

[17] O. commitee, "Systemc tlm 2.0 standard," Open SystemC Initiative, Tech. Rep., 2008. [Online]. Available: http://www.systemc.org/downloads/standards/tlm20/

[18] D. Ku and G. DeMicheli, "Hardwarec – a language for hardware design (version 2.0)," Stanford, CA, USA, Tech. Rep., 1990.

[19] W. Mueller, R. Dömer, and A. Gerstlauer, "The formal execution semantics of specc," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*. New York, NY, USA: ACM, 2002, pp. 150–155.

[20] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavango, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," in *In Proceeding of International Workshop on Hardware-Software Codesign*, 1993. [Online]. Available: citeseer.ist.psu.edu/chiodo93formal.html

[21] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Berkeley - center for electronic system design, Tech. Rep., 1992. [Online]. Available: citeseer.ist.psu.edu/sentovich92sis.html

[22] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Scheduling and memory requirements analysis with aadl," *Ada Lett.*, vol. XXV, no. 4, pp. 1–10, 2005.

[23] P. H. Chou, R. B. Ortega, and G. Borriello, "The chinook hardware/-software co-synthesis system," in *ISSS '95: Proceedings of the 8th international symposium on System synthesis.* New York, NY, USA: ACM Press, 1995, pp. 22–27.

[24] A. Pimentel, "The artemis workbench for system-level performance evaluation of embedded system architectures at multiple abstraction levels," *International Journal of Embedded Systems*, vol. 1, no. 7, 2005. [Online]. Available: http://dare.uva.nl/record/221458

[25] K. G., "The semantics of a simple language for parallel programming," *Proc. of IFIP Congress*, pp. 471–475, 1974.

[26] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers, "A methodology for architecture exploration of heterogeneous signal processing systems," *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, vol. 29, no. 3, pp. 197–207, Nov. 2001, special issue on SiPS'99. [Online]. Available: citeseer.ist.psu.edu/article/lieverse01methodology.html

[27] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[28] S. Mohanty, "Rapid system-level performance evaluation and optimization for application mapping onto soc architectures," 2002. [Online]. Available: citeseer.ist.psu.edu/mohanty02rapid.html

[29] O. Bentz, J. M. Rabaey, and D. Lidsky, "A dynamic design estimation and exploration environment," in *Design Automation Conference*, 1997, pp. 190–195. [Online]. Available: citeseer.ist.psu.edu/bentz97dynamic.html

[30] K. Ueda, K.AND Sakanushi, Y. Takeuchi, and M. Imai, "Architecture-level performance estimation method based on system-level profiling," *Computers and Digital Techniques, IEE Proceedings* -, vol. 152, no. 1, pp. 12–19, 14 Jan. 2005.

[31] J.-P. Diguet, G. Gogniat, J. L. Philippe, Y. L. Moullec, S. Bilavarn, C. Gamrat, K. B. Chehida, M. Auguin, X. Fornari, and P. Kajfasz, "Epicure: A partitioning and co-design framework for reconfigurable computing," *Microprocessors and Microsystems*, vol. 30, no. 6, pp. 367–387, 2006.

[32] L. Bossuet, G. Gogniat, and J. Philippe, "Fast design space exploration method for reconfigurable architectures," 2003. [Online]. Available: citeseer.ist.psu.edu/bossuet03fast.html

[33] P. Alexander, *System Level Design with Rosetta (Systems on Silicon).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[34] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language reference manual*. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.

[35] H. Espinoza, H. Dubois, S. Gérard, J. L. M. Pasaje, D. C. Petriu, and C. M. Woodside, "Annotating uml models with non-functional properties for quantitative analysis." in *MoDELS Satellite Events*, ser. Lecture Notes in Computer Science, J.-M. Bruel, Ed., vol. 3844. Springer, 2005, pp. 79–90. [Online]. Available: http://dblp.uni-trier.de/db/conf/uml/models2005sc.html#EspinozaDGPPW05

[36] H. Espinoza, H. Dubois, J. Medina, and S. Gérard, "A general structure for the analysis framework of the uml marte profile," in *Lecture Notes in Computer Science*, S. B. . Heidelberg, Ed., vol. 3844/2006, 2005, pp. 58–66.

[37] Y. Vanderperren and W. Dehane, "Sysml and systems engineering applied to uml-based soc design," in *roceedings of the 2nd UML-SoC Workshop at 42nd DAC*, 2005.

[38] H. Gimbert, "Rapport sur les outils de modélisation des systèmes complexes," Ecole polytechnique - Sysmantic Paris-Region, Tech. Rep., 2007.

[39] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "Uml-based multiprocessor soc design framework," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 281–320, 2006.

Chapter 5

Nessie: Concepts, Design and Implementation

Abstract

In this chapter, we present the design and implementation of our tool for hierarchical performance estimation tool called Nessie. Compared to state-of-the-art tools, Nessie focuses on performance estimation and set up dedicated mechanisms enabling easier and flexible design space exploration by making explicit all the design degrees of freedom (functional-, platform- and mapping-related) and delivering performance criteria defined by the user in return.

The performance estimation is based on mixed mapping/-analytical models (using our tool Yeti) and enable the hierarchical exploration of the platform and functionality. Based on a flexible description of the functionality and the platform, Nessie is able to perform an automatic and hierarchical mapping in three steps (scheduling, allocation and routing) and derive the performance criteria values from this operation.

After presenting all these new concepts illustrated by several small examples, we explain how the framework can be used to automatically generate graphs, activity reports and scheduling reports at the end of the simulation. Some implementation questions will finally be discussed to illustrate the flexibility of Nessie and how it can offer support to easily extend its current functionalities in the future.

5.1 General concepts

To begin this description of Nessie, we will first try to define the general philosophy and main ideas that it relies. To organize this description, we will focus on the criteria that have been established in the previous chapter to classify the different performance estimation related tools. In this section, design space exploration methodology, representation of the functional/platform structure, mapping methodology, performance criteria specification and estimation will thus be discussed in the case of Nessie.

5.1.1 Design space exploration

In chapter 4 devoted to state of the art, we saw that most of the existing performance prediction tools only offer restricted design space exploration capabilities. The input parameters related to the system are often limited to the platform and rarely include the functionality and the mapping process. Furthermore design space exploration doesn't always rely on automated processes so that solutions often need to be estimated one by one. Finally the exploration policy is rarely explicit.
Nessie tries to cope with these limitations by providing the user with a seamless exploration framework able to perform the evaluation of several solutions in a single run according to a user-defined exploration policy.

Criteria and degrees of freedom

Nessie offers a very simple interface to estimate and compare different solutions: a set of inputs defining the different degrees of freedom of the system that the designer can tune and a set of outputs representing performance variables related to different aspects of the system (called performance criteria). This interface is depicted in Fig.5.1 where M inputs (degrees of freedom) and N outputs (performance criteria) are defined: seen from the outside, the estimation of one particular solution just consists in assigning each input a value, triggering the evaluation and observing the resulting outputs. Before discussing further the different capabilities offered by this interface, let us first explain in details the concepts of criteria/degree of freedom considered until now as outputs and inputs of Nessie's performance evaluation core.

Performance Criteria Performance criteria refer to the different variables that quantitatively represent the system quality and that the designer considers as worth evaluating[1]. Nessie doesn't limit the number of criteria or restrict their nature as long as the user is able to define how they are calculated. *Execution time* is the only criterion that is mandatory since the mapping process

[1]The term *criteria* has been borrowed from the multi-criteria analysis field[1] emphasizing the fact that these variables values are used for decision making (see Sec.7.1.1).

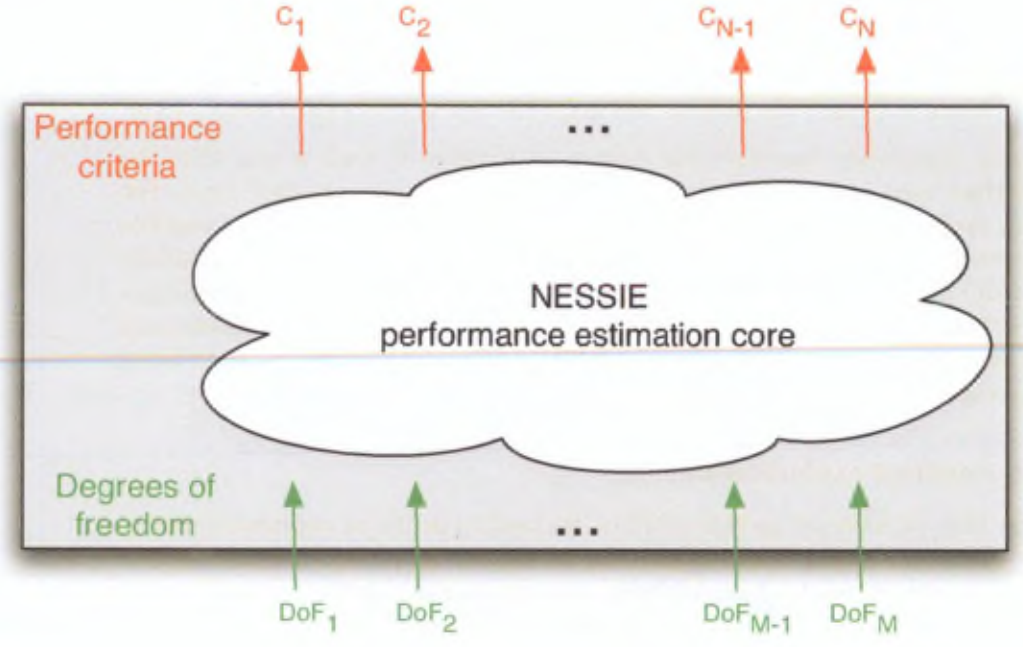

Figure 5.1: Performance evaluation interface offered by Nessie: inputs are the design related degrees of freedom while outputs are the performance criteria

includes the scheduling of the functionality where time plays a crucial role (see Sec.5.4.4).

Degrees of freedom Degrees of freedom are all the variables whose value modification might have an influence on the system performance criteria. Nessie supports different types of degrees of freedom (DoF):

- *Parameterized* degrees of freedom relate to the values that can be assigned to one specific parameter of a Yeti mathematical model.
- *Structured* degrees of freedom describe all the different structured descriptions that can be used to represent one particular functionality or platform primitive.
- *Mapping* degrees of freedom refer to all the possible choices that can be made to modify the process that maps the functionality onto the platform.

Nessie completely banalizes the degrees of freedom in such a way that the interface used to modify their value is the same whatever the DoF type. For each degree of freedom, we can get information about the number of possible values and choose one particular value among them: Nessie will, in a totally invisble and automatic way for the user, do the appropriate changes to take that new DoF value into account whatever its nature. Thanks to this mechanism all the degrees of freedom become banalized inputs which allows us to build any exploration method on top of it.

User-defined exploration policy

Now that we have set up this interface, we need to define an exploration policy for the design space. Many considerations may be taken into account like the time devoted to exploration, the maximum number of solutions to explore etc. In Nessie, the number of possible solutions only depends on the number of degrees of freedom and respective number of possible values: the total number of solutions is given by expression 5.1 where $Number_{DoF_i}$ represents the number of possible values for DoF_i (the ith degree of freedom).

$$NumberOfSolutions = \prod_{i=1}^{M} Number_{DoF_i} \tag{5.1}$$

Fig.5.2 shows how it is possible to connect Nessie DoF/criteria interface for performance estimation to a block driving the design exploration based on a determined policy. The black box on the right part of this figure illustrates any user-defined exploration policy which has access to the inputs and outputs of the estimation core interface. To evaluate one solution, the exploration method defines as input the values of the different design degrees of freedom and gets in return the value of the different performance criteria. These values can then

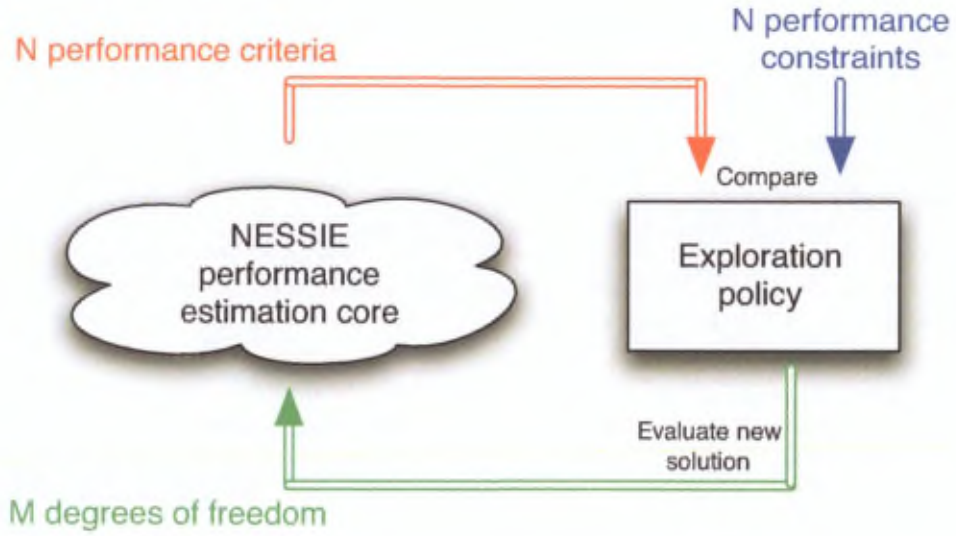

Figure 5.2: Design space exploration based on the Nessie performance evaluation core

be compared with the requirements and can be used to drive the exploration process in an iterative way if required.
We are thus able to build any kind of exploration policy in Nessie thanks to the performance estimation core which is encapsulated inside the interface presented in Fig.5.2. At the moment, Nessie only implements a full factorial exploration policy performing an exhaustive design space exploration[2].

5.1.2 Hierarchy

Along our state-of-art chapter (see Sec.4), we classified the reviewed frameworks into non-hierarchical and hierarchical tools divided themselves into recursive[3] and non-recursive hierarchy. For *Nessie*, we chose to implement a recursive hierarchy both for the platform and the functional specification. As we will now see, this choice leads us to flexibility for functional/platform description and allows us to explicitly specify different algorithms and architecture

[2]This method has been chosen for its simplicity as a demonstration case to show how we can automate the exploration of the design space. The implementation of specific methods to efficiently explore a large space of possible solutions goes outside the scope of this thesis but will be discussed in the future work (see Sec.7).

[3]As a reminder, non-recursive hierarchy implies a limited number of abstraction levels, each of them using a well-defined description semantics and syntax. On the contrary, recursive hierarchy defines recursively each building block as a structure of lower-level blocks so that there is no limitation in the number of abstraction levels.

competitors to be compared.
To formalize our hierarchical approach, we start by clearly defining some important concepts and vocabulary accounting both for the platform and the functionality.

Abstraction levels

The *abstraction level* defines the level of details used to describe the platform or the functionality: the most abstracted a representation is, the more we hide less important details to focus on the most important aspects of the system making its representation more compact hence easier to handle. In a design process, the chosen abstraction level often results from the details at disposal about the system and the understanding level of the system behaviour that the designer desires to acquire.
In Nessie, the different abstraction levels are numbered in consecutive order starting from 0 on top of the hierarchy[4]. Since Nessie implements a *recursive* hierarchy, the definition of an i*th* abstraction level entails the existence of i upper abstraction levels. Furthermore, each functional-related abstraction level must have it platform counterpart in order to preserve consistency of the system description and enable platform/functional matching at each level of abstraction.

Primitives

Each abstraction level requires the definition of specific building blocks to structurally describe both the platform and the functionality: we name them *platform/functional primitives*. Each primitive defines a particular functional operation or platform element that can be instantiated at will to build a *structure*: this primitive instance is simply called a *block*. Each platform/functional primitive will be defined by its abstraction level L and a unique identifier I inside this abstraction level: $Fc_{L,I}$ and $Pt_{L,I}$ notations will thus be used to respectively refer to these two types of primitives.
There is no restriction on the number of primitives[5] in each abstraction level and their definition is entirely left to the user. Depending on the abstraction level, the primitives may thus be very different:

- Functional primitives may vary from complex tasks at the highest levels of abstraction (decoding of a video stream, data encryption operation, storing a picture in a memory, etc.) to simple arithmetic operations (addition, multiplication, etc.).

[4] Let us mention that it is mandatory to define at least the abstraction level 0 (AL_0) since it refers to the functionality and the platform as a whole.

[5] We will also see in Sec.5.1.3 that the number of platform and functional primitives is totally unrelated except for AL_0 of course where only one platform and one functionality needs to be defined.

- Platform primitives may vary from complex computing elements at the highest levels of abstraction (ASIC's, FPGA's, MPU's, etc.) to micro-architectural inner components (multipliers, barrel shifters, register banks, etc.).

Structures

Structures describe a platform/functional primitive based on an organized collection of instantiated primitives of the immediately lower abstraction level. Each primitive may be associated with several structures: they obviously cannot be chosen simultaneously but represent alternative implementations that the designer could be interested in comparing with each other. At the platform point of view, it may enable system or micro-architectural exploration for a given functionality while it makes algorithmic exploration possible at the functional point of view. In other words the introduction of multiple structures per primitive at one abstraction level allows the designer to evaluate the best match between different platform/functional implementation couples. Therefore the structure selected for one particular performance estimation run is one of the degrees of freedom that the user may be able to play with inside Nessie (see Sec.5.6.1).

Fig.5.3 gives an illustrating example of two structures defined for a platform primitive $Pt_{2,5}$ of level 2 abstraction level. These structures are built upon instances of the 5 available platform primitives $Pt_{3,X}$ defined inside abstraction level 3. As an example, the first of the two possible structures ($Struct_0(Pt_{2,5})$) is defined by the connection of two instances of primitive $Pt_{3,4}$, one $Pt_{3,0}$ and one $Pt_{3,1}$ while structure $Struct_0(Pt_{2,5})$ uses a different number of primitives connected differently.

This previous example shows that a structure does not necessarily have to instantiate all the available primitives and that different structures for the same primitive don't have to instantiate the same primitives. More importantly, all the primitives of an abstraction level L share a common pool of primitives of the lower abstraction level $L+1$ to build up structures. However some structures describing a particular primitive of AL_L could be based only on a subset of the AL_{L+1} primitives, the other ones being not relevant. For instance, let us consider a microprocessor and an FPGA being two platform primitives defined at the upper abstraction level and CLB's[6] as a possible primitive of the lower abstraction level. If we could easily imagine different structures of an FPGA primitive based on a different amount of CLB's connected using different network topologies, it's however obvious that microprocessor based structures won't be composed out of CLB's. So it's up to the user to build structures using only lower abstraction level primitives that are relevant in the context of this particular primitive.

[6]Configurable Logic Blocks are small processing elements usually connected by a dense interconnect network and providing FPGA's with their high parallel computing capabilities

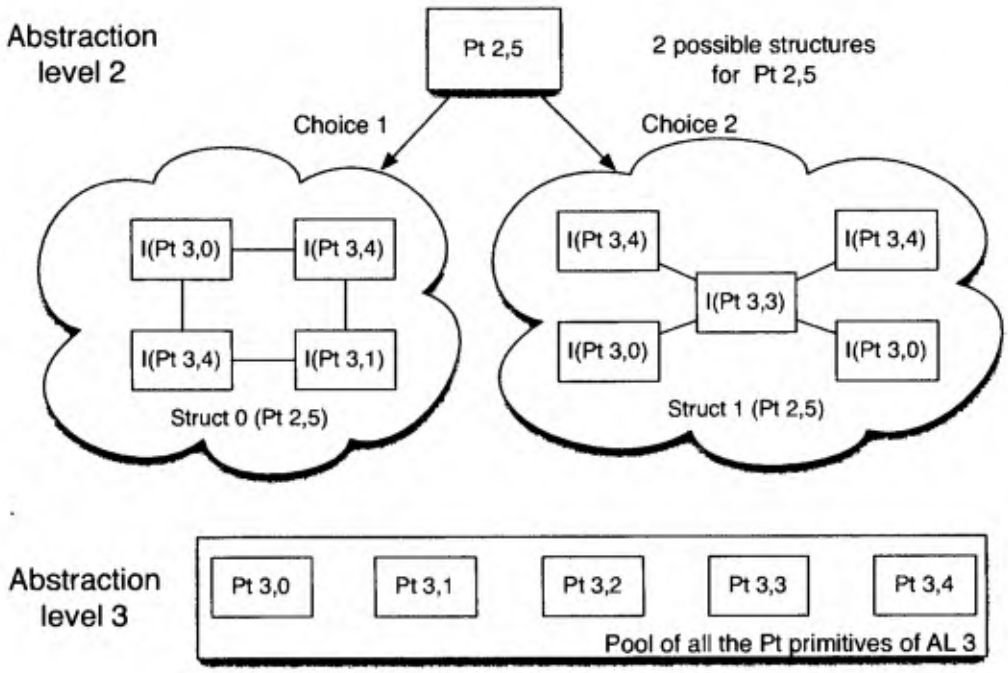

Figure 5.3: Example of two different platform structures build upon a set of lower abstraction level primitives

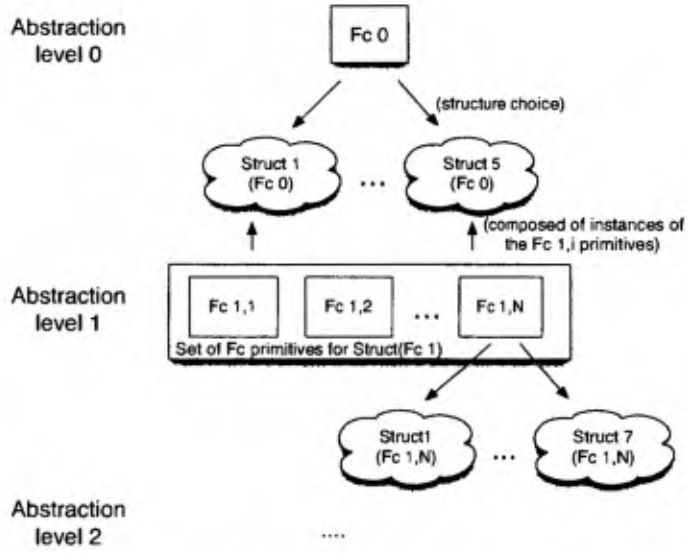

Figure 5.4: Example of two different functional structures for a functional primitive build upon a set of lower abstraction level primitives

Finally it may useful to mention that Nessie assumes that the different available functional structures for a same primitive leave the functional specification unchanged: it means that for a given set of stimuli, the behaviour of the system will remain unchanged.
To summarize, we can illustrate Nessie hierarchy as depicted in Fig.5.1 for the case of the functionality. Starting from the top of the hierarchy (abstraction level 0), we have the functionality of the entire system that can be described in different manners by using specific structures. These different structures are composed out of instances of smaller functional primitive instances (called blocks) of the lower abstraction level and represent different ways to describe the *same* functionality. Each primitive can again be described by lower level structures and so on for the whole hierarchy.

Hierarchy: the big picture

This hierarchical description mechanism shared by the functionality and the platform have several advantages offering a lot of flexibility inside Nessie:

- The semantics of each level is entirely described by the user using primitive blocks.
- The entire hierarchy is simultaneously exposed making trade-offs between depth of exploration and exploration time possible.
- Using the same hierarchical description formalism both for the functionality and the platform allows us to draw strong links between them and

makes the consistency between them easier to establish.

5.1.3 Functionality and platform consistency

Now that we have defined the hierarchical nature of Nessie platform and functional representation, we need to have a closer look on the way they interact. Indeed functionality and platform represent two complementary aspects of the system that need to be jointly represented in order to simplify the task of the mapping process. To do so, we chose a demand/offer approach to characterize the functional requirements and the platform-related services that can be delivered.

A demand/offer paradigm

From the pure functional point of view, it is impossible and absurd to define any kind of implementation cost: only the conformance to a specified behaviour can be tested (using for instance software formal verification methods). Indeed it's only when a functional block has been mapped onto a platform component that the notion of implementation cost appears (execution time, energy consumed to complete the operation, silicon area required, etc.). In a software programer perspective, it may be tempting to believe that execution time of an application can be reduced by optimizing code *independently* from the platform running it (a computer in the present case). Even if we can't deny the potential efficiency resulting from the commonly used methods, there is an implicit knowledge of the platform driving this optimization process: each programmer knows that a microprocessor is a sequential machine fetching instructions of heterogeneous complexity out of a hierarchical cached memory. That's why reducing the number of code lines, organizing table memory accesses to avoid data swapping in and out of the cache memory as well as decreasing the branches number in the code will probably tend to reduce the execution time. For further optimizations, the programmer has to get a better understanding of the platform, hence go for assembly-level code modifications or rely on the compiler to do it. Annotating high-level instructions with cost is another way to hide the platform dependency but these implementation costs are usually drawn from an analysis of the platform performances for any of these possible high-level instructions ([2][3]).
The previous discussion shows that the notion of cost is intimately related to the platform: that's why we decided to clearly separate platform from functionality and adopt an offer/demand[7] interaction mechanism. This is an important choice that highly impacts the way Nessie will work and that's why we spent some time to justify it.
Nessie thus defines the roles of functionality and the platform as following:

[7] It is important to understand that the concept of offer/demand as we intend it has no relation with its economical counterpart.

- The functionality *demands* a certain service to be delivered by the system to meet the *functional requirements*: no cost is associated to the demand. At any abstraction level, each functional primitive $Fc_{L,I}$ thus defines a particular service.
- The platform *offers* a certain service to the system with associated implementation costs which are precisely the *performance criteria*. At any abstraction level, each platform type $Pt_{L,I}$ thus defines *one or several* services that can be delivered by the system with their respective costs.

From these preliminary definitions we can already draw some very important implications.

Independent functional specification The functionality is always specified independently from the platform: this is very important since it gives the opportunity to explore different platform choices without having to modify the functional specification. If this choice may seem very interesting in a performance exploration perspective, it however has the drawback of putting the stress on the mapping tool since no functional customization will be performed for the platform.

Platform implementation costs The platform is only responsible for defining implementation costs associated to the service(s) that it delivers. However knowing about the platform is not sufficient to compute the value of the performance criteria: we first need to define which services of the different platform blocks will be used and their activation order. This is precisely determined by the functionality defining the number and type of services that need to be activated and by the policy used to determine their activation order and the platform components that they will be assigned to. In other words, it's the adequacy between the functionality and the platform performed by the mapping method that will lead us to the estimation of performance criteria values.
Translated into Nessie formalism, it means that mapping any functional type $Fc_{L,J}$ instance onto a platform primitive instance $Pt_{L,I}$ will result in the estimation of the criteria defined by the user. This is illustrated in Fig.5.5. that identifies clearly the three degrees of freedom (platform, functionality and mapping policy) and shows how performance criteria are estimated.
Nessie thus defines performance criteria as being the values exchanged between abstraction levels to enable their interaction and makes their evaluation mandatory for any platform/functional primitive adequacy whatever the abstraction level.

Functional/Platform compatibility A particular platform component may be able to deliver more than one service, each one at the price a different cost i.e. different performance criteria values. Furthermore, one particular service can be delivered by more than one platform primitive: if this will make

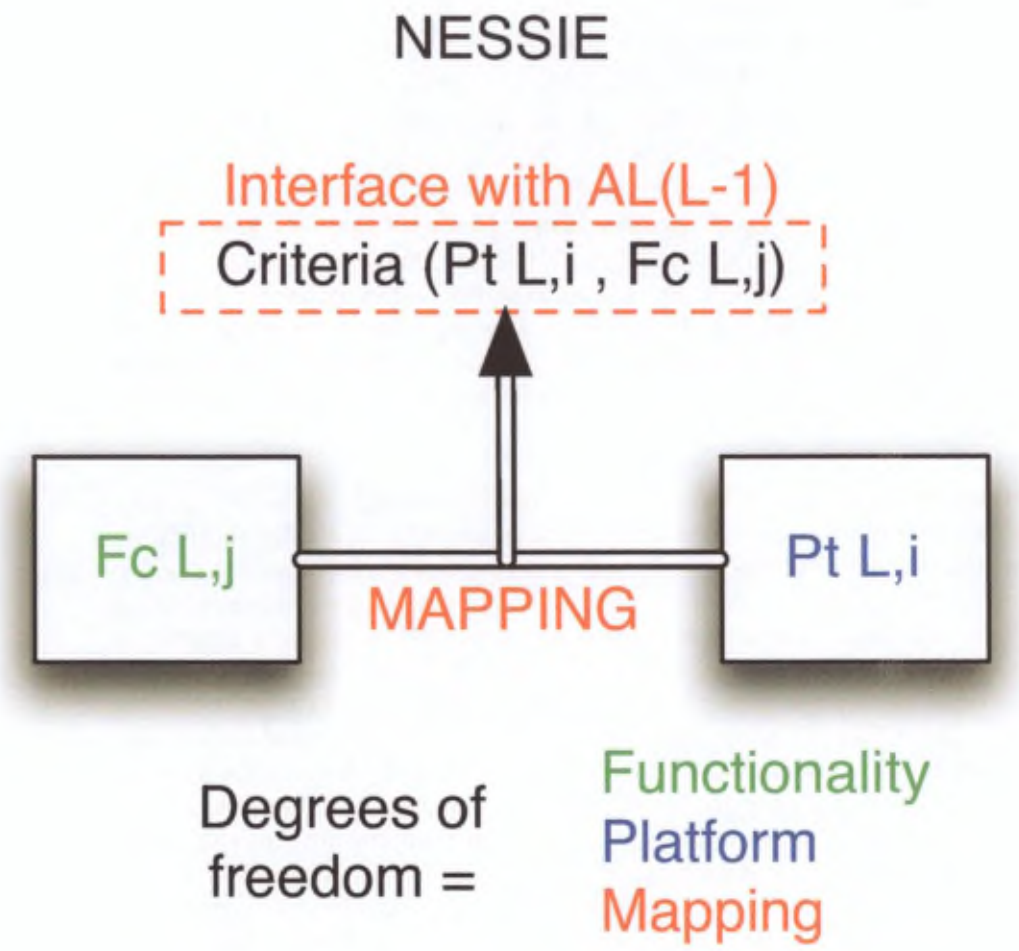

Figure 5.5: The three degrees of freedom (functionality, platform and mapping) used by Nessie to estimate performance criteria values

no difference at the functional point of view, it will lead to different possible performance trade-offs and leave room for choice during the mapping. For instance let us consider an FFT operation and two different platform components able to deliver that service: a low-power micro-controller and an FPGA. If we choose power consumption and execution time as the two relevant criteria, the FPGA will probably execute the FFT faster than the micro-controller but at the price of a higher power consumption.
In Nessie, each functional type defines a service required by the system and each platform type comes with a *list of compatibility* containing the different services that it is able to provide the system with and the costs associated to it.

5.1.4 System hierarchical exploration

In the previous subsections, we discussed the hierarchical approach chosen for the representation of the platform and the functionality inside Nessie. However we didn't explain how the exploration of this hierarchy can be performed neither how it is possible to determine the depth of exploration: in the current section we deal with these questions.

Explicit mapping VS Yeti model

Recursive hierarchy models both the functionality and the platform as trees where each node represents a primitive that can be described by a structure composed out of lower abstraction level primitives i.e. children of that node element. However we cannot unfold this tree forever: at some abstraction level (i.e. a certain tree depth), we need to define *atomic primitives* that will be tree leaves and won't be further decomposed hierarchically.
Defining how the functional/platform hierarchy will be explored consists in distinguishing the primitives that will be deeper explored through the hierarchy from those which won't. Estimating the performance criteria resulting from the mapping of a $Fc_{L,j}$ block on a $Pt_{L,i}$ can be done in two different ways that are illustrated in Fig.5.6:

1. *Explicit mapping* of the functional structure related to $Fc_{L,j}$ on the platform structure related to $Pt_{L,i}$ (left part of Fig.5.6). This method will estimate the performance criteria and perform the mapping of the functionality onto the defined platform based on the recursive criteria estimation of the immediately lower abstraction level primitives. This mapping process is quite complex and will therefore be detailed in a specific section (see Sec.5.4).
2. *Yeti model-based* performance estimation (right part of Fig.5.6). To estimate performance criteria of atomic primitives[8], we have chosen to in-

[8]Atomic primitives refer to the fact that they will not be explored further in the hierarchy. They are

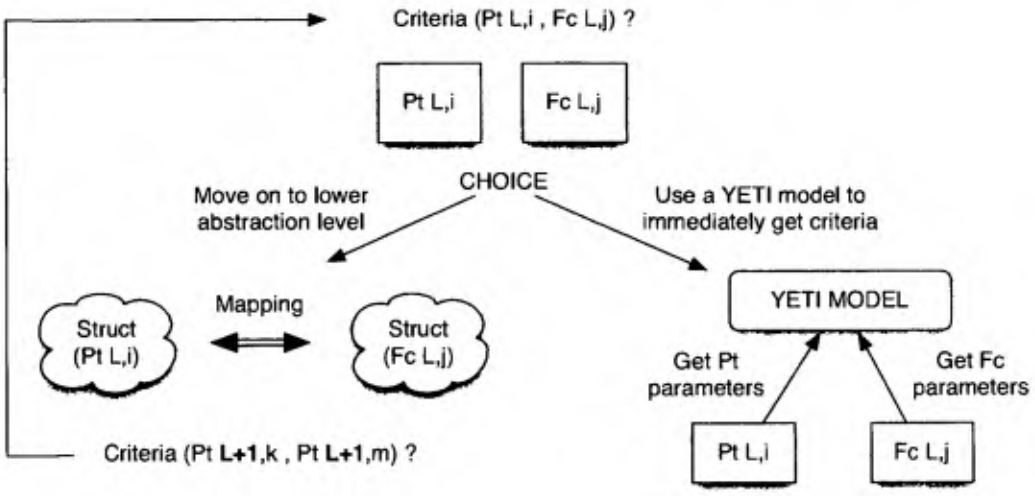

Figure 5.6: Estimation of performance criteria of a functional/platform primitives pair through explicit mapping or the use of a Yeti model

terface Nessie with Yeti models to take advantage of its flexible modeling capabilities. This model is fed with *parameters*[9] capturing the characteristics of the functionality and the platform (more details are given in the following section 5.1.4.

Explicit mapping has the advantage of using more detailed information about the functionality and the platform than Yeti models do: results are therefore expected to be more accurate. This accuracy gain however comes at the price of an increased exploration time: choosing between the two methods thus results from a compromise between accuracy and time devoted to exploration.
In its current version, Nessie provides implementation mechanisms for defining different hierarchical exploration policies in a flexible way. At the moment, we have implemented as a demonstration example a full-depth exploration policy: hierarchy will be explored as deep as possible until atomic primitives are reached. Many other policies could be considered: limited abstraction level policy (the exploration goes on until abstraction level L is reached), estimation time driven policies (limiting the exploration depth based on an upper bound for estimation time), etc.

---

defined by user's structure related degrees of freedom and not by the existence of the structures for this described primitive.

[9] Parameters have already been defined in the chapter devoted to Yeti (see Sec.2.4.1). As a reminder, they are simply named variables with an attached floating point value.

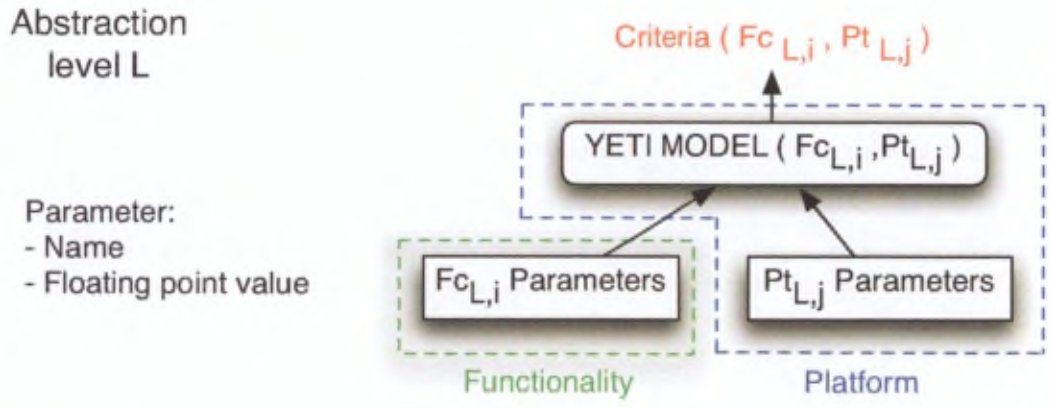

Figure 5.7: Yeti modeling for performance criteria estimation based on functional and platform parameters

Yeti for performance criteria estimation

If Yeti is also a stand-alone tool, it has originally been designed to provide Nessie with fast performance criteria estimation based on analytical models. Its flexibility makes dynamic model loading and execution possible which are features used in Nessie to build models at run-time and define Yeti parameters as possible functional/platform degrees of freedom. When we don't want to further explore the hierarchy of the different abstraction levels, Nessie uses a Yeti model capturing the platform and functional characteristics to estimate the performances criteria.
Fig.5.7 depicts how a Yeti model is defined at a particular abstraction level L to estimate the performances criteria of a $Fc_{L,i}/Pt_{L,j}$ combination and shows that the system is divided into two different parts:

- The *functional* part consists of a set of parameters relative to the i*th* functional primitive of abstraction level L.
- The *platform* part consists of a set of parameters relative to the j*th* platform primitive of abstraction level L but also includes the model representing the link between the functional/platform parameters and the criteria[10].

To link the model input parameters with the functional/platform parameters, Nessie simply proceeds by name identification. It means that all the input parameters defined in the Yeti model need to contain the union of the $Fc_{L,i}$ and $Pt_{L,j}$ parameters, otherwise Nessie will generate an error. This name

[10]We could wonder why the Yeti model used to compute the criteria values is part of the platform and not the functionality. We made that choice because the platform precisely determines the cost of the system (the criteria in our vocabulary) and that's why the model is attached to the platform.

binding is realized once and for all at the initialization of a Nessie performance estimation run to save as much execution time as possible for the exploration phase itself. Thanks to that name matching, a single change of the parameter value of a given functional or platform primitive will immediately affect all the models where it is involved. Thanks to that mechanism it becomes very convenient to associate a parameter with a degree of freedom and modify its value.

Furthermore it is very important to mention that one model is defined for each combination of a functional $Fc_{L,i}$ and a platform $Pt_{L,j}$ primitive. This allows the user to define a different model for each possible couple of primitives giving him a lot of flexibility to define the way a functionality matches a platform. Parameters are not necessarily the same for all the models of a particular abstraction level so that a functional/platform primitive simultaneously encapsulates all the parameters required for the execution of all possible models.

With the mechanism of parameters presented so far, there is still one big problem left that cannot be solved by name identification only: the parameter name and value consistency between the different abstraction levels and primitives. The upper part of Fig.5.8 illustrates the example of two abstraction levels L and $L+1$ for a system where power consumption is part of the performance criteria. Therefore we define Yeti models for each primitive including a supply voltage parameter named "Vdd". At abstraction level L, an architecture composed out of different components connected to the same supply voltage will thus be represented by a structure of blocks of the four platform primitives, each of them with a parameter called "Vdd". However to have a common supply voltage, we will have to define for each of these parameters the same value implying as many value changes as primitives and leading to potential consistency problems. Furthermore if we would like to measure the overall impact of a particular block supply voltage change for instance (block $Pt_{L,3}$ on the figure), there is no information available for the user to select the right parameter among all the parameterized degrees of freedom named "Vdd". To solve these ambiguity and consistency issues, we have decided to separate parameters into two categories: *global* and *local* parameters.

- A *global parameter* is defined for all the abstraction levels and all the primitives. Each time a new model is created, each input parameter with a name corresponding to a global parameter name will directly be associated with it. Only one degree of freedom is created for this parameter and changing its value will affect all the models where this parameter is used which prevents several parameters with the same name from having different values solving the problem of inconsistency.
- A *local parameter* derives from a global parameter and particularizes its value for a given primitive at a particular abstraction level. This allows the user to define a default value by assigning it to a global parameter

and by customizing it for a particular primitive by using a local parameter with the same name. However to define a dedicated degree of freedom for each local parameter and avoid name ambiguities, Nessie automatically renames the local parameter with a suffix referring to its nature (functional or platform related), abstraction level and primitive identification key. To simplify user's model definition, models always refer to the original global parameter name instead of the local parameter: Nessie will adapt the names and correctly link the local parameters to the model.

Using this mechanism of global/local parameters, we can see (lower part of Fig.5.8 that we now have as many degrees of freedom and parameter names as values removing both ambiguity and consistency problems.

5.1.5 Summary

Taking back the different classification criteria established for the tools reviewed in the state of the art, we can explain how Nessie performs compared to previous tools:

- *Platform and functional description*: functionality and platform are both hierarchically described by structures based on lower abstraction level primitives. As an alternative for structure, each primitive may also be described by a collection of parameters. Functionality description is totally independent from the platform and has no notion of implementation cost contrarily to the latter.
- *Allocation and scheduling*: the offer/demand paradigm adopted for the description of functionality/platform clearly defines their interaction and sets a good basis for the mapping. Allocation, scheduling and routing have several degrees of freedom allowing the user to try and compare different mapping strategies.
- *Performance criteria*: they are defined by the user based on the relevant aspects of the system to represent and compare. There is no restriction on the number of criteria as long as we are able to define how to compute their value and combine them. Only execution time is a mandatory performance criterion since it is required by the scheduling process.
- *Design space exploration*: Nessie performance estimation core relies on a DoF/criteria interface so that any user-defined exploration policy can be implemented. Additionally, Nessie also supports the specification of user-defined hierarchy exploration policies defining how deep in the abstraction levels the hierarchy should be explored.
- *Synthesis*: no synthesis is currently supported, Nessie focuses on performance estimation.

From these different criteria, we can see that, compared to other tool, Nessie is especially meant for performance estimation and design space exploration and combines many mechanisms to offer as many flexibility as possible.

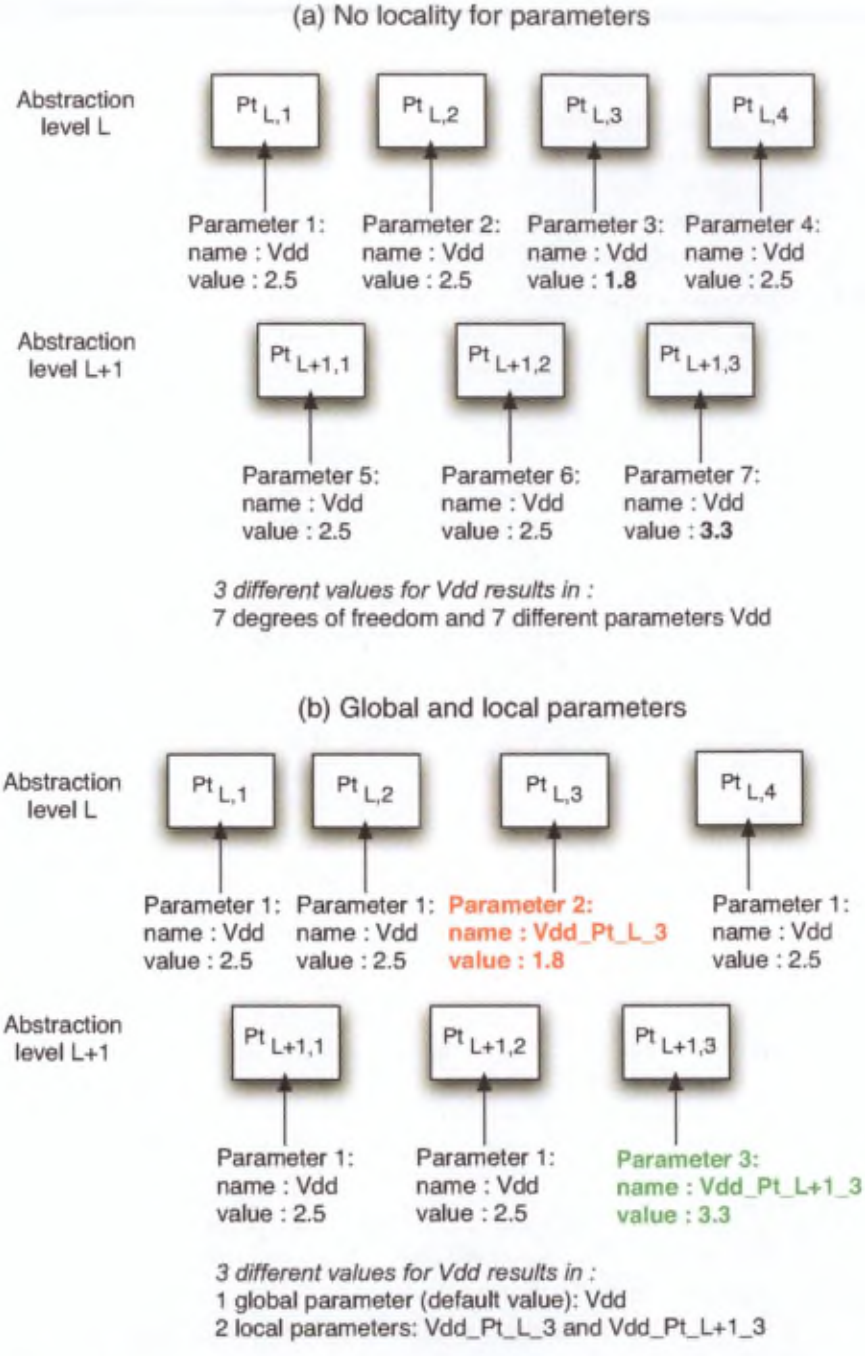

Figure 5.8: Locality for parameter definition: a) shows the case of a system where a common parameter "Vdd" is defined for each primitive while b) shows the same example with a global parameter for all blocks and local parameters overriding the value of the global parameter for different primitives

In the next section, we begin the complete description of Nessie with the platform and its related models.

5.2 Platform description

So far we have described how Nessie allows the user to hierarchically describe the platform using our *structures* built upon lower abstraction level primitives. In this section we present in details how those structures can be described for the platform side of the system.
Contrarily to functionality, there is almost no standard format for structured platform description so that many exploration tools (see Sec.4.4) choose to define their own format. In our case we also decided to define our own format in XML to ensure compatibility and guarantee easy integration with Yeti file format.
To define a structured description format for platforms, we established the following requirements:

- Ensuring an easy recursive hierarchical description of the platform that remains valid and unique for all the abstraction levels
- Offering a way to describe all the elements of an architecture whose implementation involves a certain cost (not only the computation blocks but also memories and interconnects).
- Providing a mechanism to compute the performance criteria of a whole structure based on the criteria values of the individual platform blocks

The next subsections respectively discuss those three different questions.

5.2.1 Hierarchical platform structures

To define platform structures we went for a netlist-based approach describing the different instantiated primitives and how they are connected. Basically XML files describing these netlists contain a list of all the platform blocks and a list of *point-to-point* connections between these blocks (more details will be given in Sec.5.4). This XML description is then automatically turned into a platform structure attached to the corresponding platform primitives. The resulting structure related to a platform primitive $Pt_{L-1,i}$ is composed out of three different types of elements:

- *Platform blocks* are instances of the platform primitives $Pt_{L,X}$: since all these primitives of abstraction level $L-1$ may also themselves be described using structures we obtain the desired recursive hierarchy.
- *Ports* are communication interfaces for the platform blocks and offer a connection point for a link to another block. Depending on the abstraction level, ports may represent very different elements of an architecture:

from a FIFO buffer in a network-on-chip router to a potential latch between two logic gates for instance. Because ports may represent so many various components depending on the considered abstraction level, the user is allowed to associate performance criteria representing their implementation cost[11].

- *Logical links* connect two ports together and establish the communication topology of the different platform blocks. These links are not to be mistaken for interconnect related elements which are real components with implementation costs contrarily to logical links that don't add any overhead to performance criteria. Interconnect related elements need to be modeled as platform blocks instead since they participate in the total implementation cost of the system.

To illustrate these concepts, Fig.5.9 depicts an architecture composed out of two microprocessors and one memory connected through a shared bus. This architecture can be modeled as a structure describing a platform primitive $Pt_{L-1,i}$ based on three primitives of the immediately lower abstraction level L representing the microprocessor, the memory and the shared bus. Instances of these primitives are exchanging information through ports and logical links defining the communication paths.
From this example, we can see that the interconnect part of the architecture is modeled explicitly by a platform primitive. This shared bus is connected to the other platform blocks by the mean of three ports: this shows how it is possible to connect several blocks to a shared communication medium despite the point-to-point connections established by the logical links between the different primitives.

5.2.2 States

Operation on data

Nessie focuses on applications described from a data-centric point of view: each functional block produces and consumes data that are exchanged with other blocks. To make the parallel at the platform point of view, the platform structure uses the concept of *data token* which is an amount of data of a given size required or produced by a given platform block. In Nessie, we identified three different and not mutually exclusive basic operations that can be performed on these data tokens by a platform block:

1. *Data processing*: the aim of a platform is to provide the material support for the functionality specified by the designer which can be expressed in a data-centric point of view as transforming an input data set into the

[11] Ports are required to properly describe the structure and their presence is therefore mandatory. In the case where ports would not have any relevant implementation in the real architecture, the user can associate them with a null contribution to the performance criteria.

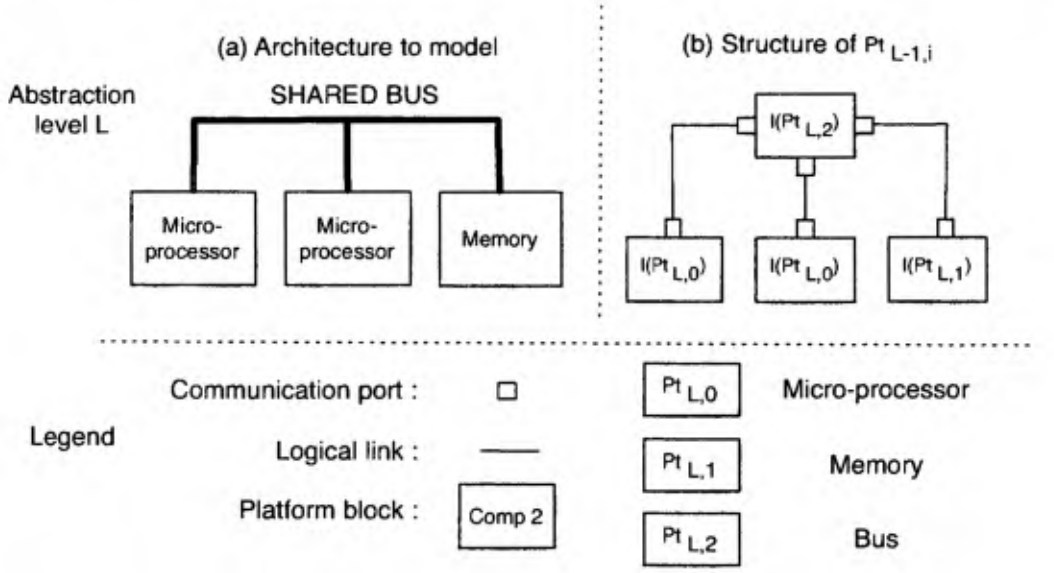

Figure 5.9: Example of platform structure modeling an architecture based on a shared bus communication medium

required output data set. A platform block performing such an operation is usually referred to as a computing node or processing element.

2. *Data movement*: a platform structure is built of several blocks performing computational operations and therefore need to exchange data between each other. The ability to move data across the surface of a platform is therefore essential to feed computation blocks with input data and is carried out by all interconnect resources devoted to communication on a chip.
3. *Data storing*: computing nodes need to be fed with input data resulting from the processing of other nodes. However data are not always required just after they are produced so that storing them until they need to be used is very important: this concept refers to memorization.

So far we have identified three different types of operations on data: computation, communication and memorization. From the designer point of view, the functional specification directly defines the different computation operations to be performed while memorization and interconnect are unavoidable overhead required to map the functionality onto the platform[12].

[12]This justifies the choice that we need to make in the functional specification where only computation related operations need to be made explicit while memorization and communication is drawn from it to best match the platform (see Sec.5.1.3).

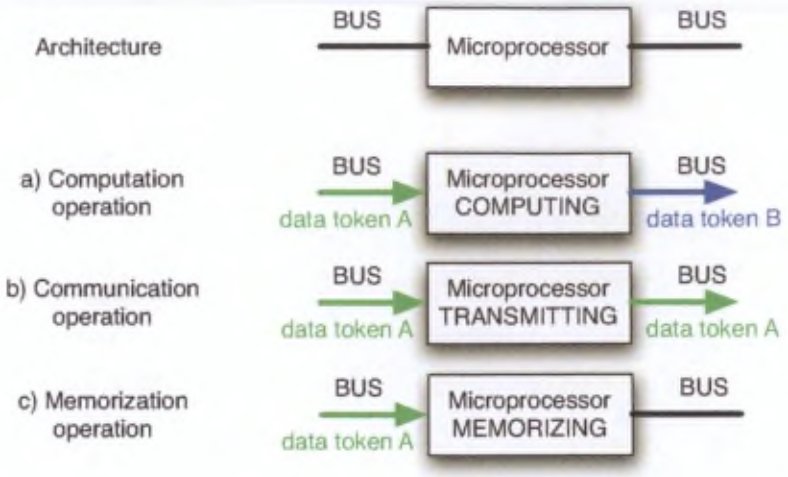

Figure 5.10: Example of a microprocessor connected to two busses able to perform data memorization, transmission and computation operations

Associating a platform with a single data manipulation operation (communication, computation and memorization) is not always simple and representative of its real operation mode.
For instance, let us consider a microprocessor connected to two different busses like the one depicted in Fig.5.10: the processor may manipulate data in three different ways. First the microprocessor can use a data token A coming from one bus, perform a computation operation and put the resulting data token B on the other bus (part a) of Fig.5.10). Second, the microprocessor could act like a bridge in a communication network and forward the same data token A from one bus to another: in this case the microprocessor acts like a communication element rather than a computation element(part b) of Fig.5.10). Finally, if the microprocessor has an internal memory it could use the data sent over the input bus to store them until another computation element requires to use them (performing a memorization operation in that case, part c) of Fig.5.10). From this example of a simple processor we understand that assigning one single data operation type to a platform block is too restrictive. Rather than defining strict and isolated data operation based categories, we thus need to find a mechanism able to represent how a functional block data operation is able to evolve over time avoiding to attach this information to the block itself: the following section deals with that issue.

Application to Nessie

As explained earlier, each platform block is composed out of a *core* connected to logical links by the mean of communication ports. To represent the evolution

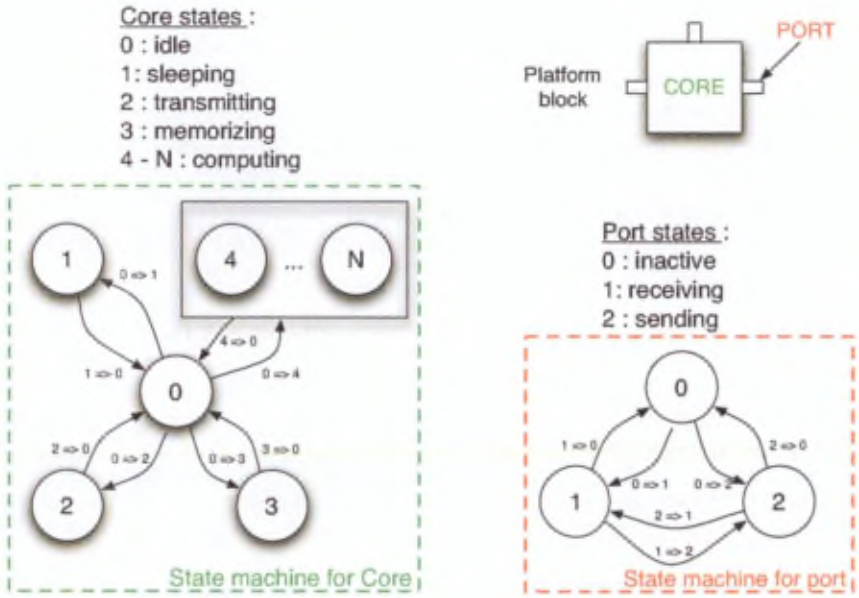

Figure 5.11: State machines and their transitions associated to the core and ports of a platform block

over time of the core and its ports data operation mode, we define separate state machines for each port and core as depicted on Fig.5.11. As in any state machine, one state may be occupied at a time which reflects the fact that a core or a port may only be in one single data operation mode at a time.
Let us first examine the core (left part of Fig.5.11) and define the different data operation states that it may occupy:

- *Transmitting* state relates to any operation sending a data token from one block to another
- *Memorizing* state relates to a storing/fetching data token operation
- *Idle* state means that the platform block is currently associated with no data operation but is ready and waiting to execute any operation
- *Sleeping* state is an additional state provided to represent power savings modes that may partly switch off the platform block at the price of a wake-up time.
- *Computing* state is particular in the sense that it simultaneously encapsulates different states relative to the functional primitive compatible with the platform primitive associated with the current platform block. There are as many computing states as functional primitives compatible with the platform primitive.

Each transition from any state to another is allowed[13] and the user is able to specify the transition times for each possible transition by adding a double entry *transitional time table* to the state machine description. This feature can be used to represent any latency between state switchings: for instance, it can model the wake-up recovery time from sleep to idle mode of a given component in power-save mode or the time overhead due to context switching between two distinct threads in a microprocessor.
Each port instantiates its own state machine defined by the following possible states (right part of Fig.5.11):

- *Inactive* state is occupied by a port when it's not sending nor receiving any data
- *Receiving* state refers to the reception of a data token
- *Sending* state refers to the emission of a data token

Each state must of course be associated with all the user defined criteria but some states require additional criteria in order to provide the mandatory information for the scheduling and routing operations performed during a functional/platform structure mapping. Fig.5.12 summarizes all the required criteria for each core or port state: besides the user defined criteria, we have:

- *Latency* is a criteria required for transmitting and memorizing core states but also receiving and sending port states. It represents the time separating the activation from the real beginning of a data token transmission operation[14].
- *BW* is a criterion needed for the states that also require latency. It represents the data rate per time unit associated to a particular data memorization or communication operation. For instance, the bandwidth associated with a memory state represents the amount of data per time unit that can be stored/fetched in/from a memory.
- *time* is a criteria exclusively required for the core computing states: it represents the time required by the current platform block to execute the functional primitive associated with the computing state.

Besides the criteria required for each data operation type, Fig.5.12 illustrates the fact that all these criteria can only be modeled by the use of a Yeti model except for computing states where an alternative explicit mapping method can be used. This highlights the fact that criteria resulting from the mapping of a functional block on a platform block can be computed using a recursive evaluation based on the lower abstraction levels as explained in Sec.5.1.2. However this leaves still one question unanswered: how it is possible to combine the

[13] To avoid Fig.5.11 from being overloaded with arrows, only transitions from/to state 0 *idle* are represented in the case of the core state machine.

[14] Further information about the latency will be given in the section devoted to data token routing (see Sec.5.4.5).

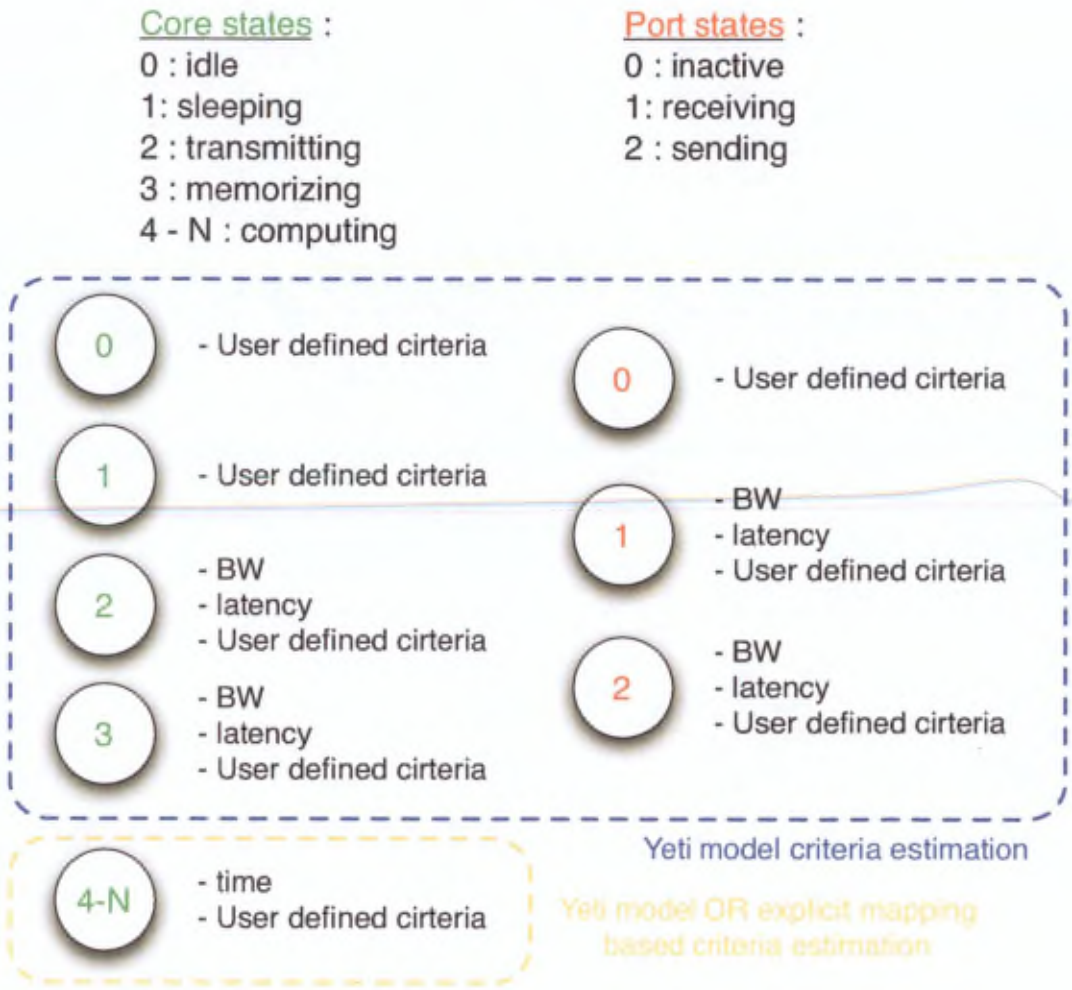

Figure 5.12: Mandatory criteria for port and core states and their evaluation method

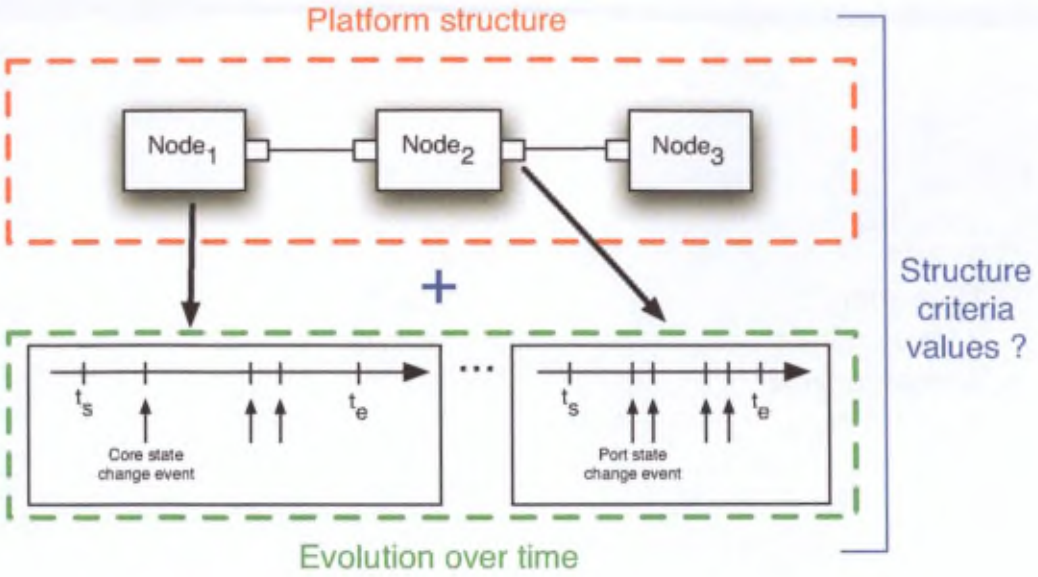

Figure 5.13: Criteria estimation for a platform structure based on time and structural information

performance criteria values of the different blocks composing a platform structure to derive the global performance criteria values of this structure. This is the question that the next section tries to answer.

5.2.3 Criteria integration

We have seen that performance criteria values can be directly estimated based on a Yeti model but how it is possible to derive them when we perform an explicit mapping? Therefore we first need to anticipate a bit on the mapping process description[15].

General approach

Basically the mapping proceeds by successively allocating and scheduling the different functional blocks on the platform blocks. This process results in the complete scheduling of the functionality and the evolution over time of all the port/core states of each platform block of the structure (called the *platform activity*). The activity of the platform structure and the knowledge of the performance criteria for each state of each block composing this structure (obtained thanks to a Yeti model or through recursive mapping) are required

[15] More details will be given in Sec.5.4 devoted to the mapping process.

to compute the performance criteria values of the whole structure. This is illustrated by Fig5.13 depicting a platform structure composed out of three interconnected platform blocks representing two computation nodes (nodes 1 and 3 on the figure) connected by an interconnect link (node 2). For each port/core of any block, we see a timeline where each timestamp represents a state change hence different criteria values for this block. Starting time t_s and ending time t_e are common for all the timelines of the platform blocks composing the structure and respectively represent the absolute time of the beginning and the end of the scheduling: their difference is thus the execution time. This figure points out the essential role of time and platform structure in the formulation of the problem: it's only by combining them both that we can obtain the value of performance criteria for the whole structure.

Criteria integration over time and structure

Nessie proceeds in two different steps to compute the criteria values of the platform structure:

1. For each platform block, the value of each criterion over the whole timeline is computed based on the evolution over time of the criterion value according to a *time-dependent rule*. This phase is relative to the time integration of the criteria.
2. Based on the criteria values of each platform block, the value of each criterion is computed for the whole structure by using a criterion specific *composition rule* over space. This phase is relative to the structure-related integration of the criteria.

By defining these two rules, a user is able to define any criterion and the way it is calculated: this is the mechanism that Nessie relies on to provide the user with flexible multicriteria definition. Let us now define these two types of rules.

Time-dependent rule A criterion is defined by its time dependency thanks to a *time-dependence rule*: based on the evolution over time of the criterion, the final value for the whole timeline duration can be computed. A time-dependent rule operates on a timeline where each timestamp refers to an event representing a potential change in criteria value. If we suppose that we have a timeline with N events each of them corresponding to a timestamp t_i (with $i = 1 \dots N$) with an associated value V_{C,t_i} for criterion C, the time dependence rule determines how to combine all the V_{C,t_i} to determine the value $V_{C,timeline}$ for the whole timeline. At the moment, Nessie implements the following time-dependence rules:

- *Additive* time dependence rule successively adds all the values associated with each timeline event of the considered criterion as represented (see

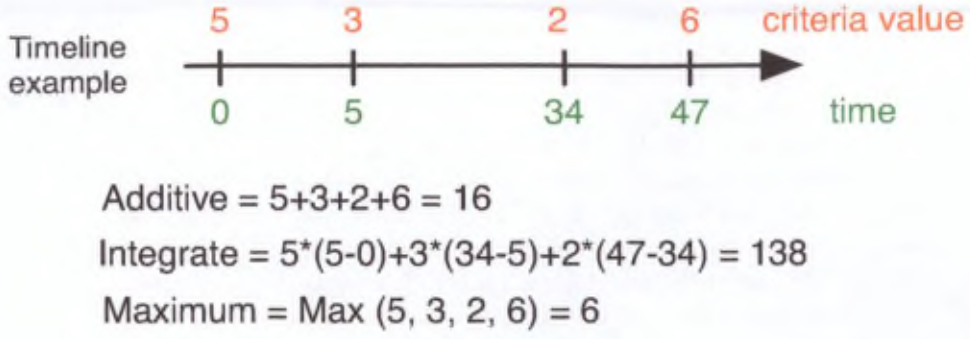

Figure 5.14: Criteria estimation for a platform structure based on time and structural information

Eq.5.2)

$$Additive\_time\_rule : V_{C,timeline} = \sum_{i=1}^{N} V_{C,t_i} \tag{5.2}$$

- *Integrate* time dependence rule adds for each event the value of the criterion before state change multiplied by the time separating the previous event from the current event (see Eq.5.3)

$$Integrate\_time\_rule : V_{C,timeline} = \sum_{i=1}^{N-1} (V_{C,t_i} * (t_{i+1} - t_i)) \tag{5.3}$$

- *Maximum* time dependence rule simply selects the biggest criterion value out of the different criterion values associated to the timeline events (see Eq.5.4)

$$Maximum\_time\_rule : V_{C,timeline} = max(V_{C,t_i}) \quad i = 1 \ldots N \tag{5.4}$$

- *Time-independent* rules are not dependent on time: the value of the criterion can be computed once and for all for the complete timeline

As an illustration, Fig.5.14 describes an example where criteria values are successively estimated for a common event timeline and an additive, integrate and max different time dependence rules.

Structural combination rule a criterion is defined by its structural dependence thanks to a *structural composition rule*: based on the platform blocks criterion value, the value of the complete platform structure is computed by integrating the criterion over the different platform blocks. In other words, the composition rule defines for a given criterion C how to combine the value

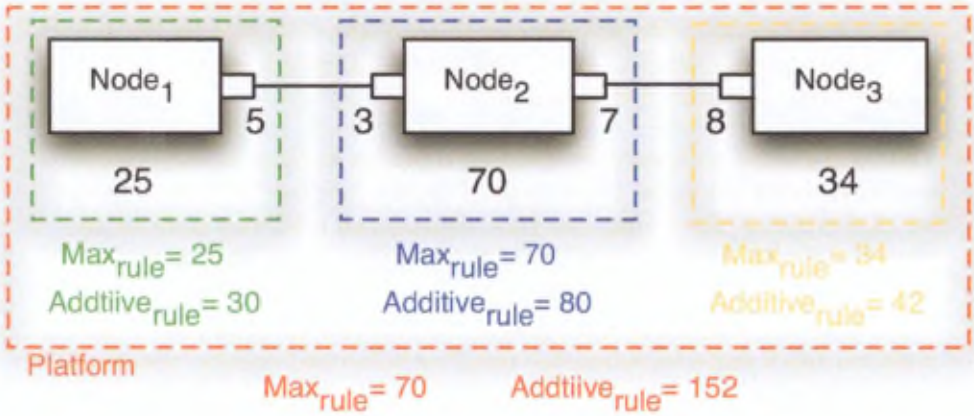

Figure 5.15: Example of the maximum and additive composition rules for a platform structure

of each of the M platform blocks V_{C,Pt_j} to evaluate the criterion value of the complete platform structure $V_{C,structure}$. Nessie defines at the moment two different rules:

- *Additive* composition rule sums the criterion value of each different platform block up as represented (see Eq.5.5)

$$Additive\_composition\_rule : V_{C,structure} = \sum_{j=1}^{M} V_{C,Pt_j} \tag{5.5}$$

- *Maximum* composition rule selects the maximum criterion value over the different platform blocks composing the structure (see Eq.5.6)

$$Maximum\_composition\_rule : V_{C,structure} = max(V_{C,Pt_j}) \tag{5.6}$$

As an example of criteria integration, Fig.5.15 shows how to combine the criteria values of the different platform blocks of a platform structure to get the global criteria values. This mechanism is done in two successive phases. First the composition rules are applied inside each platform block to the core and the ports composing the platform block: this results in the criteria values for each platform block. Second the composition rules are used to get the whole platform criteria values based on the criteria values of the individual platform blocks.
To illustrate that dependence on time and platform structure, here are a few examples of criteria and their associated time and structural combination rules:

- The silicon *area* of a chip is a parameter that is independent on time: adding inner components will however increase the chip area by the surface of these components. Area is thus a time independent criteria with an additive structural combination rule.

- The total *energy* consumed by an architecture is the sum of the components individual energies: the structural dependence rule is thus additive for this criteria. Regarding the time dependence rule, two options are possible. First we want to estimate the energy based on the power consumption of each platform block: in this case, we use an *integrate* time dependence rule to multiply the power consumed by the duration of each platform block state to obtain the resulting consumed energy. Second if each state change event of the timeline triggers an energy defined transaction, the criterion of the platform blocks represents the energy instead of power and the time combination becomes *additive*.

- The *temperature* of the chip is an important design parameter to avoid reliability issues due to overheating by taking appropriate heat dissipation measures. If the maximum temperature is thus the dimensioning factor, its associated criterion will thus be represented by a maximum structural composition rule and a maximum time combination rule.

5.3 Functionality description

In the previous section, we have discussed the hierarchical platform structure that we adopted inside Nessie and explained why it is well suited to the evaluation of flexible multi-criteria performance metrics: in the current section, we present its functional counterpart.
Contrarily to platform structures, there are plenty of ways to describe a functionality and its execution semantics based on models of computation[16]. All of these models of computation have their strengths and weaknesses so that restricting Nessie from the beginning to one sole of these MoC's would have been too limiting for the future. However we don't want to compete against tools like Ptolemy or El Greco that have been designed for the particular purpose of implementing a large variety of MoC's. Therefore we decided to design a flexible mechanism to enable easy addition of new MoC's inside the performance core of Nessie and to implement *Petri Nets* as an example. Before describing further this particular model of computation, let us review the different features that may be required for the representation of the functionality in a performance prediction perspective.

[16] Models of computation were previously defined and briefly discussed in Sec.4.3.1.

5.3.1 Basic features of a MoC for performance exploration purpose

To select an appropriate model of computation for functional structure description, we must remember that the only purpose of Nessie is the evaluation of performance criteria based on the mapping of a functionality on a platform. Therefore only the order and time spent in each state of the different platform blocks resulting from the execution of the functionality are of interest: the data results themselves are of no importance and won't be calculated.
The chosen MoC should be able to represent all the aspects of concurrency present in a functionality i.e. :

- Sequentiality: operations need to executed one after the other to preserve the functionality
- Parallelism: operations can be performed in parallel without altering the functionality
- Data dependency: one operation waits for a data to be produced by another operation before starting its execution
- Control dependency: one operation waits for a control order to be triggered before starting its execution

Among the modeling languages that can represent the sequential and parallel nature of the functionality, we chose Petri Nets[17] that we slightly modified to fit our needs: this topic will be discussed in next section.

5.3.2 Petri Nets

Structure

Petri Nets[4] are a graphical and mathematical tool to represent discrete-event distributed systems. The constituting elements of a petri network are depicted in the upper part of Fig.5.16:

- *Tokens* are exchanged during the execution between the different places
- *Places* store tokens until they can be consumed by transitions
- *Transitions* establish conditions and execution dependency between the different places
- *Arcs* link places to transitions and transitions to places and are tagged with a weight representing a particular number of tokens.

The execution of a petri network is paced by the consumption and production of tokens exchanged between the different places. Transitions are said to be *enabled* when each place connected to it contains at least as many tokens

[17] Other MoC's could probably have been chosen to represent the functionality but we went for Petri nets because of the mathematical background they rely on but also because we had some previous experience with it.

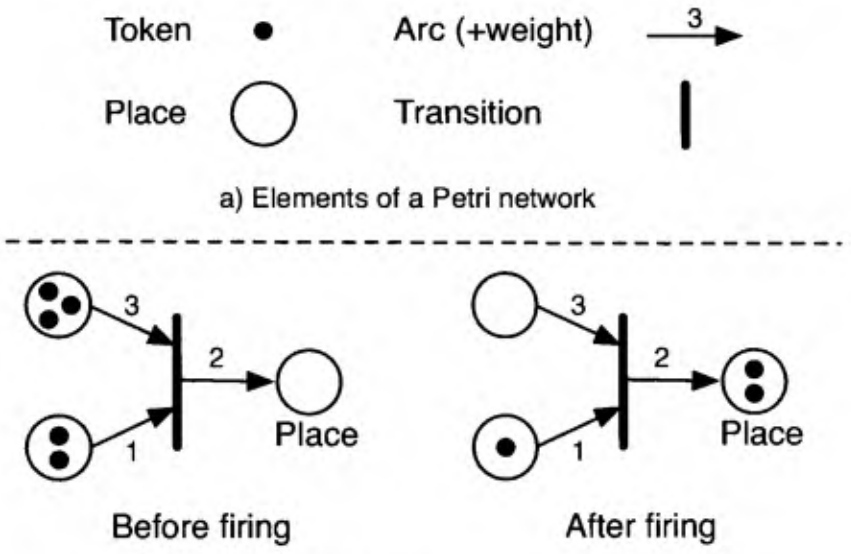

Figure 5.16: a) The different elements forming a petri network ; b) an example of petri network before and after transition firing

as the associated incoming arc weight. When enabled, the transition may fire: the tokens at the incoming arcs are removed from the places and tokens are generated in each place connected to an outgoing arc: the number of consumed/produced tokens depends on the weight of the arc connecting the place to/from the transition. This mechanism is illustrated in the lower part Fig.5.16 where a simple Network composed out of three places is depicted. Two input places are connected to the transition: the first place contains three tokens while the second place contains two tokens. Since there is a sufficient number of tokens present in both input places according to the weights of the respective arcs, the transition is enabled. After firing, three tokens are consumed in the first input place and one token in the second input place to produce two tokens at the output place.

Properties

The most important characteristics of Petri Nets as a modeling language are the following:

- *Concurrent*: both parallelism and sequentiality of operation can be easily represented in a Petri network (Fig.5.17)
- *Asynchronous*: Petri nets are asynchronous systems in the sense that there is no synchronization method to order the firing of the different

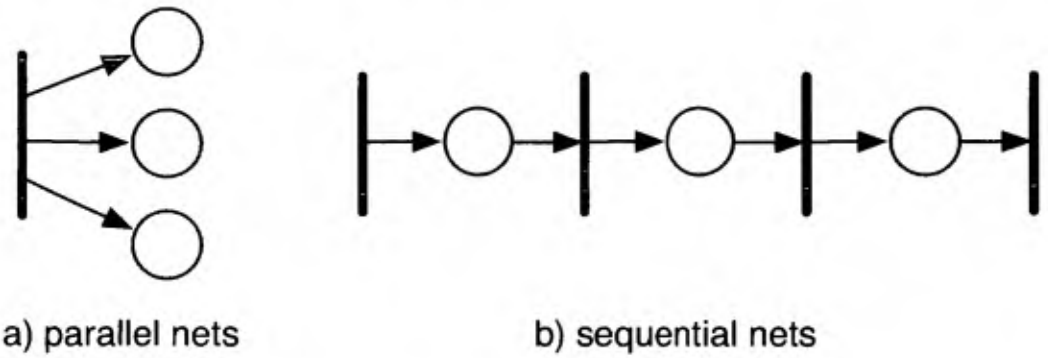

Figure 5.17: Parallelism and sequentiality of operation in Petri Nets

transitions. Only one operation takes place at a time even if several transitions are enabled so that Petri Nets give no guarantee on their triggering order contrarily to a synchronous language where all transitions would be fired simultaneously.

- *Non deterministic*: some Petri networks show some unpredictability in the order of execution of the transition and the way tokens are generated. Fig.5.18 illustrates a *conflict* where two transitions are able to consume the same token, the order of firing will thus determine the transition that will be triggered at the expense of the other.
- *Formal*: petri nets rely on a strong mathematical background allowing the theoretician to define petri places and their transitions using a matrix-based formalism and to mathematically prove several static and dynamic properties (liveness, boundedness, reachability[18], etc.) of the network.

Concurrency is a characteristic that we are looking for while asynchronism is suitable for describing discrete-event parallel computing systems. Non-determinism, particularly due to conflict, is a little more annoying and that's why we decided to modify the possible networks to avoid that kind of problems. Finally the mathematical theory related to Petri nets could be interesting in the perspective of future functional analysis and verification. However we focus at the moment in Nessie on the practical aspects of petri nets and exploit their representation potential to represent the execution of concurrent applications. From the mathematics behind petri nets, we just kept the specifications required to properly and completely define a network: a set of places, transitions, arcs, a list of weights with their associated arcs and an initial marking (the number of tokens contained inside each place of the network before it is executed).

[18]The interested reader is advised to have a look at the very complete and comprehensive [5] to have further information about those properties.

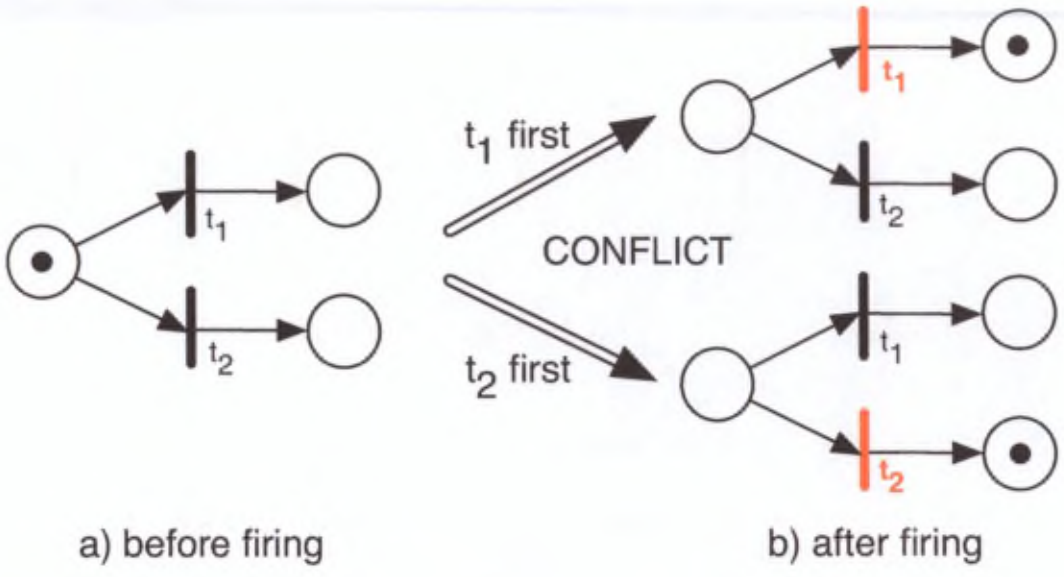

Figure 5.18: Conflict example in a petri network resulting in two possible network states after transition firing

In the next section we explain how we slightly modified petri nets to cope with the lack of predictability and data dependence representation.

5.3.3 Integration of petri nets inside Nessie

If Petri nets are very well suited to the description of concurrent distributed systems, they are not meant to explicitly represent the data exchanged between the different places. This precise point is however required to express the data dependency present in a functionality but also makes the mapping easier since platform blocks communicate by exchanging data.

Turning transitions into data/control dependency

In order to represent the concurrency together with data/control dependency of a functionality inside petri nets, we attached semantical aspects to the different constituting elements of the network:

- *Places* represent the different operations to execute inside the functionality. Each time a place is asked to generate a token, the operation linked to this place begins: the place effectively generates a token after a time corresponding to the *execution time* of the operation.
- A *transition* implies that the execution of the places pointed by the outgo-

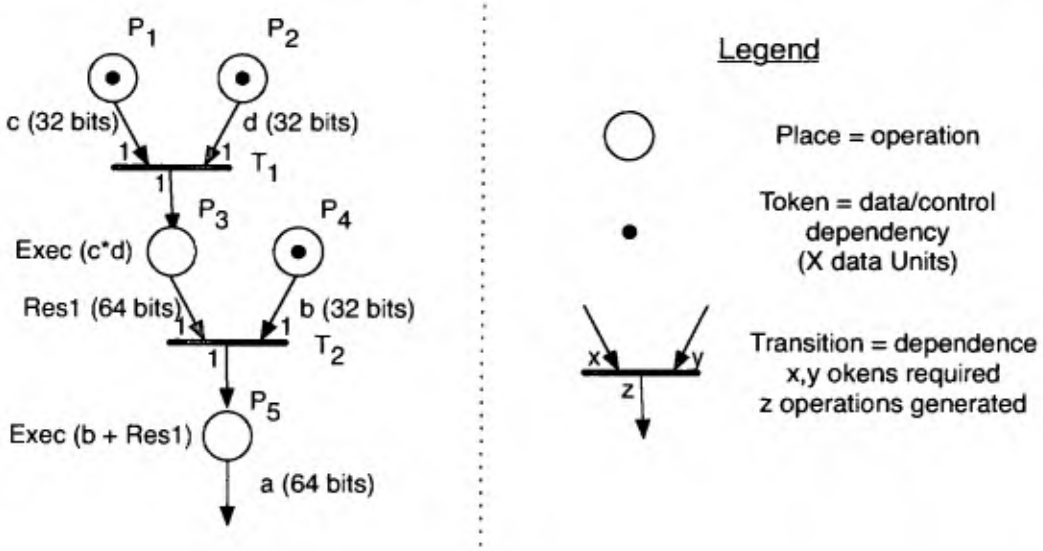

Figure 5.19: Transformation of the operation $a = b + c * d$ into a petri network

ing arcs is dependent on the execution of the places linked to the incoming edges (data or control dependency).

- *Tokens* represent the functional counterpart of control and data tokens exchanged through the different communicating platform blocks. Each time an operation ends, a token is generated in the corresponding place while its equivalent data token is generated by a platform block as the result of the operation. There is a one-to-one correspondence between data tokens and tokens used inside petri nets[19].
- *Arcs* have a different meaning depending on their orientation. Incoming arcs (from a place to the transition) represent the data/control dependency determining (according to their weights) how many petri tokens of each place are required for the transition to be fired. Outgoing arcs weights (from a transition to a place) represent the number of petri tokens that will be generated in each connected place: in other words there will be as many operations released by the transition as tokens generated by the sum of all outgoing arc weights.

As an example, Fig.5.19 depicts a Petri network representing the different operations composing the functional structure of $a = b + c * d$. This network shows the initial marking of the three petri tokens corresponding to b, c and

[19]To avoid any ambiguity, we will clearly refer to *data tokens* when referring to data exchanged in a platform structure and talk about *petri tokens* for the tokens exchanged between the places of a petri network.

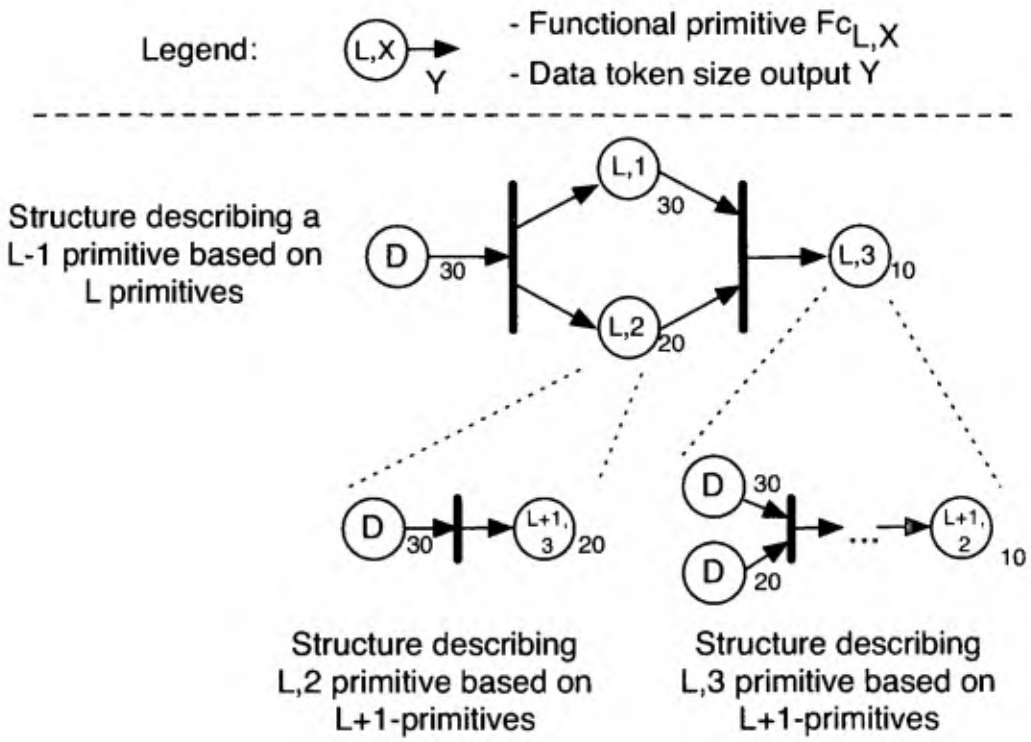

Figure 5.20: Hierarchical building of a structure based on petri nets

d in places P_4, P_1 and P_2: this means that these three data are available for computation and can be used to fire enabled transitions. If we look a bit closer at transition T_1, we can see two 1-weighted arcs linking places P_1 and P_2 and one 1-weighted outgoing edge linking place P_3: this construction implies that one single operation related to place 3 will be triggered if one token is present in place P_1 and one token in place P_2. The transition will consume both tokens (32 bits data) to produce a token in place P_3 corresponding to the result of the $c * d$ operation (64 bits data). Once this new token has been generated, the result of $c * d$ can be added to b so that the next operation is now allowed to take place: this shows how data dependency is expressed through this data communication mechanism. From the network it may also be noticed that only incoming arcs are tagged with data size values: the operation triggered by a transition implicitly consumes all the tokens flowing through the incoming arcs as inputs so that only the size of the token result need to be specified.

Hierarchical functional structure

As for the platform, functionality can be defined hierarchically by using recursive structures through the different abstraction levels. In Fig.5.20, we illustrate how this hierarchical structure building can be achieved with petri nets. In this figure we can see one structure representing a $L-1$ functional primitive based on functional blocks deriving from primitives of abstraction level L, themselves described by structures based on primitives of abstraction $L+1$. These structures are based on petri networks where each place refers to a particular functional type $Fc_{L,X}$ that can itself be defined by another petri network. Furthermore each place is associated with a data token information representing the size of the data resulting from the operation related to the functional type linked to their place. This has a major impact on structure consistency: indeed each petri network representing the structure of a functional primitive must comply with the number and size of the inputs and outputs defined by this primitive. This is precisely the case for the place derived from $Fc_{L,2}$ that receives at the input a 30 data size token and produces in return a 20 data size token. The petri network describing this $Fc_{L,2}$ primitive thus has the same number and size of tokens at the input and the output.
We can notice that the first node of this structure has no type and is instead tagged as a D (dummy) place: this particular type of place has been added to petri nets to store tokens and does not represent any operation contrarily to other places deriving from functional primitives. The consistency between abstraction levels is left to the care of the user and Nessie doesn't perform any checking of this type at the moment. Initial token marking can be performed by filling dummy nodes with tokens to determine the initial state of the petri network.
If each petri place can be hierarchically described by another structure, it is not limited to other petri networks: any functional structure could be used as long as it respects the consistency constraints in terms of input/ouput data size. Using this mechanism it could be possible to mix different models of computation in the same hierarchical performance estimation of a system allowing the user to choose the best suited MoC for each functional structure. From the mapping point of view, we however need a common interface to communicate with the structure whatever the nature of the model of computation it relies on[20].

Dummy nodes

As we have seen in Sec.5.3.2, petri nets are a non-determinisitic model of computation due to possible conflicts in the network illustrated in the previous figure 5.18. To avoid this problem, we restricted the building of petri nets to places with one single output arc: the token generated by a place thus has one

[20]This interface problematic will be further discussed in Sec.5.4.3.

and one possible output which avoids conflicts.
However by restricting places to one output arc, we have no possible way to send a token to more than one output place which limits in turn the data dependency between operations. To illustrate that issue, let us take the example of a sequence of three operations given by Eq.5.7. As we can see the first operation computes the value of a which is then consumed by two other operations whose execution thus depends on the completion of the first operation. Translated into petri networks (see Fig.5.21), we can express this dependence by duplicating the token corresponding to the result of a using the transition T_2 and feed it back to two different intermediate places P_5 and P_6 linked to the transitions T_3 and T_4 that will trigger the two operations P_8 and P_9 dependent on this token. These two intermediate places P_5 and P_6 are called *dummy* places because they don't represent any operation contrarily to regular places and are just used as filler elements inside petri networks to enable data dependence on more than one subsequent place.

$$\begin{aligned} a &= b + c; \\ d &= 2 * a + g; \\ e &= 3 * a + f; \end{aligned} \tag{5.7}$$

Dummy nodes have the following characteristics:

- The token at the input is instantaneously transmitted at the output of the dummy place
- No associated operation: since dummy nodes represent no operation, they are not associated with a platform block so that they don't add any cost in terms of performance criteria to the complete platform.

Aside from resolving data dependence issues, dummy nodes can be used to establish the initial token marking of a network as previously explained.

5.3.4 Summary

In this section we have seen how Nessie hierarchically defines functional structures without restricting the choice of the model of computation. To provide Nessie with an adequate functional representation in a performance prediction perspective, we have established the needs in terms of concurrency, parallelism and data/control dependency representation capabilities. As an example of model of computation, we described petri nets and how they are able to represent the above aspects of a functionality. We implemented petri nets in Nessie and attached some semantical aspects to represent the data/control dependency more accurately to make further mapping of the functionality on the platform easier.

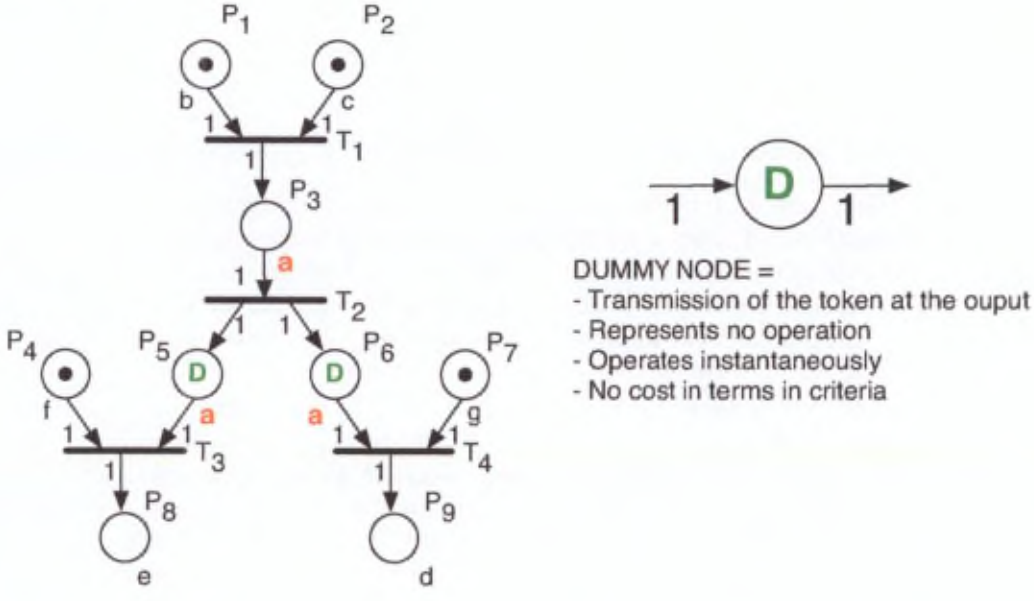

Figure 5.21: Introduction of dummy nodes for the transformation of the sequence of operations 5.7 into petri nets

5.4 Mapping

So far we have defined how Nessie hierarchically described the functionality and the platform using structures respectively based on petri nets and component netlists. In a demand/offer fashion, the functionality defines a sequence of operations that fulfills the functional requirements while the platform enables the execution of these operations at the expense of a certain cost (the performance criteria). Thus it's not only the functionality or the platform that defines these costs but the way we perform the adequacy between them: this will determine the activation order and time spent in each state for the different platform blocks, hence the performance criteria. This operation that we call *mapping* deserves a lot of attention in Nessie for two reasons:

1. The mapping is performed according to a given *policy* that determines the performance criteria to be optimized (time, area, power, etc.) which can thus lead to very different results for the same functionality and platform.
2. Nessie should be able to perform the adequacy between any type of functionality and platform: this flexibility constraint therefore puts a lot of stress on the mapping process.

This section is divided as follows: Sec.5.4.1 defines the problem and explains on a concrete example how the functional and platform structure interact dur-

ing execution while Sec.5.4.2 reviews some of the common techniques used to proceed to mapping. In Sec.5.4.3, we present the general methodology used to enable the mapping and its different steps.

5.4.1 Introduction of the problem

The core of the mapping problem is the ability to define the functionality/-platform interaction and extract their performance criteria whatever their respective level of parallelism and topology. In the previous sections, we detailed functional and platform structures but in order to further understand and define the mapping possibilities, we need to have a closer look at the possible interaction between platform and functionality. To this end, we will comment a simple example representing one of the possible mapping of a petri network on a platform structure and its execution over time. Fig.5.22 depicts a simple petri network composed out of three different places: P_1 deriving from functional primitive ($FC_{L,1}$) and P_2 ($FC_{L,2}$) represent parallel operations while P_3 ($FC_{L,3}$) requires the latter execution to be completed in order to start. The platform is based on a simple shared bus connecting three different computational blocks with different compatibility lists offering the choice to determine which functional block can run on which platform block.
The complete process of execution of the functionality on the platform goes through different steps which are the following[21] (see Fig.5.22 and Fig.5.23):

a The initial situation shows, according to the transition t_1 of the petri network, that the operations $I(FC_{L,1})$ and $I(FC_{L,2})$ are ready for execution. The two platform blocks $I(Pt_{L,0})$ and $I(Pt_{L,1})$ are compatible with functional block $I(FC_{L,1})$ while $I(FC_{L,2})$ can be executed by both platform blocks $I(Pt_{L,0})$ and $I(Pt_{L,2})$. In our present case, $I(FC_{L,1})$ begins its execution on block $I(Pt_{L,1})$ and $I(FC_{L,2})$ on block $I(Pt_{L,0})$.

b Execution of $I(FC_{L,1})$ ends (in this case we suppose that $I(FC_{L,1})$ executes faster on $I(Pt_{L,1})$ than $I(FC_{L,2})$ does on $I(Pt_{L,0})$). At the functional point of view, the place P_1 produces a petri token to indicate that execution is over and checks for the dependency of the transition t_2 linked to this place. Since the incoming arc linking P_2 has a weight of 1 and P_2 currently contains no petri token, the transition will not enabled until $I(FC_{L,2})$ ends. At the platform point of view, the end of the execution results in the production of the data token D_1 whose size is determined by the related functional primitive $FC_{L,1}$ (600 bits in this case). This token is then stored inside platform block $I(Pt_{L,1})$ for further use.

c Execution of $I(FC_{L,2})$ ends. As in the previous case, a petri token is generated in the place P_2 corresponding to the operation $I(FC_{L,2})$ and a data token D_2 (100 bits) is generated and stored inside platform block

[21] We insist on the fact that each mapping decision (allocation, order of execution, etc.) of this example have been chosen arbitrarily for the sake of simplicity.

$I(Pt_{L,0})$. Now that one token is present in both places P_1 and P_2, transition t_2 is enabled.

d Transition t_2 is fired consuming one token in both places P_1 and P_2 becoming empty. Since there is only one single arc outcoming from transition t_2 with a weight of *one* pointing to place P_3, it means that *one* instance of functional block $Fc_{L,3}$ linked to place P_3 is generated. The transition t_2 implies a data/control dependency from all the outcoming arcs linked to functional blocks with the incoming arcs linked to functional blocks: $I(Fc_{L,3})$ thus needs to consume data tokens D_1 and D_2 before starting its execution. The functional block $I(Fc_{L,3})$ can only be executed on platform block $I(Pt_{L,2})$ and is thus assigned to it. To transfer the data tokens to the platform block $I(Pt_{L,2})$, only the shared bus ($I(Pt_{L,3})$) can be used: the data token D_2 is sent first from $I(Pt_{L,0})$ to $I(Pt_{L,2})$ where it is stored.

e Once the data token D_2 transmission is over, data token D_1 can be sent from $I(Pt_{L,1})$ to $I(Pt_{L,2})$ where it is stored.

f Once data tokens D_1 and D_2 have been transmitted to the $I(Pt_{L,2})$ platform block, the execution of $I(Fc_{L,3})$ may begin.

g Execution of $I(Fc_{L,3})$ ends. A petri token is produced in place P_3 and a data token D_3 (200 bits) resulting from the computation of the functional block is stored inside the platform block $I(Pt_{L,2})$.

In this example we presented a petri net-based functional structure mapped onto a platform structure and their interaction along the execution. In this example, the mapping was implicitly solved and all the possible decisions relative to mapping choices were arbitrarily made for the sake of illustration. However it allowed us to better understand the different aspects of the problem:

- *Choice of the platform block*: we have seen in our example that some functional primitives like $Fc_{L,1}$ can be executed by more than one platform primitive ($Pt_{L,0}$ and $Pt_{L,1}$). So how do we decide on which platform block to execute a given functional block if we have multiple possibilities? On which criteria should we make that choice?
- *Scheduling of the operations*: when several functional blocks are ready for execution, how should we determine their execution priority?
- *Routing of the data tokens*: in our example we had a simple shared bus linking the different computational platform blocks. In more complex cases, how do we establish the route between a platform block producer and consumer? Do we send a token to different consumers requiring it in different transmissions or do we try to broadcast it?
- *Memorizing resulting data tokens*: in our example we supposed that the platform blocks all have the ability to store data tokens once they are produced and until they are sent to all the platform blocks consuming it.

But what about the blocks that have memorization capabilities: do we send the produced tokens to a memory or directly to a block consuming it?

The answer to these questions will determine the way that the mapping proceeds and is the main topic of this chapter. In the next section, we first investigate to find out if existing mapping techniques can fit our needs and be reused in Nessie instead of developing them from scratch.

5.4.2 Existing mapping methods: high-level synthesis

The problem of mapping as we defined it -the adequacy between a functionality and a platform at any abstraction level- is usually not addressed in such a general manner in the literature. The closest problematic is the *High Level Synthesis*[6][7] that tries to convert a high-level description of a system behaviour (C, VHDL, state diagrams, etc.) into a netlist of components executing the required functionality. Additionally a component library gathering all the computational and communication resources is defined along with their associated operations and area occupation. The optimization process driving the high-level synthesis is usually based on area and time constraints while some authors extended it to power consumption[8].

Before the high-level synthesis itself can take place, the functional description is often separated into two different parts: the control related part and the data related part. The control-less parts of the functionality are turned into data flow graphs (DFG) that will be used as input for the rest of the high-level synthesis process. In our present case, we will mostly discuss the data related aspect since Nessie focuses on the data dependency.
The high-level synthesis problem is often divided in two successive parts:

1 The *scheduling* consists in defining the sequence of operations over time with respect to their data/control dependency.

2 The *allocation* consists in associating each operation with a computation block that will execute it.

Since the allocation efficiency highly depends on the scheduling, both steps are usually performed together: we will now briefly discuss both operations.

Scheduling

Scheduling usually starts from a data flow graph and tries to find a possible order for the DFG operations execution to take place. The most common and simple algorithms used to perform this scheduling are the $ASAP$ (As Soon As Possible) and $ALAP$ (As Late As Possible) algorithms[9]. In a basic version, they make a few assumptions:

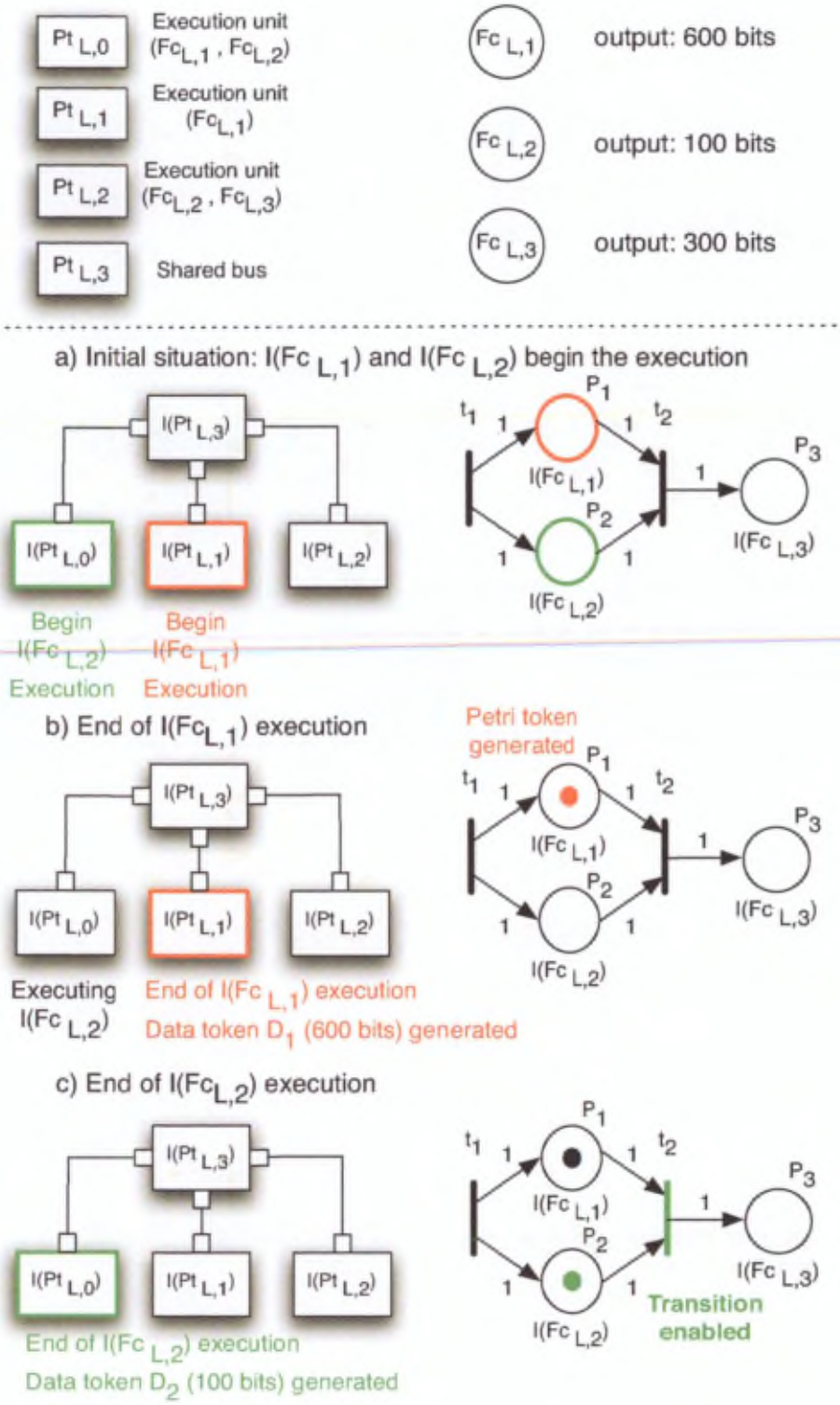

Figure 5.22: Example of mapping and execution of a petri net based functional structure on a platform structure

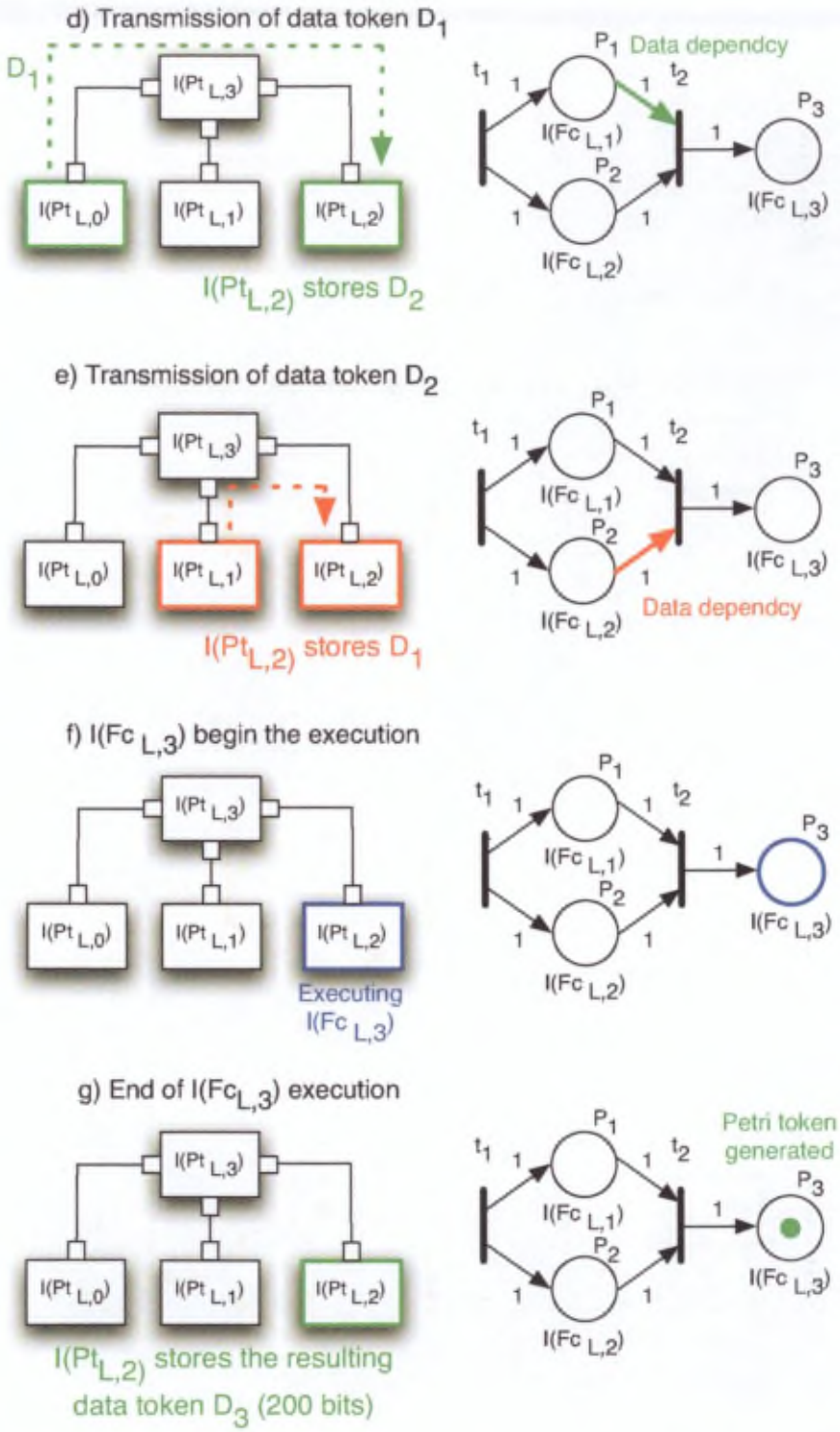

Figure 5.23: Example of mapping and execution of a petri net based functional structure on a platform structure (suite)

- The execution time of each operation is supposed to be the same. This assumption may be relaxed in more advanced versions of the algorithms
- Each operation can be executed by only one resource and each resource is uni-functional.

The ASAP (ALAP) algorithm will schedule each operation of the DFG as soon (late) as possible with respect to the data dependency. In such a method we suppose that we have an infinite pool of resources so that the execution of an operation is never limited by the availability of a resource. Fig.5.24 illustrates the ASAP/ALAP scheduling for an example[22] of a sequence of operations described in Listing 3: the result of the ASAP and ALAP algorithms show two different possible schedulings respectively in part a) and b) of the figure. From the scheduled graph, we can easily calculate the number of required resources by determining for each different operation the maximum number of simultaneous operations to perform in any control step. This gives us four multipliers, one comparator, one adder and one subtracter for the ASAP policy while the ALAP requires one less multiplier tan the previous solution.

Algorithm 3 Example of a code used as input for ASAP/ALAP scheduling algorithms

```
1: while x < a do
2:   x1 = x + dx;
3:   x1 = u − (3 ∗ x ∗ u ∗ dx) − (3 ∗ y ∗ dx);
4:   y1 = y + (u ∗ dx);
5:   x = x1;
6:   y = y1;
7:   z = z1;
8: end while
```

If ASAP and ALAP produce a latency minimum operation schedule (for a fixed DFG i.e. with no tree-height reduction[23] allowed), they require a large number of resources: other compromises in terms of resource area and execution latency are however possible. Other algorithms have therefore been developed that can be classified in two different categories according to their policies:

- *Resource constraint scheduling* consists in minimizing the latency given a constraint on the total number of resources
- *Latency constraint scheduling* consists in minimizing the number of resources (or total resource area) given a certain latency constraint

[22] This example is taken from [9] where more information can be found about the ASAP/ALAP algorithms.

[23] Tree-height reduction is a method used to change the expression of a DFG by using the associativity of the mathematical operations performed in it[10]. Using this technique it becomes possible to reduce the number of control steps required in the DFG scheduling depending on the mathematical expression represented by the initial DFG.

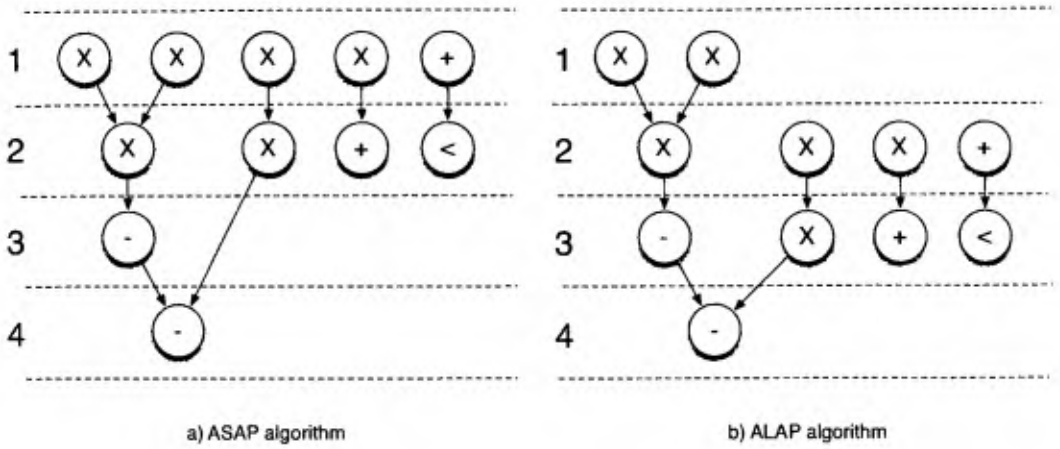

Figure 5.24: Example of a DFG scheduling resulting from an ASAP/ALAP algorithm for the operations presented in Alg.3

Most of the algorithms can be declined to answer both problems like Integer-linear programming[11], Hu's heuristic algorithm[12] or list-based scheduling[13]. Some of these algorithms also rely on ASAP and ALAP algorithms results to determine the mobility[24]. Aside from the most common algorithms, we can also mention simulated annealing, path-based scheduling and DFG restructuring[9].

Allocation

The allocation phase consists in converting the previously scheduled operations into a register transfer level (RT) structure and is divided into two different parts:

- *Unit selection* determines the number and type of RTL components to be used
- *Unit binding* associates each individual operation with an RTL component (computational, interconnect or memory) according to the scheduling

Different methods can be used to perform the allocation, however the basic idea is always to *generate* a RTL netlist based on the scheduling while Nessie tries to evaluate the adequacy between a functionality and a *predefined* platform: allocation is thus of no use in our problem. The interested reader is advised to

[24] The mobility of an operation is measured by the difference between the control step where this operation is scheduled in the ALAP and the control step in the ASAP policy. It represents the criticality of an operation: a low mobility means that differing its execution will probably increase the latency compared to the delay optimal ASAP/ALAP scheduling solution. Operations situated on the critical path all have a zero-mobility measure.

have a look at [9] for complementary information about the allocation related algorithms.

High-level synthesis application within Nessie

As we just saw in the previous section, allocation cannot be applied to Nessie but the question still remains for scheduling. Therefore let us consider the different assumptions usually made in High-Level synthesis scheduling algorithms:

- Many algorithms suppose that each operation will require the same execution time because such an assumption makes the scheduling easier
- The component library is usually defined in a way that each operation can usually be executed by only one resource type. Furthermore, multi-functional units (like adder/suntracter) are not always accepted by scheduling algorithms
- Algorithms are based on the optimization of the area/latency compromise (area- or latency- constrained scheduling) and do not often take other performance criteria into account (like power consumption or energy).
- Algorithms usually try to meet a given constraint so that they are quite computation intensive: some of them are even based on heuristics methods.

All these considerations about high-level scheduling methods don't match Nessie's philosophy of separating the functionality from the platform description to examine how their different combinations perform. Furthermore, our needs for generality in the mapping problem of Nessie are not compatible with and the methodologies proposed by high-level synthesis algorithms: therefore we will have to develop new mapping methods to suit our needs.

5.4.3 Mapping in Nessie

Establishing the requirements

The main requirements for the mapping inside Nessie are the following:

- A mapping policy easy to customize and able to express different compromises based on the performance criteria (not limited to area and latency as in the case of the previously described scheduling methods)
- A method appropriate for the performance estimation and the exploration of large design spaces. It may be less optimal than usual methods used in practice but should give a good idea of the performances resulting from the adequacy of a particular platform and functional structure and be able to rank solutions in terms of quality even if the absolute performance numbers are not fully accurate. Furthermore estimation speed remains

highly important since it determines the number of solutions (hence the size of the design space) that can be explored in a given amount of time.

- The generality of the mapping process should be preserved since the main problematic addressed by Nessie is the performances comparison of systems characterized by high functional/platform heterogeneities.

General approach

To fulfill the requirements established for the mapping, we have chosen an online mapping policy based on a discrete-event co-simulation of the functional and platform structure. The method that we propose can be categorized in the online methods since the functionality is mapped step by step on the platform based on their evolution over time: the problem is solved little by little contrarily to offline methods that examine the problem in its whole to proceed. In Sec.5.4.1 we presented an example of a mapping resulting from the adequacy of a petri network and a platform structure: Fig.5.25 illustrates the approach that we can take to address such a problem in a general way. The co-simulation of the functional and platform structure execution can be represented as a loop where the functional structure is initialized and provides the platform structure with the first functional blocks to be executed. As the simulation progresses over time, these functional blocks are executed by the platform structure and stimulate, when their execution is over, the functional structure that may provide in return new functional blocks to be executed depending on the data/control dependence of the functional structure. The co-simulation goes on until all the functional blocks generated by the functional structure have been executed on the platform.
The great advantage of such a co-simuation engine is the ability to keep the platform and functional specific simulation mechanisms independent while enabling them to interact through a general interface: the platform tells the functionality which functional blocks have been executed while the functionality sends new functional blocks to be executed independently from their inner content. Furthermore this interface-based communication implies no restriction on the internal functional/platform mechanism and representation as long as they are able to respond to the requests by generating the desired answer. This is precisely how we allow different heterogeneous models of computation to be encapsulated inside the functional structure and preserve the mapping mechanism intact whatever the nature of the MoC.
Before going further, let us define briefly how the internal simulation core works for both structures types.

Petri network simulation For petri networks, the reception of a functional block *end-of-execution* event generates a petri token inside the petri place related to this functional block. The simulator checks the transition condition related to this place and triggers it if the condition is fulfilled. The petri net-

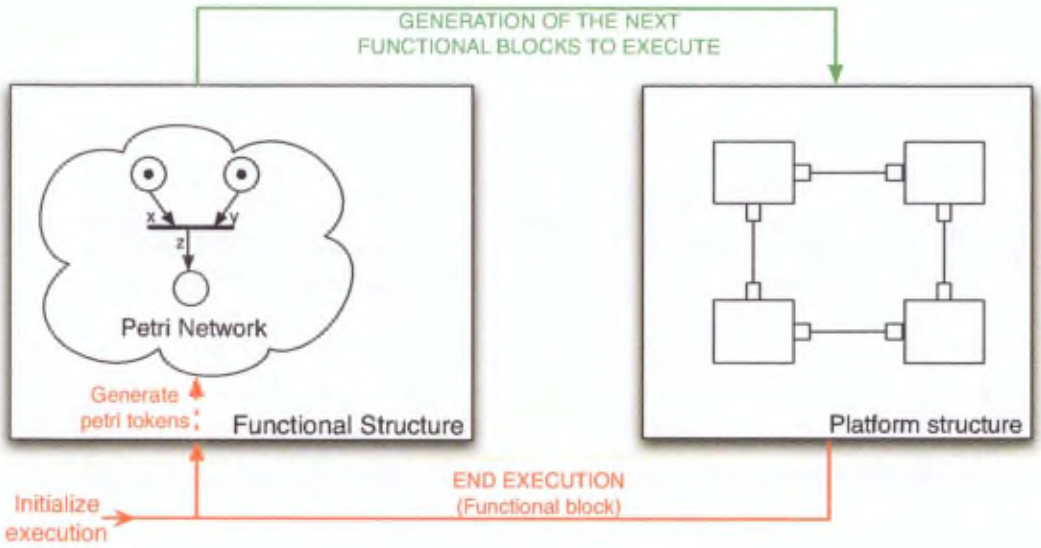

Figure 5.25: Co-simulation of the platform and functional structure

work state is updated and the next functional blocks that need to be executed according to the transition output places are generated and sent back to the platform structure. Time is not explicitly represented because it is related to the platform costs: only the data/control dependencies determined by functional aspects intervene in this simulation process. Aside from the remarks made in Sec.5.3.2, the simulation is entirely driven by the petri network execution semantics.

Platform structure simulation The platform structure simulation is more complicated than for petri networks. Not only does it embed the scheduling/allocation[25]/routing mechanisms but also is it responsible for the online computation of the performance criteria determined by the states of the different platform blocks performing the execution. We will thus spend the remaining of this section to talk about platform related aspects.

Platform simulation

Looking at the inside of the platform structure core, functional blocks issued by the functional structure go through several steps illustrated in Fig.5.26:

1 The functional blocks coming from the functional structure are first stored

[25] In Nessie, allocation is used in a totally different meaning than in the context of high-level synthesis: rather than being the generation of a RT description, the allocation within Nessie consists in assigning a functional block with a platform block where it will be executed.

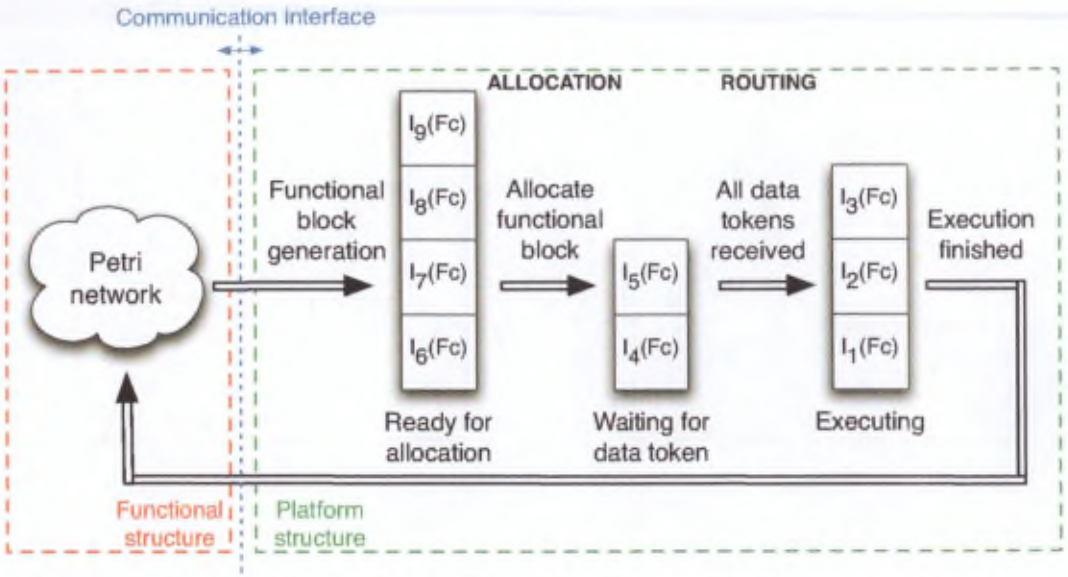

Figure 5.26: Internal simulation of the platform structure

in a *ready-for-allocation* queue. All these functional blocks waiting in this queue compete for allocation.

2 An allocation policy decides which functional block should be elected for allocation and the platform block where it should be allocated. Once a functional block is allocated on a platform block, it is moved from the *eady-for-allocation* queue to the *waiting for data tokens* queue.

3 Once all the data tokens required by a functional block have been transmitted to the platform block to which it is allocated, the execution may begin. This functional block is then transferred from the *waiting for data tokens* queue to the *executing* queue.

Once the execution of the functional block is over, it leaves the *executing* queue and a message containing this functional block is sent over to the functional structure closing the loop.

A data token oriented mechanism The execution from the platform structure point of view revolves around the creation, exchange and storage of data tokens[26] and is based on a producer/consumer paradigm. A data token is

[26] Data tokens may also represent control dependency messages exchanged between platform blocks: in that case, the amount of data linked to this token is equal to zero. It may however seem surprising that we call them *data* tokens but we used that terminology to distinguish them from *petri* tokens generated inside the petri network.

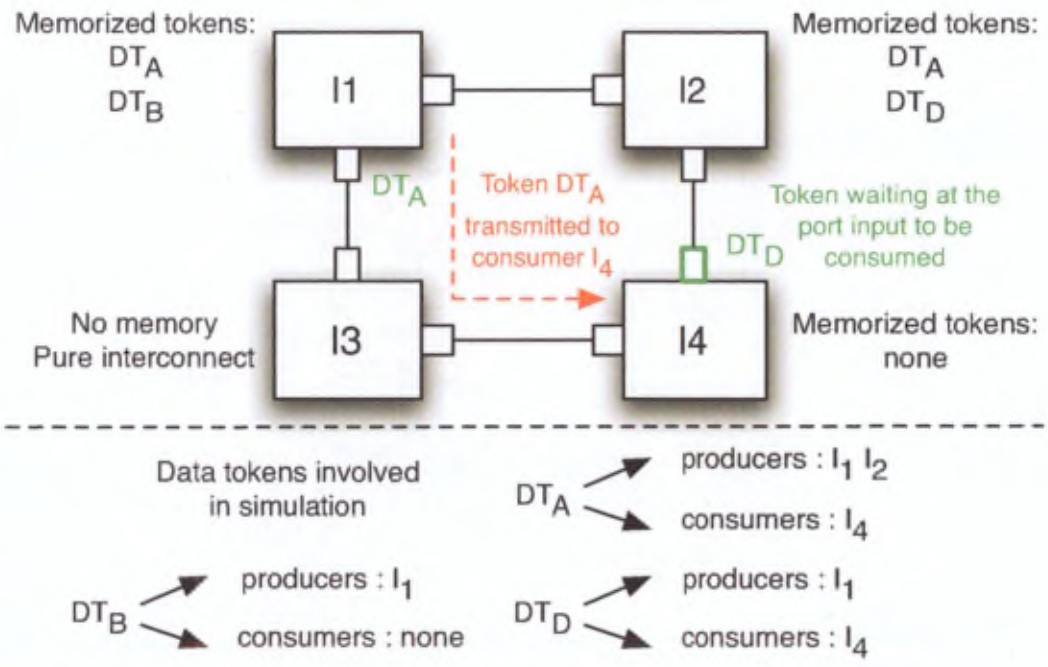

Figure 5.27: Example of data token flows between producer/consumer platform blocks

generated by a platform block once the execution of an associated functional block ends: this token thus represents the data resulting from the performed computation. After its generation, a token can be stored inside the platform block (if it has memorization capability) or remains inside the block blocking it until the token has been transmitted: in both situations, the platform block is said to be a producer for this particular token. The functional blocks that depend on this data token will be *consumers* of this particular token and by extension the platform blocks that they will be mapped onto.
To illustrate this concept, Fig.5.27 depicts a platform structure composed out of four blocks exchanging data tokens: many interesting things can be observed and illustrated on this example:

- Each platform block that has memorization capabilities has a list of all the produced tokens: that's the case of I_1, I_3 and I_4 but not for I_2 that is a pure interconnect (no computation and memorization state, see Sec.5.2.2 for more information about core states).
- A data token doesn't necessarily have to exist in one single instance: it can be duplicated as many times as required, for instance to store it in different memories that will make them available to their immediate

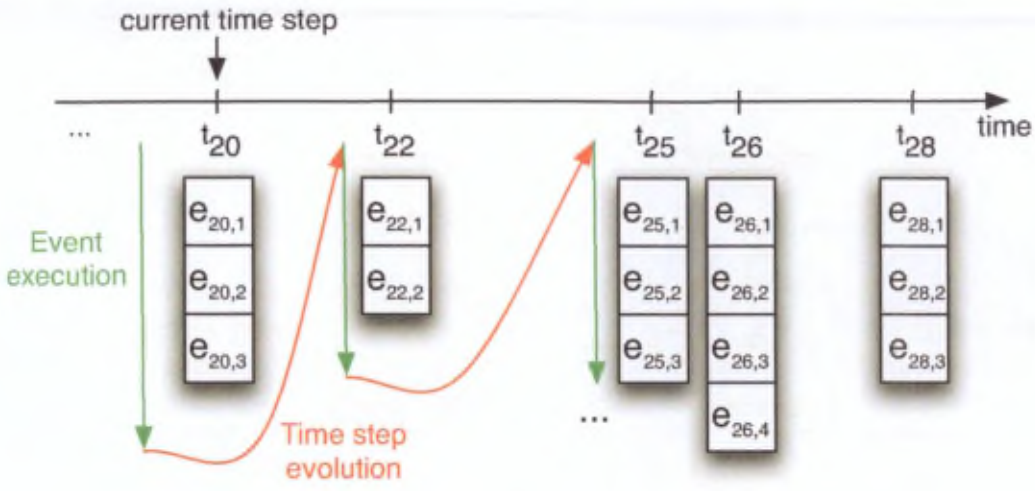

Figure 5.28: Example of timeline used for mapping process

neighbours. However all these instances refer to the same data and once all the consumers have been satisfied with their requests for this particular data token, all these data tokens instances become obsolete and can be erased.

- A data token can be sent only to direct neighbours sharing the same link or along a path where all intermediate platform blocks have interconnect capabilities. Looking at the figure, we see that I_2 has been allowed to send the data token DT_D to I_4 while I_1 is able to send DT_A to I_4 through I_3 that has interconnect capabilities.
- To manage the consumer/producer information, each token is tagged with a list of all its consumer and producer platform blocks: this will make further data token routing much easier (bottom of Fig.5.27).

Event-based mapping From the platform point of view, the mapping process is based on the evolution over time of the platform blocks state and their associated data tokens. To manage the dynamic nature of the platform structure, Nessie establishes a mechanism based on events driving the evolution of the mapping process. Each decision taken by the mapping policy generates events that are meant to be triggered at a defined time step so that the simulation evolves from time step to time step. To implement this mechanism, Nessie relies on a discrete timeline containing all the scheduled events as depicted in Fig.5.28.
A timeline is a representation of all the past and future events ordered by triggering time: for each time step, a vector contains all the associated events.

The simulation goes on from one time step to the next making the simulation progress in time. At each time step, events are triggered one after another in an FCFS order. When events are generated, they need of course to be scheduled at the current time step or in the future since it makes no sense to plan an event for the past. If the time step associated to the new event doesn't exist, it is created: otherwise the event is added at the end of the corresponding vector. The simulation thus progresses by triggering events for each successive time step until they are no more.
Four types of events are currently defined and can be generated during the mapping process[27]:

- *State change events* trigger the core or port state change of a platform block. The event contains the previous and new state and an information concerning the platform part related to this state change (the core or a particular port). When switching to another state, it means that the performance criteria of this block might change: the performance criteria values of this platform block are updated based on the current time step value and the previous time step value of the mapping as explained previously in Sec.5.2.3.

  Triggering this event will additionally check for the end of execution of a platform block that occurs when a core changes his state from *computing* to *idle*. In such a case, the data token is generated and handled according to the data memorization policy while the functional structure is told that a functional block has ended its execution and may generate in return new functional blocks that are stored inside the ready-to-allocate vector (see Fig.5.26).
- *Platform release event* is related to a particular platform block to check if it can be released for the execution of another functional block. Indeed it can be possible that this platform block may not be available for execution because it is in blocking mode waiting for a data token to be consumed by another platform block.
- *Data token reception event* notifies a particular platform block that a data token it was waiting for has just arrived. If all the data tokens required by the functional block assigned to the platform block are either stored inside this block core or present at one of its input port, the execution may begin.
- *Data token memorization event* notifies a platform block that a data token is now stored inside its memory and that it is now able to use it or transmit it to another block if required.

Alg.4 describes in details the sequence of operations driving this event-based mapping:

[27]The number of event types is not restricted and the implementation easily allows the programmer to add events and integrate them inside the mapping process (see Sec.5.6.5).

1 The functional structure is initialized and generates some functional blocks to be executed

2 At each time step, events are popped out of the vector and then triggered until there are no event left. Then the mapping tries to schedule and allocate the functional blocks available for execution. Events that might have been generated from the previous allocation and schedule phase for the current time step are then triggered. After that, the data tokens are routed from the available producer to the consumer platform blocks and the remaining events for the current time step are once triggered. This process is iterated as many times as there are time steps in the timeline.

3 At the end of the mapping, the performance criteria that are independent from time are estimated: the one dependent on time have been progressively estimated during the different events.

Algorithm 4 General algorithm for the event-driven mapping method inside Nessie

```
Get the first functional blocks from the petri network initialization;
while No time step remains in the timeline do
    Go to next time step;
    Trigger all the remaining events;
    Schedule and allocate functional blocks;
    Trigger all the remaining events;
    Route data tokens;
    Trigger all the remaining events;
end while
Calculate the values of the time-independent performance criteria;
```

Deadlock detection As previously explained, when platform blocks don't have memorization capabilities and produce a token as a result of an operation, they switch to blocking mode: no interconnect or computation operation can be performed until the produced token has been consumed by another platform block. This blocking mode may however be problematic since it could possibly lead to deadlocks because some routes may be unavailable due to blocked platform blocks along the path.

This deadlock problem is depicted in Fig.5.29 presenting a platform structure composed out of four different blocks without memorization. Each of them have finished their execution and have produced a data token DT_i where i corresponds to the block ID for the sake of simplicity in this example. Platform block I_4 needs to consume data token DT_1 but cannot receive it because the blocks (I_2 and I_3) on the path are currently blocked. To be released, blocks I_2 and I_3 need to deliver their produced token respectively to I_3 (impossible since I_4 is blocked) and to I_2 (impossible since I_1 is blocked). This cyclic dependence

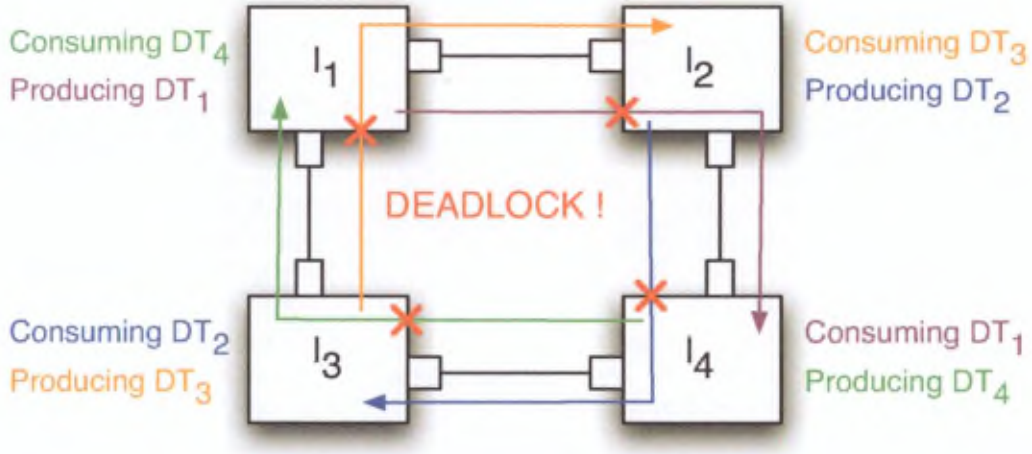

Figure 5.29: Example of deadlock due to platform blocks in blocking-mode: all blocks need to transmit a data token to the neighbour situated at their diagonal

is a deadlock and cannot be solved if no block gives up transmitting a data token to free a path for other blocks to send their data tokens.
Deadlocks result from the definition of a platform structure with too few resources (or limited data transmission capability) compared to the available parallelism of the functionality combined to the greed of the allocation mapping policy. Two solutions are possible to avoid deadlocks:

- *Limit the resource utilization:* by limiting the resource usage ratio at any time of the online mapping process to a given maximum percentage, we decrease the chances of facing a deadlock. However this percentage depends on the functional structure, the number of links between the functional blocks and their topology so that it is difficult to define a resource usage ratio for which we can guarantee the absence of deadlocks.
- *Recover from deadlocks:* when facing a deadlock, we could use a backtracking method to recover from it. Going back in the timeline of events, we could prevent the functional block creating the cyclic token dependence from being allocated and differ it until enough resources have been released.

Since the aim of Nessie is the prediction of performance of a functionality/platform/mapping combination and not optimization, no deadlock removal policy has been implemented to avoid some solutions from taking too long evaluation times due to computational intensive backtracking methods. However we must make sure that Nessie is able to detect a deadlock when it occurs and warn the

user about it by invalidating the particular solution. Therefore, the mapping engine checks if all functional blocks vectors (*ready-for-allocation*, *waiting for tokens*, *executing* vectors) are empty when there are no remaining events to trigger in the timeline: if this is not the case, we are in a deadlock situation.

In summary, the mapping core inside Nessie relies on the co-simualtion of the functional and the platform structure by establishing a mechanism based on events to make both structure and platform block states evolve over time. The main advantage of such a technique is that it leaves the scheduling, allocation and routing policy completely independent of the simulation core so that we can modify and define these operations without changing anything to the general mapping mechanism. The next section describes in more details the scheduling, allocation and routing methods.

5.4.4 Scheduling/allocation

At the beginning of each time step of the co-simulation, we need to test which functional blocks out of the ready-for-execution ones can be executed and on which platform block: the choice of the functional block and its associated platform block are precisely determined by the scheduling and the allocation[28]. Like we explained in Sec.5.4.2, the performances results of the allocation highly depends on the scheduling and that's why both operations are usually carried out simultaneously and so does Nessie.

Scheduling

Contrarily to the scheduling performed in high-level synthesis, the execution time of each operation is not fixed. Indeed usual HLS scheduling algorithms suppose that each operation can only be mapped onto one single type of platform block which entirely determines the duration of each operation. In the case of Nessie, each functional type may be simultaneously compatible with several platform types which makes the execution time dependent on the platform block choice hence on the allocation policy. The notion of mobility -at the basis of several HLS scheduling- makes no sense in our case since the lower and upper possible execution start/end time cannot be determined anymore. In such a context scheduling thus becomes more about determining the order of execution rather than determining the absolute execution control step for optimality (see Sec.5.4.2).
Though very complex scheduling policies could be implemented inside Nessie, we have chosen a simple FCFS method for the selection of the functional block in an effort to optimize the execution time. Among all the possible functional

[28] In Nessie, we define allocation as the action of associating a functional block with a platform block. This slightly differs from high-level synthesis where allocation refers to the *generation* (and not the the association) of a platform at a RT level for a pre-scheduled DFG.

blocks waiting in the ready-to-allocate queue, we will try to allocate the first one present in the queue, then the second and so on until we reach the end of the queue. When a functional block is elected, it is simply erased from the queue (whatever its position which makes it slightly different from a pure FCFS policy). Step by step, functional blocks that haven't been elected yet will slowly move on to the front of the queue which gives them a growing priority with time: this policy thus avoids the phenomenon of starvation[29]. This policy is well adapted to the scheduling of functionalities described by petri nets that are regular in terms of data/control dependency i.e. that have paths of fairly similar depth. The analysis of this dependence based on the notion of mobility generalized to multi-functional platform blocks could certainly lead to better results but such a topic is sufficiently large and complex to deserve a specific research so that we don't investigate it further in the context of this work.

Allocation

The allocation task consists in associating a functional block elected by the scheduling policy with a platform block. The selection of the platform block relies on two criteria:

1. The selected platform block need to be able to execute the functional block, this is verified using the compatibility list (see Sec.5.1.3).
2. Second the platform block that will be selected out of the compatible ones will be the most *efficient* for the particular functional block to execute. Since the notion of efficiency is relative to the concerns of the designer, it's up to Nessie's user to define how to calculate the execution efficiency of a functional/platform couple.

This allocation efficiency is represented by a weight value calculated using a Yeti model for all the competing functional/platform couples determined. This model uses all the possible user-defined performance criteria[30] as inputs: the model can be purely based on execution time, energy, area etc. or any mathematical combination of these input parameters. Based on the weight calculated by this Yeti model, all the competing platform blocks are compared and the one with the smallest weight[31] is selected for allocation. If the user has no idea how to define the selection policy, defining a model independent

[29] Starvation is a phenomenon that may appear in systems managing the attribution of shared resources based on a priority mechanism: it is a commonly encountered issue in computer science and particularly in thread management. Starvation happens when a low priority consumer of a resource gets never provided with it because of the constant existence of competitors with higher priority that always get served before. The solution to that problem always revolves around priority modification with time.

[30] The criteria always include the execution time that is the default criterion for each computing state of a platform block.

[31] The smallest weight will always be selected: if the user desires to select the maximum weight instead, he just need to calculate the inverse value of the weight by modifying the underlying model.

of any criterion and equal to a constant will make the policy degenerate to a pure FCFS policy.
Our policy that relies on a user-defined model based on performance criteria values is thus very flexible, enables the use of multi-functional platform blocks (unlike many algorithms) and is well suited for an online mapping algorithm. Clever use of the models used to compute the weights for allocation selection allows the user to represent different performance compromises with the same mechanism. The model for allocation weight calculation defines a mapping-related degree of freedom that the user is able to modify.

5.4.5 Routing

In the previous section, we have explained how scheduling/allocation works, the first operation performed at each time step of the functional/platform co-simulation. It's now time to discuss the second operation called routing.

The problem of routing

The routing process basically consists in determining the path through the communication network that data/control tokens will take to flow from the producer platform blocks to the consumers. Although it may sound quite simple, there are many problems surrounding the routing as illustrated in Fig.5.30:

- *Determining a route*: platform structure definition in Nessie is entirely left to the user so that no assumption can be made a priori on the complexity of the interconnect topology: a routing method independent from the communication network and supporting bidirectional and unidirectional links is thus required to solve our problem. Aside from route determination, we would like this method to rely on a user-based definition of the quality of a route rather than on a predetermined metrics.
- *Events determination*: if the scheduling determines the functional blocks to be executed first by allocating them on platform blocks, the exact timing relative to the activation and execution start of the different platform blocks is determined by data tokens progressing from one block to another. We thus need to find a method to generate these activation events depending on the latency and the bandwidth of the different blocks.
- *Selection of the data producer/consumer couple*: at a precise time step, several functional blocks mapped onto platform blocks may be simultaneously waiting for data tokens in the queue. Since interconnect capabilities are often limited, all requests for data token sending cannot always be satisfied: therefore we have to elect the producer/consumer pair among the competing ones for data token transmission. Should we favor platform blocks in blocking mode to release them for the execution of other op-

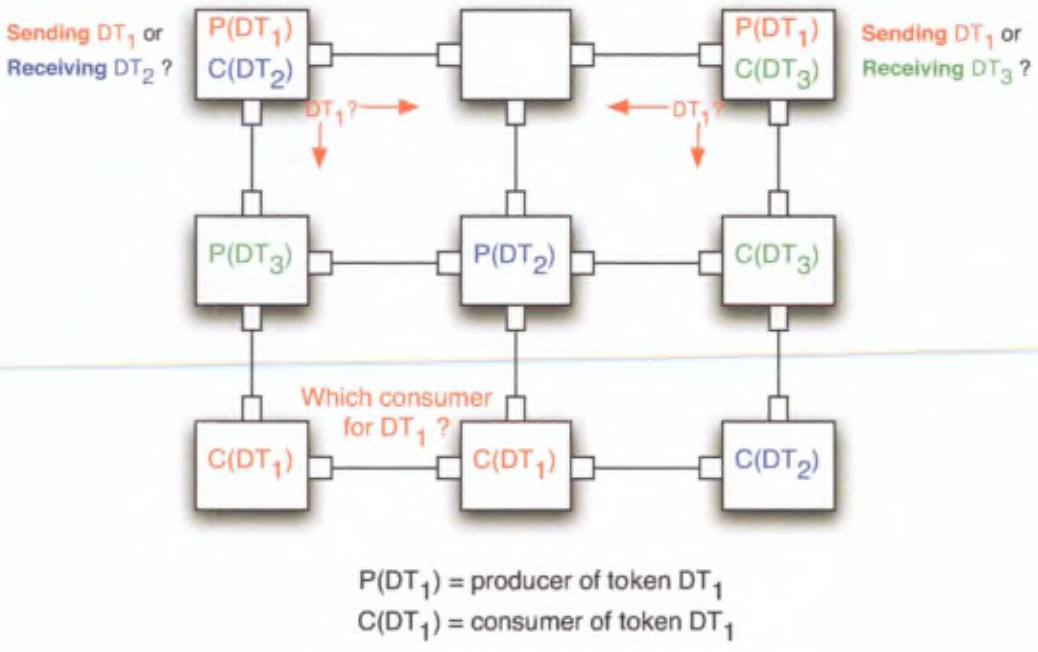

Figure 5.30: General problematic of data tokens routing illustrated on an example of platform structure. This figure shows different routing possibilities for producers/consumers of the same data token DT_1.

eration or not? In the case of multiple producers, which one should we choose? Which consumer should be served first?

- *Token broadcasting*: sometimes a same data token may be consumed by different platform blocks at the same time: when the interconnect network supports it, data token broadcast may offer more interesting compromises in terms of latency and be more friendly regarding network congestion than several point-to-point communications. The questions of defining a method to determine those multiple paths routes and evaluating these different compromises however need to be answered.

These different points will be the topics discussed in this section devoted to routing: as a starting point, we will describe a general method to determine a route between a producer and its potential consumer.

Dijkstra's routing algorithm

To enable the routing of data tokens, the very first step lies in the ability to find a route from a given source to a destination node. Since the communication architecture is totally defined by the user, we need a very flexible method to perform the routing: therefore we chosen Dijkstra's routing algorithm.
The Dijkstra's routing algorithm[14], widely used in networks, finds the shortest path from a source to any reachable node of a communication architecture represented by a graph of vertices linked by non-negative edges. Fig.5.31 illustrates such a graph with vertices representing communication nodes while the edges linking them represent the communication links with weights defining their usage cost: the higher the weight, the less interesting it is to take that path. Dijkstra's algorithm also supports unidirectional links represented by oriented edges instead of non-oriented edges for bidirectional links.
Dijkstra's routing algorithm is described in Alg.5 where V is the set of all vertices present in the input graph while U is the set of vertices left to explore. Each vertex v_i of the graph will be defined by two attributes evolving during the Dijkstra's shortest path search: the precedent vertex in the shortest path and its distance i.e. the cumulated weight of all edges from the source.
This algorithm proceeds basically in two successive phases:

1 The vertex among the unexplored list U with the smallest weight is chosen as the next vertex to explore.

2 All the neighbours around this vertex are explored: their distance and the previous vertex in the shortest path are updated if necessary.

This process goes on until all vertices have been explored (until U is empty) or if the destination vertex has been found if it was specified.
The Dijkstra's algorithm complexity grows like $O(N^2)$ where N is the number of communication nodes: it can even be reduced to $O(N * \log N)$ if some efforts are spent on the minimum distance vertex election in line 10 of Alg.5.

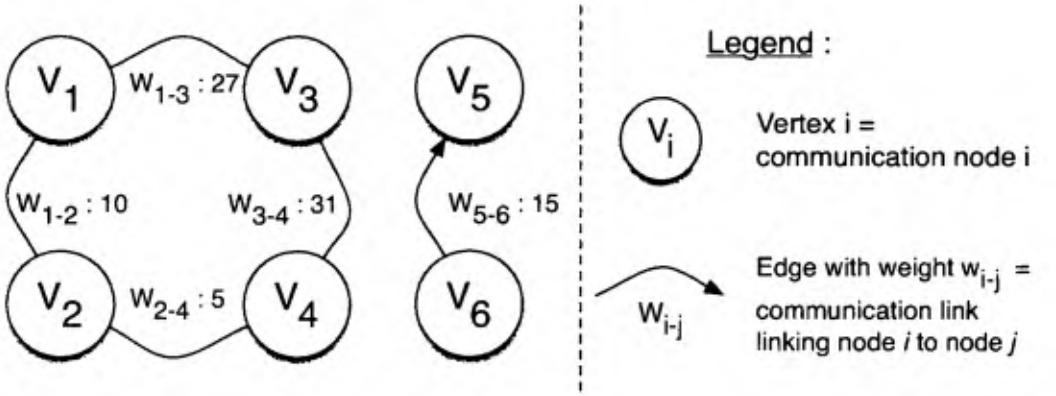

Figure 5.31: Illustration of a graph representing a communication network: vertices are communication nodes while edges represent communication links

Algorithm 5 Dijkstra's routing algorithm(V,v_{source})

1: {Initialization of all the vertices of the graph}
2: **for** All vertices v_i of V **do**
3: Set the distance of v_i to infinite
4: Set the previous vertex of v_i to undefined
5: **end for**
6: Set the source vertex v_{source} distance to zero
7: Copy vertices of V into U
8: {Beginning of the algorithm in itself}
9: **while** U is not empty **do**
10: Find vertex v_{min} with the smallest distance and remove it from U
11: **for** each vertex v_j neighbour of v_{min} **do**
12: {This step compares the current distance of each vertex surrounding v_{min} and checks for each of them if the new path has a smaller distance or not}
13: **if** $dist(v_{min}) + weight(v_{min}, v_j) < dist(v_j)$ **then**
14: Previous vertex of v_j in the shortest path becomes v_{min}
15: The distance v_j becomes equal to $dist(v_{min}) + weight(v_{min}, v_j)$
16: **end if**
17: **end for**
18: **end while**

Applying Dijkstra's algortihm to platform structures

Although the Dijkstra's algorithm is very flexible and general, platform structures make some assumptions that prevent us from directly using this routing algorithm. Indeed the input graph of the Dijkstra supposes that computation nodes are vertices and communication links are edges. Such a simple separation cannot be done for platform structure since some platform blocks may be able to perform both communication and computation operations. Furthermore there may be more than one interconnect node separating two computation blocks: this makes the use of a single edge between two vertices problematic. To solve these issues, we established two important rules to define which blocks can be crossed and how the source/destination distance will be calculated based on the weight of the individual blocks present on the path.
First, each time a platform block -whatever its nature- is crossed, its weight is added to the total distance of the path crossing it. This means that all the blocks (except the source) present in a route contribute to the total distance of the path: several successive interconnect blocks will thus have their weight added as if they were one block with a weight equal to the sum of their respective weights. Links connecting ports are virtual and don't represent any implementation cost: they won't contribute to the path cost.
Second, due to the different states (computation, memorization and communication) that a given platform may occupy at different times, we need to establish rules to determine if a block can be crossed or not depending on its actual state. Source and destination blocks will always be computation blocks producing/consuming a data token or blocks sending/receving the content of a memory. To transmit a data token from a consumer to a producer we need to find a path exclusively composed out of blocks having communication capabilities. In other words, this means from the Dijkstra's algorithm point of view that a platform block Pt_{center} will have surrounding neighbours only if:

- The directionality of the port doesn't prevent Pt_{center} from sending data tokens to the potential neighbour
- Pt_{center} has communication capability or Pt_{center} is the source block used to initialize
- The tested neighbour is in idle mode i.e. actually not busy and not reserved for a future transaction.

Given these simple rules, it becomes possible to use Dijkstra's routing algorithm on a platform structure. Left part of Fig.5.32 presents an example of platform structure composed out of platform blocks with and without communication capability (respectively with green and red backgrounds) linked by a network of logical links. Using our previous rules for the identification of valid and reachable neighbours, we can see the resulting graph that is a valid entry point for the Dijkstra's algorithm initialized with block I_1 as source (right part of Fig.5.32). As we can see in this example, one platform structure with no

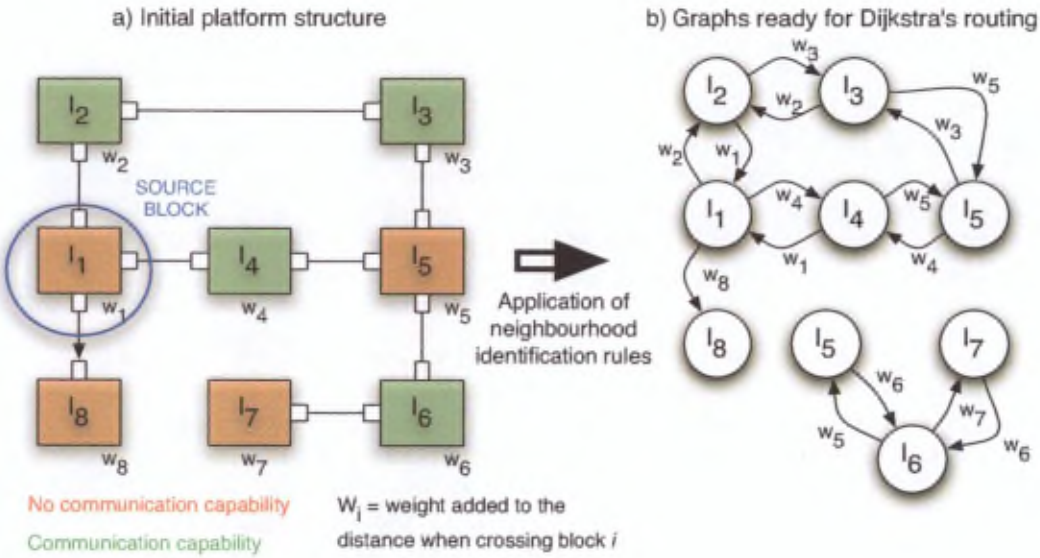

Figure 5.32: Application of the neighbourhood identification rules to a platform structure (a) in order to obtain the equivalent graph representation (b) required by the Dijkstra's routing algorithm

isolated block results in two disjoint graphs representing the nodes that can exchange data tokens with each other. This results from the fact that block I_5 splits the platform structure in two parts that cannot communicate together since I_5 has no communication capability. Visually it is very easy to identify pairs of blocks that are able to communicate: the source I_1 must be separated from the destination by a path of successive green blocks only with links directed towards the data flow direction. We may also note that since I_1 is a source block, it is able to communicate with I_8 even if there are no intermediate interconnect block in their way: this can be useful to allow the user to evaluate the performances of a system where computation nodes communicate immediately together with no interconnect overhead.
Additionally, each directed edge of the graph is annotated with a weight associated with the destination vertex: this weight will be added to the total distance when including the pointed vertex to the route. When a link is unidirectional (as it is the case for the I_1 to I_8 link), only one directed edge will be present in the resulting graph.

Dijkstra's algorithm example To illustrate Dijkstra's routing algorithm, let us take back our previous platform structure example and explain step by step using Fig.5.33 how it works when I_1 is the source node:

1 Since the source is initialized with a zero distance, we select it as the starting block. All the neighbours around the source may be explored if they are currently unused since a non-communication capable block may be the neighbour of the source block. All these neighbours have not been explored yet (infinite distance) and their distance is thus updated with the sum of the current distance of the path (0) cumulated with the weight of each neighbour: the algorithm has established three initial routes to blocks I_8, I_4 and I_2. Source block is now popped out of the unexplored block list and the block with the smallest weight is elected for the next iteration: in our case this is I_8.

2 I_1 is the sole neighbour that can be reached from the currently explored block I_8. Dijkstra's algorithm will discard this option for the smallest route election since adding I_1 weight to the current route distance of block I_8 will increase the distance of I_1 from 0 to $8 + 5$. Thanks to this mechanism, the Dijkstra can guarantee us that no route with cycles will appear as long as the assumption of strictly positive weights is respected for all blocks. I_8 is tagged as explored and I_2 becomes the next block to explore since it has the smallest distance among the remaining unexplored blocks.

3 I_2 explores its two neighbours and updates the distance of I_3 (15+6) that becomes the new unexplored block with smallest weight. The link from I_2 to I_3 is now included inside the shortest path.

4 I_3 explores its two neighbours and updates the distance of I_5 (21+7) and a route towards it is established. I_3 is tagged as explored and I_4 becomes the next block to be explored (because its distance is smaller than I_5).

5 I_4 explores its two neighbours: no change is made since both blocks I_1 and I_5 have a smaller distance than the potential new one. I_5 becomes the new block to explore.

6 I_5 has no neighbour since it has no communication capability and thus cannot transmit anything to any contiguous block. The algorithm ends here since the two blocks I_6 and I_7 that remain unexplored have an infinite distance meaning that they cannot be reached. This result could have be anticipated from the fact that there were no directed edge leading from I_1 to I_6 and I_7 in the graph represented in Fig.5.32.

The only assumption made by the Dijkstra's algorithm is the non-negativity of the weights of the input graph representing the communication architecture. In practice, this hypothesis entails that a route cannot become better whenever a block is added to it. However the non-negativity of weights assumption could be relaxed using other routing methods such as Bellamn Ford algorithm[15]. If this algorithm is able to handle negative weighted edges at the price of a longer processing time than the Dijkstra, some important measures are to be taken to avoid the creation of infinite cycles during the routing process. Such a case is

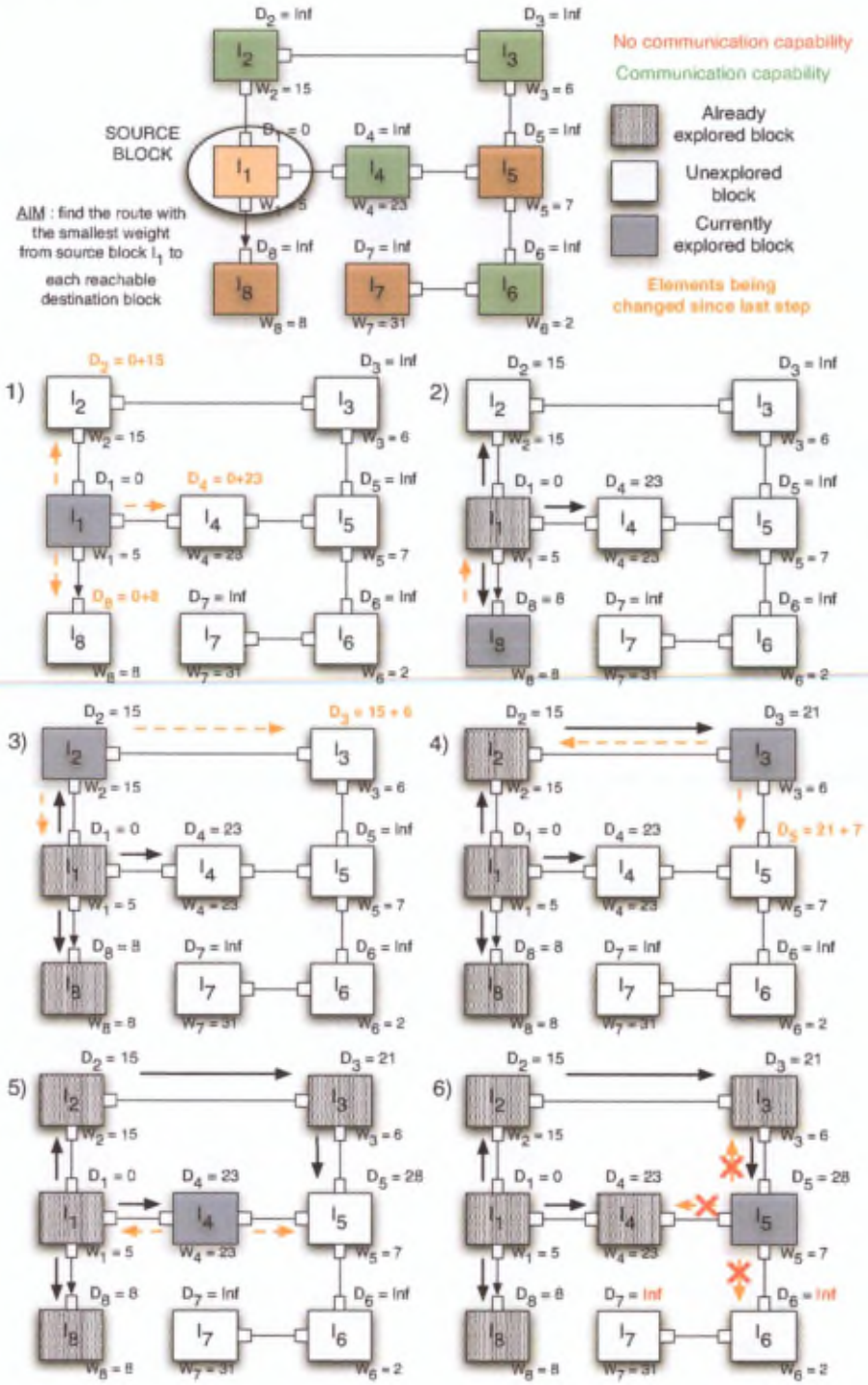

Figure 5.33: Example of the Dijkstra's routing algorithm applied on a platform structure

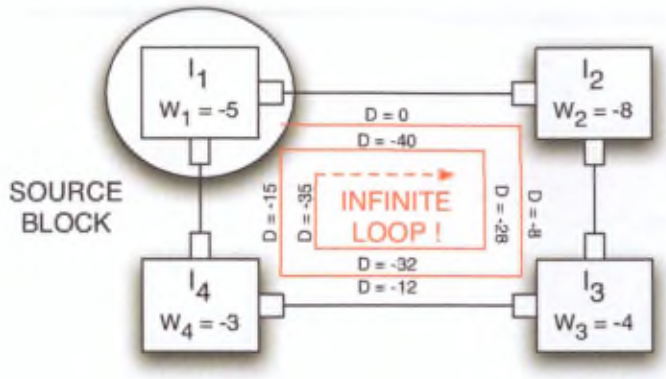

Figure 5.34: Example of a Bellman Ford routing algorithm for a platform structure resulting in infinite loop route

illustrated in Fig.5.34 for a simple 2 by 2 mesh platform structure where each block has a negative weight. The source block for the routing algorithm is set to block I_1 and the smallest weight route is progressively established through blocks I_2, I_3 and I_4. Each time a block is crossed, the route becomes less and less costly since each negative weight decreases the total distance of the route. Once the routing algorithm will get to block I_1, it will be included in the smallest route for the same reason so that we have returned to the initial block. However, crossing again and again the blocks using the same path will constantly improve the quality of the route by decreasing its total distance so that the routing process will infinitely loop on the same path which leads to a deadlock in the routing algorithm.
Several techniques exist to avoid this loop problem but for the sake of simplicity and processing time we will only use the Dijkstra's algorithm prohibiting at the same time the use of negative weights. Any attempt to use negative weights for the routing will thus result in a run-time error generated by Nessie.

Weight determination

In order to allow the user to give its own definition of the quality of a route, the weight used for the routing algorithm is user-defined as it was already the case for allocation weights. We also define a Yeti model common to all abstraction levels that can be customized and involve several parameters relative to one platform block of the network[32]:

- *latency* of a platform block corresponds to the time separating the complete reception of a data token from its complete sending

[32]The described parameters for weight determination are referred to their current name in the model: the user should respect these names when using these parameters inside the Yeti model.

- BW is the bandwidth of the platform block
- *numberOfNeighbours* is a parameter representing the number of neighbours around the platform block. This can be useful since a route passing through a block with many neighbours will be more penalizing in terms of network congestion since all these neighbour blocks won't be able to use this route anymore until this block is released.
- *numberOfCompatibleSWtypes* represents the number of functional primitives that the crossed block is compatible with: if it has no computation capability, this parameter will be equal to zero. If a route is established through a block compatible with several possible functional primitives, this block cannot be used for allocation making this route potentially less interesting than others.
- *hasInterconnectCapability*, *hasComputationCapability* and *hasMemorizationCapability* are boolean parameters whose value is equal to 1 if the block has interconnect, computation or memorization capability and 0 otherwise.

Based on these parameters, the user is able to build any mathematical combination representing different possible policies for the choice of the route without having to change the Dijkstra's routing algorithm. In networks, the most simple measure of a route distance is the hop count i.e. the number of crossed blocks whatever their characteristics. This hop count routing distance measure can be easily implemented inside Nessie by defining a weight model always equal to 1: each time a block is crossed, the weight equal to 1 will be added to the path distance so that the resulting distance will be equal to the number of crossed blocks.
It may also be mentioned that the implementation is sufficiently flexible to enable the easy addition of new parameters for the weight calculation aside from the already existing ones.

Data token broadcast

Sometimes several platform blocks require to consume the same data token at the same time so that sending it simultaneously to all the consumers could be more time efficient than sending it separately to each consumer. If the communication blocks enable it, data token broadcasting could therefore be an interesting option and we thus included that functionality inside Nessie.
Based on the Dijkstra's algorithm, it is fairly easy to find the best route from a producer $P(DT_1)$ to multiple consumers $C_i(DT_1)$ for the transmission of a given data token DT_1: this situation is depicted in Fig.5.35. The different steps required to get the route are the following:

- Select the producer for the desired token as the source block of the Dijkstra's routing algorithm
- Perform the Dijkstra's routing algorithm

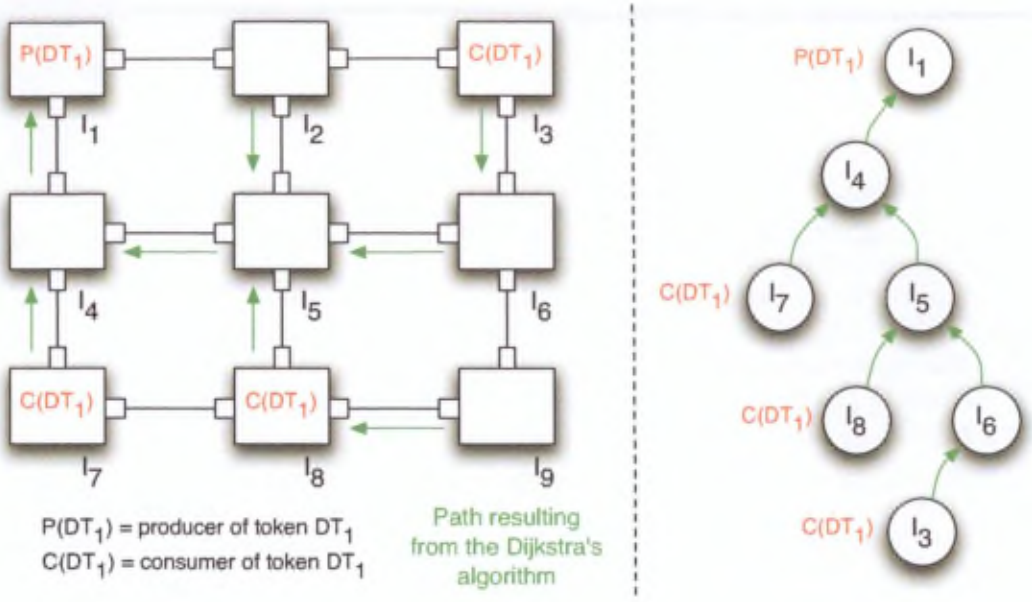

Figure 5.35: Example of the optimal route for the broadcast of a DT_1 token from a producer block $P(DT_1)$ towards all the consumers $C(DT_1)$

- For all the reachable consumers, trace back the route until the source is reached and memorize the different intermediate blocks
- Merge the different common parts of the individual routes of each consumer into a single path

As we can see on Fig.5.35, the application of this method results in a graph representing the best route from the producer block $P(DT_1)$ to all the consumer blocks $C(DT_1)$. The broadcast routes towards all the consumers can thus be built for all the produced tokens of a particular source block without having to perform again the Dijkstra's routing algorithm.

Producer/consumer block selection policy

Now that we are able to find a route, we need to define a policy to determine the consumer platform blocks that will be served first. All the blocks waiting for data tokens present in the *waitingForDataTokens* queue compete for the use of the communication network so that we have to use it as efficiently as possible to maximize the data token total bandwidth.
For each consumer C_i in the list, we can get all the data tokens D_k missing for execution and get the different producers P_j for these data tokens. The policy established for the selection of the routes is described by Algo.6 and relies on

two major points:

1 The selection of the consumer block(s) for data routing

2 For each selected consumer block, the determination of the most profitable data token/producer pair

Algorithm 6 Policy for the selection of the consumer/producer/data token combination elected for routing

1: **for** Each data token consumer C_i in the order of the queue of waiting consumers **do**
2: Determine the set D of data tokens awaited by C_i
3: **for** Each possible producer P_i of a token of D **do**
4: Perform the Dijkstra's algorithm with P_i as the source
5: **for** All the tokens D_k produced by P_j at destination of C_i **do**
6: Evaluate the route distance $Dist(P_j, C_i)$
7: Evaluate the route broadcast distance $Dist(P_j, Consumers(D_k))$
8: Select the highest value among $\frac{\#Consumers(D_k)}{Dist(P_j, Consumers(P_j, D_k))}$, $\frac{1}{Dist(P_j, C_i)}$ and the memorized value and put it back in memory as the better solution found yet
9: **end for**
10: **end for**
11: Commit the routing of the path with the biggest $\frac{\#tokens}{RouteDistance}$
12: Reserve all the blocks along the selected route
13: **end for**

The attribution of the communication network for data token routing ends when all consumer blocks have been explored.

Selection of the consumer block Consumer blocks are selected based on a FCFS policy to ensure that the first tokens arrived in the *waitingForTokens* queue will be tested before the others/

Selection of the data token/producer Once the consumer has been chosen, the algorithm will try to find the most profitable route transmitting one of the token that the consumer is waiting for. We first proceed by establishing a list of all the producers able to deliver one of the expected data tokens to the consumer. For each of these producers, we perform a Dijkstra's routing algorithm and then test all the tokens produced. For all the generated data token D_k/producer P_j pairs, we will calculate the metrics represented by Eq.5.8 and select the biggest value representing the most profitable route. This metrics is based on the two following parameters:

- $\#consumers(P_j, D_k)$ represents the number of consumer blocks that can be reached given the current occupation state of the platform blocks for a route starting from P_j and distributing token D_k

- $Dist(P_j, Consumers(P_j, D_k))$ is the total distance of the path linking the producer P_j to *all* the currently reachable consumers $C(P_j, D_k)$ of data token D_k: this route includes, of course, the consumer block C_i that has the priority for data token routing. The total distance is calculated by adding *only once* the routing weight of each platform belonging to the broadcast path so that blocks that are part of common segments will contribute only once to the distance. Indeed a segment used for broadcasting doesn't cost more in terms of network occupation or performance criteria cost than a segment used for a single route.

$$Metric_{token/producer\ selection} = \frac{\#consumers(P_j, D_k)}{Dist(P_j, Consumers(P_j, D_k))} \tag{5.8}$$

Basically the idea lying behind that metric is that we want to find the solution that has the smallest total distance per data token sent. This is a much more fair method for comparing different producer/data token pairs since having more consumers served by a route of a given distance increases its profit.
Once the consumer/producer/data token solution has been chosen, all the blocks along the path are *reserved* meaning that they cannot be used for any other transaction until the data token has crossed them. This mechanism of reservation avoids the overbooking of a block for multiple data token transmissions and several emissions of the same data token to a particular consumer block. The only drawback of this method is that the consumer block won't be available until it has received the data token: however in a system where network latencies are short, this should not be an issue[33].

Calculating events

When a route has been established between a source and one or multiple destinations, the last operation left is the determination of the events for the different blocks involved in the data token transmission. This is required to maintain the consistency of the block states and calculate their release time to allow other routes to use them as soon as possible after they have been used.
The first thing to do is to compute the value of the bandwidth for the whole path. Since several communication blocks can be chained with different individual bandwidth values, we set the bandwidth of the complete path to the minimum value BW_{min} of the different block bandwidths. A token DT_i of $Size_{DT_i}$ data size will thus be sent from the source to the consumer(s) at a rate of BW_{min} meaning that a block without any latency will complete the emission of this token after $t_{emission}$ time units as described by Eq.5.9. If we consider that a path from a source block to a consumer contains N intermediate blocks, all the latencies from the source, the destination and the N blocks must be added to the emission time to get the total transmission time t_{trans,DT_i} from the moment where the token begins to be sent from the source

[33] A solution to this problem will be discussed in the chapter dedicated to future work (see Sec.7.2.1).

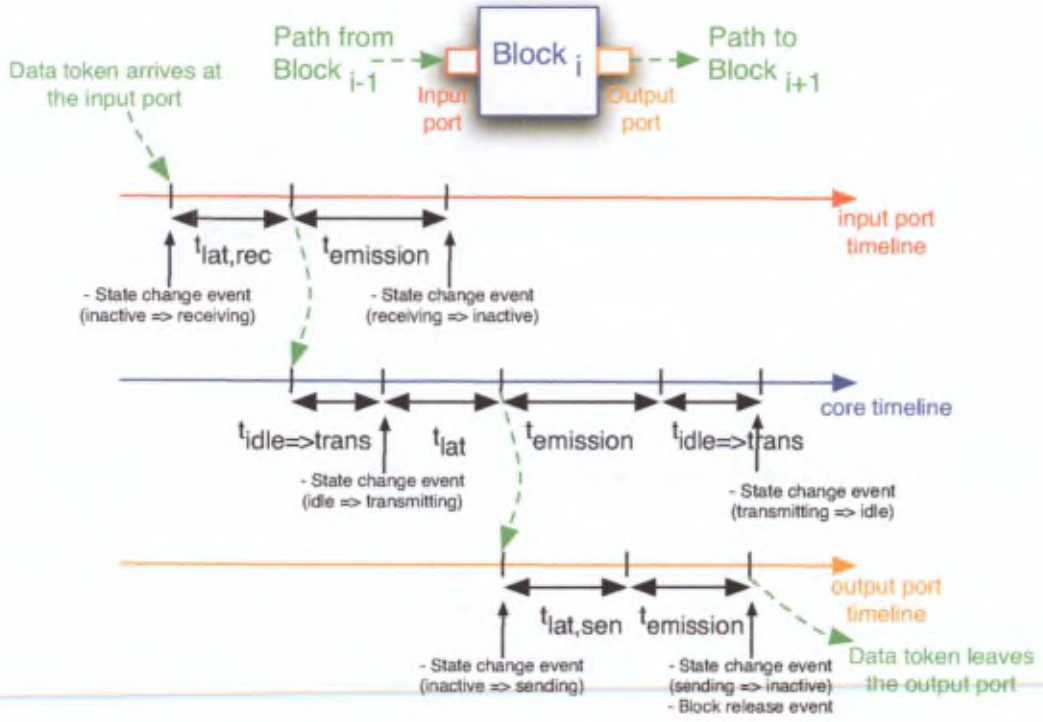

Figure 5.36: Mechanism of event generation inside a platform block based on the results of a routing operation

to the moment where the token has been entirely received by the destination block. This transmission time is given by Eq.5.10 where $t_{i,lat,inPort}$ is the latency of the input port of the block i, $t_{i,lat,outPort}$ is the latency of the output port, $t_{i,lat,core}$ is the transmission latency of the block i and $t_{i,transition_{idle \Rightarrow trans}}$ is the transition time from the *idle* state to the *transmitting* state for block i.

$$t_{emission,DT_i} = \frac{BW_{min}}{Size_{DT_i}} \tag{5.9}$$

$$t_{trans,DT_i} = t_{source,lat,outPort} + t_{destination,lat,inPort} + t_{emission,DT_i} + \sum_{i=1}^{N} (t_{i,transition_{idle \Rightarrow trans}} + t_{i,lat,core} + t_{i,lat,outPort} + t_{i,lat,inPort}) \tag{5.10}$$

Fig.5.36 illustrates how the mechanism of event generation works for all the platform blocks along the route. Considering one intermediate block i through which flows a token, events are generated separately for the input port, the output port and the core of the block. The mechanism is always the same for the different parts (core and ports) of the block:

- Once the data token arrives, an event changing the state of the port/core is scheduled
- After a duration corresponding to the latency, the data token really begins to be sent meaning that the following block part is stimulated
- Once the token has been sent (after a $t_{emission}$ time), the block part state is reverted back to its idle state and an additional event to release that block is scheduled.

Events are scheduled using this method for all the blocks until we reach the destination where an event for the reception of the data token is scheduled. From the figure we can also mention that the core related events are scheduled while taking into account the potential transition time associated to state change (see Sec.5.2.2).

5.5 The Nessie framework

In this section, we describe from a user point of view the input and output of Nessie, give some hints to deal with the management of the input files and explain the different mechanisms used to detect errors in the input files.

5.5.1 Introduction

Nessie is a C++ framework entirely interfaced by the mean of XML files as depicted in Fig.5.37. The program receives only one argument which is the name and location of the input file setting up the performance prediction experiment and its parameters along with other dependent XML input files. Once the input files have been read and turned into a structure of C++ objects, the Nessie internal core proceeds to the evaluation of the performance criteria according to the defined hierarchy and mapping policy. Each possible combination of the degrees of freedom is tested one after the other and the results are reported inside the output files.

5.5.2 Input and output files

The XML files used as input for Nessie are the following:

- *Simulation.xml* is the main file used to initialize a Nessie performance prediction simulation. It contains the platform and functional description (structure definition for the different abstraction levels and a reference to Yeti performance criteria models) and the different structural and parameter related degrees of freedom with their possible values. Additionally, this file defines the different performance criteria and the way to compute them based on their dependence over time and the platform structure. The name of this file is user-defined and is passed to the C++ program as main argument.

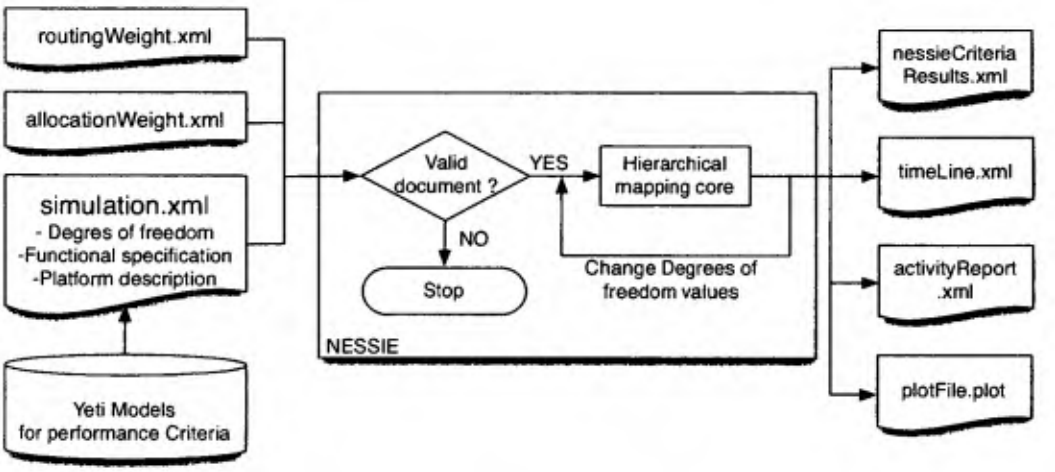

Figure 5.37: Input and output XML file organization in Nessie

- *allocationWeight.xml* is a file defining a Yeti model for the calculation of the weight of the different platform blocks competing for allocation. This model yields for all the abstraction levels and the file name defining it must be respected by the user.
- *routingWeight.xml* defines a Yeti model for the calculation of the incremental cost of the extension of a route through a particular block. This model yields for all the abstraction levels and the file name defining it must be respected by the user.
- *performanceCriteriaModels.xml* defines Yeti models for performance criteria estimation and each of these files is associated to a particular platform/functional primitive combination.

The performance criteria model files may quickly become numerous as the number of abstraction levels and the number of possible functional/platform primitives increases. Although no file naming convention is imposed to the user, it may be interesting to define one and stick to it to avoid file name ambiguities and allow the user to handle the files more easily. We thus advise the user to use the following file naming convention based on a $HW\_X\_Y\_STATE$ description. HW[34] is used as a prefix referring to the definition of the platform performance criteria model, X is the abstraction level of the platform

[34]We may wonder why we deliberately use the prefix HW (SW) to refer to platform (functionality) while we established clearly the difference of vocabulary between those two words in Sec.4.2. The reason is simple: a part of the C++ code and XML schemas were written before we decided to make that difference so that we keep here the terms *hardware* and *software* to remain consistent with the code. Furthermore HW and SW prefixes are much more self-speaking than PT and FC to anyone who is not familiar with our vocabulary.

primitive and Y is the ID of the platform primitive inside abstraction level X. Finally $STATE$ is the state of the platform primitive core/port for which the model is defined: it can be *INACTIVE*, *SENDING* or *RECEIVING* for a port and *IDLE*, *TRANSMITTING*, *MEMORIZING*, *INACTIVE*, *SLEEPING* and *COMPUTING_Z* (where Z is the ID of the compatible functional primitive associated with the platform primitive) in the case of a core state. Allowing this convention leads to a clear organization of the XML files in the repository of models and avoids many possible confusions.
The output files generated by Nessie after a performance evaluation are the following:

- *NessieCriteriaResults.xml* is a file summarizing the results of all the performed estimations: it gives the different performance criteria for each tested combination of degrees of freedom.
- *timeLine.xml* files gives for each solution the sequence of events resulting from the mapping of a particular platform structure on a functional structure. These information deliver the complete scheduling of the functionality and allow the user to visualize how Nessie exactly performed the mapping. Each generated file is named following a particular convention $NessieTimeLine\_SOL\_I\_AL\_J\_HW\_K\_SW\_L$ where I is the ID of the solution among the pool of explored solutions, J is the abstraction level and K is the platform primitive mapped onto functional primitive L.
- *activityReport.xml* is a file containing information about the absolute and relative occupation time of each platform block involved in a mapping. For each state and each part of any block of a platform structure, we can have these occupation time information which can be very helpful to determine the average time of computation or transmission of a particular block and estimate the efficiency of a particular platform structure. The activity report files follow the same file naming convention as timelines.
- *plotFile.plot* is a file containing all the values of the variables to plot and can directly be interpreted by a plot program to deliver the desired graph. The axes values may be selected among degrees of freedom or performance criteria so that it is very easy to visually compare different performance trade-offs.

Sometimes the generation of the activity report and the timeline files is not necessary: Nessie allows the user to disable it when only the final resulting performance criteria values are required. This has the advantage of speeding up the exploration and sparing a lot of memory since simple simulations may quickly generate very large XML output files.

5.5.3 XML format

To define a simulation and its parameters, the user has to create and fill all the XML files with the required information. To avoid any risk of syntax error, all

the XML files used in Nessie come with an associated schema: this enables the automatic verification of the compliance of the XML file with a structured and well-defined grammar (see Sec.B.1 for more details). The Nessie framework will validate the document and generate a run-time exception before exiting if any validation error is found during the parsing. Additionally, Nessie checks some other aspects that schema's are not able to define or constraint. Among others, Nessie will generate an error if there are abstraction level discrepancies in the definition of structures, undefined block ID's to build a structure, degrees of freedom referring to undefined values, undefined models for some required states in a platform block etc. A complete description of XML files, their content and the different aspects checked by Nessie is given in Sec.B.3.1.
Defining complex XML files by hand may quickly become a burden if no appropriate tool is provided to the user. An XML editor is therefore of great use since it is able to automatically complete a document with the mandatory content defined by the user leaving only the user with the only task of entering the pure data and tag-less content. Furthermore a graphical user interface could be easily built in the future above the XML input file layer so that we could keep the verification process intact and integrate it in the core of the tool.

5.6 Implementation

With over than 60 classes (even without Yeti) and a few ten thousands lines of codes, Nessie is a program with a quite complex architecture. Trying to describe it in details from a software point of view is pointless so that we will in this section focus on the most important aspects that bring flexibility and explain the features of the tool. As we also did for Yeti in Sec.2.6, we will only use partial UML diagrams keeping only visible the most relevant information for the purpose of our discussion. Due to the complexity of the complete UML diagram, we will split it into several parts and divide the rest of this section according to them.

5.6.1 Performance criteria and degrees of freedom

Performance criteria and degrees of freedom are the outputs and the inputs of the performance estimation core and are therefore grouped in the same section.

Criteria The criteria related classes are *criteria*, *criteriaResults* and *criteriaSet* and are depicted in Fig.5.38.
The *criteria* class attributes include the name, a time dependence rule and a structural combination rule defining the way to calculate this particular criterion for a platform block. *EstimateCriteria* is the main method of this class enabling the calculation of the criteria value evolution over time and over the different parts of a platform block: this method uses the value of the criterion

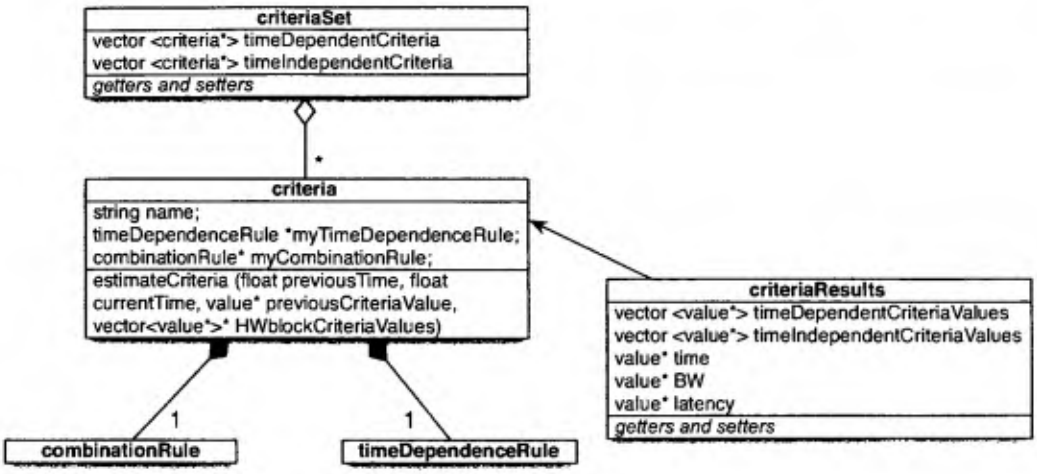

Figure 5.38: UML diagrams for criteria related classes in Nessie

at the previous time step and its value for the ports and core of the platform at the current time step and returns the resulting criterion value.
The *criteriaSet* class is used to gather all the criteria defined within the current performance estimation simulation and classifies them into time dependent and time independent categories to provide the mapping method with consistent information about the different criteria.
The *criteriaResults* class represents the results of the criteria estimation including all the time (in)dependent criteria along with time, latency and bandwidth if relevant. This is particularly useful to communicate with other classes and encapsulate into one object the criteria resulting from the mapping of a functional structure onto a platform structure. Furthermore this object can then be stored to avoid the evaluation of already calculated criteria and the waste of time resulting from redundant evaluations.

Time dependence and combination rules *TimeDependenceRules* and *CombinationRules* respectively represent the method used to combine criteria values over time and over the platform structure to obtain the resulting criteria value (see Fig.5.39 and Fig.5.40). For both classes we have defined subclasses to specify different time and combination rules and make use polymorphous methods for their evaluation.
The *timeDependenceRule* has three different subclasses:

- *additiveTimeDependenceRule* adds the current criteria value to the criteria value of the previous time step
- *maxTimeDependenceRule* compares the current criteria value with the

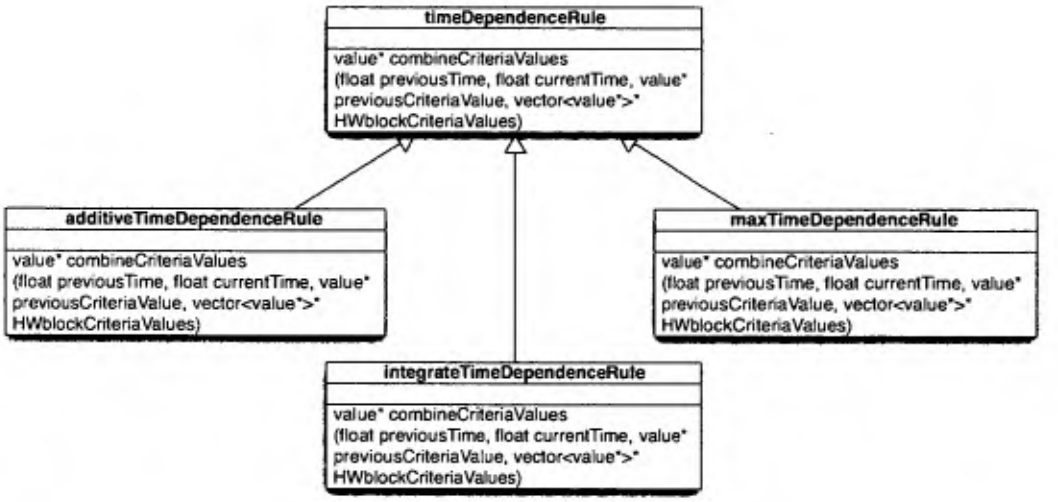

Figure 5.39: UML diagrams for time dependence rule related classes

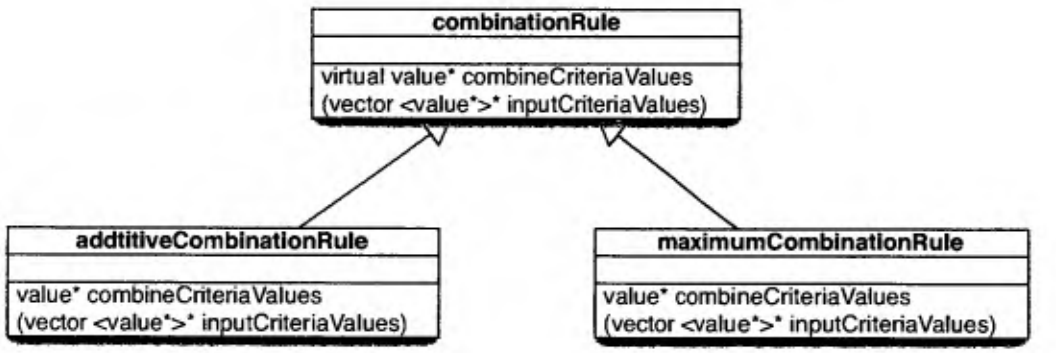

Figure 5.40: UML diagrams for combination rule related classes

criteria value of the previous time step and selects the maximum

- *integrateTimeDependenceRule* adds to the previous criteria value the difference of the current value and the previous criteria value multiplied by the difference of the time stamps.

The combinationRule has two different subclasses:

- *additiveCombinationRule* sums all the criteria values of the different parts of a platform block or the different blocks of a platform structure
- *maxCombinationRule* selects the block/part of the platform block with the maximum criteria value.

Thanks to this inheritance mechanism, new rules can be easily added in no time by defining new subclasses deriving from the corresponding superclass.

Degrees of freedom Each degree of freedom is represented inside Nessie by an instance of the *DoF* class containing all the information to change the different degrees of freedom values that define one particular solution. This super class contains a static reference to the *explorer* (see Sec.5.6.2) to enable DoF subclasses to change parameters related to the hierarchical exploration. The two main methods allow the other classes, whatever the exact nature of the degree of freedom defined by the subclass, to get the number of possible values for their degree of freedom and to set it to one of these possible values. Thanks to this mechanism, we are able to modify a degree of freedom independently of its nature which is very convenient to establish a transparent interface to build any design space exploration on top of it.
The three different classes deriving from the *DoF* superclass are the following (see Fig.5.41):

- *HWstructureDoF* is a degree of freedom defining the platform structure used for a specific abstraction level and platform primitive
- *SWstructureDoF* is a degree of freedom defining the functional structure used for a specific abstraction level and functional primitive
- *valueDoF* is a degree of freedom defining the value of a Yeti local or global parameter used in an analytical model for performance estimation.

5.6.2 Hierarchy

This part describes all the classes and implementation mechanisms surrounding the hierarchical functional/platform description and evaluation in Nessie. Fig.5.42 represents a UML diagram summarizing the different classes and their composition that we describe in the coming paragraphs.

SWtypes and HWtypes *SWtype* and *HWtype* are the classes representing the functional and platform primitives and are very similar due to the close interaction existing between the functionality and the platform.

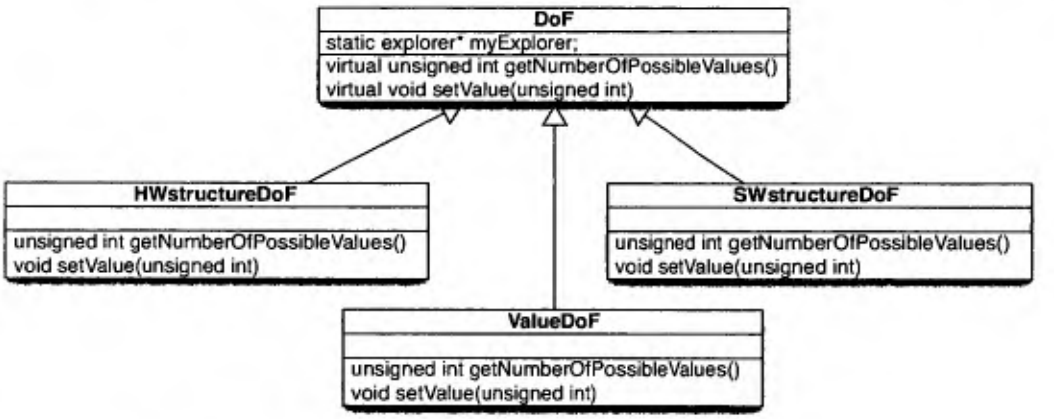

Figure 5.41: UML diagrams representing degrees of freedom related classes

The *SWtype* contains attributes to define the abstraction level, the ID inside this abstraction level, the size of the output tokens produced by this functional primitive and all the possible functional structures that can be used to define this functional primitive at the lower abstraction level. Additionally the class contains a list of all the functional parameters used in Yeti models and a list of all the platform primitives that can execute this functional primitive at this abstraction level.
The *HWtype* contains attributes defining the abstraction level, the ID of this platform primitive within the abstraction level, platform structures related to this primitive and a list of the platform parameters used for the *Yeti* model. More importantly, platform primitives define Yeti behaviour objects for the performance criteria evaluation of the core states, the computing state associated with the different functional primitives and the port states. The methods contained in *HWtype* enable the use of Yeti models (in the case where no further hierarchical exploration needs to be performed) and return a *criteriaResult* object holding the evaluated performance criteria values.
One of the important role of the HWtype is also to store the *criteriaResults* objects resulting from the functional/platform mapping combinations already explored: this saves a lot of computation time by avoiding many redundant evaluations.

SWhierarchy and HWhierarchy *SWhierarchy* and *HWhierarchy* respectively contain all the functional and platform primitives organized by abstraction levels and ID in vector to ensure an easy access to these data.

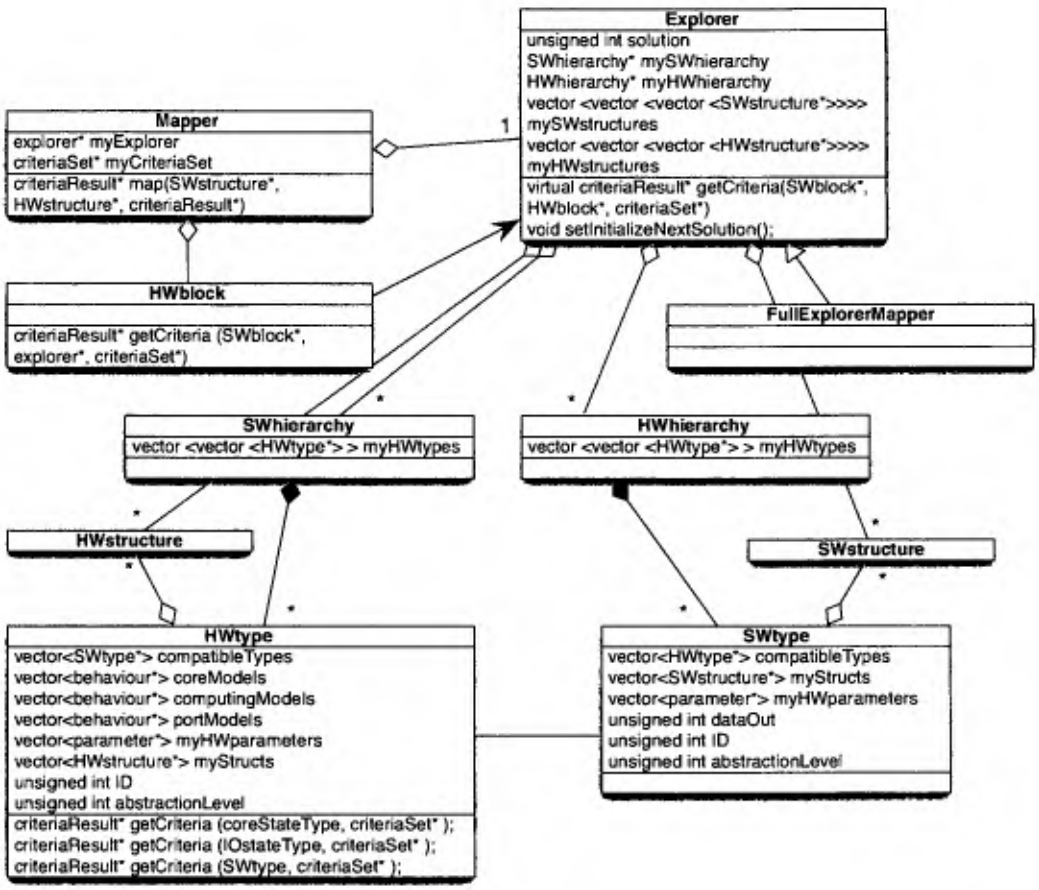

Figure 5.42: UML diagrams representing the classes related to the functional and platform hierarchy

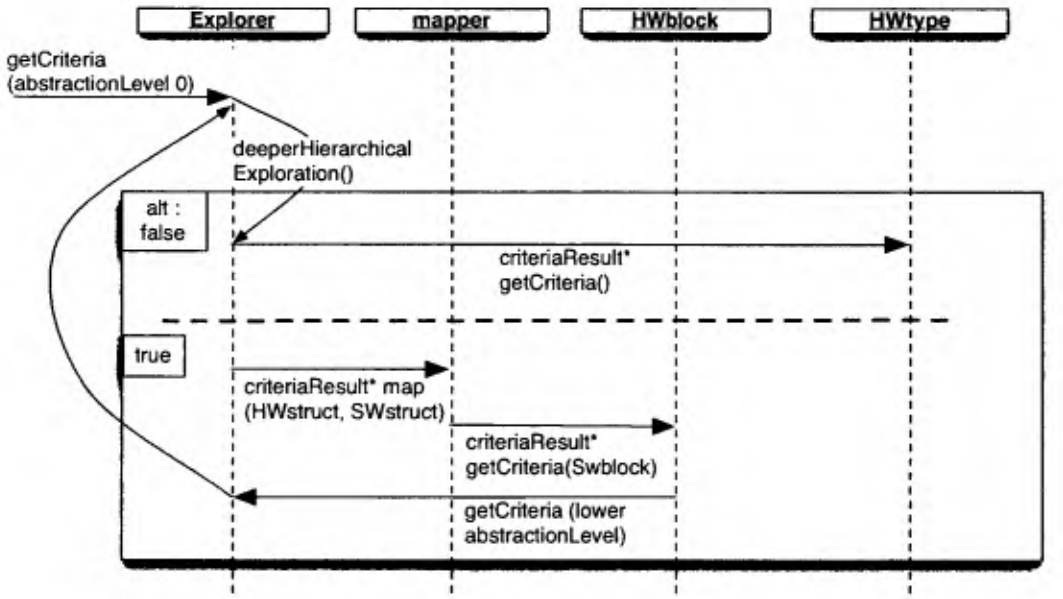

Figure 5.43: UML collaboration diagram describing the message passing mechanism between the different actors responsible for performance estimation.

Explorer The explorer and the mapper have an important role in the performance criteria evaluation: the first one defines the method of hierarchical exploration while the second one maps a functional structure onto a platform structure and returns the resulting criteria values.

The *explorer* contains a reference to all the platform and functional structures organized by abstraction levels providing all the necessary information to implement any hierarchical exploration policy. It also maintains a counter of the currently performed solution and can reset all the parameters of the platform and functional structure to initialize the exploration of a new solution. To enable the easy definition of different design space exploration policies, subclasses of *explorer* can redefine the virtual method *estimateCriteria*. At the moment, only the full-depth exploration policy has been defined which consists in exploring the hierarchy as long as there are structures associated with primitives (this is defined by the structured degrees of freedoms). This inheritance mechanism enables to easily add other policies in the future without having to add anything else than a new subclass in the code.

The message passing between the different classes involved in mapping and exploration is described in the collaboration diagram of Fig.5.43. To initiate the performance criteria estimation, we call the explorer *getCriteria* method for the abstraction level 0 corresponding to the mapping of the application on

the chip. According to the exploration policy of the explorer subclass, we have two different possible choices:

1 No exploration of the lower abstraction level is desired. In that case, the *getCriteria* HWtype method is called, triggers the evaluation of the corresponding Yeti model and the *criteriaResult* object is returned to the explorer.

2 The exploration of the lower abstraction level is required. In that case, a functional and a platform structure determined by the explorer (according to the degrees of freedom or its internal exploration policy) are passed as parameters to the *map* function of the mapper class to get the resulting *criteriaResult* object. Following its own policy, the mapper will map the functional structure onto the platform structure and will, when necessary, call the *getCriteria* method of the *HWblock*. The latter will in turn ask the explorer for the performance criteria estimation of a functional/platform block couple of the lower abstraction level: this will trigger a new request for exploration enabling the recursive estimation of criteria along the different abstraction levels. This exploration will thus go on until the exploration policy defined by the explorer decides not to go down in the hierarchy and uses Yeti models to get the performance criteria.

This message passing mechanism has two main advantages:

- The mapping and the exploration processes are completely independent thanks to the *mapper* and *explorer* class separation and their communication mechanism. Changing the exploration policy inside the *mapper* class will thus have no impact on the *explorer* and inversely.
- The explorer is always called when a functional block is mapped onto a platform block and will, before deciding to call the mapper or use a Yeti model, check if this combination has already been tried. If it's the case, the explorer will thus provide the memorized *criteriaResult* as a result instead of performing the performance estimation once again sparing a lot of computational time.

5.6.3 Functional structure and petri nets

This section deals with the implementation of the functional structure and describes how Nessie provides the support for multiple models of computation while presenting the current petri net implementation. The UML diagram including both these aspects is pictured in Fig.5.44.

Functional structures Inside Nessie, we define the superclass *SWstructure* from which derives each particular functional structure representing a model of computation: this enables to handle all the functional structures independently from the nature of the underlying MoC. Each functional structure may be

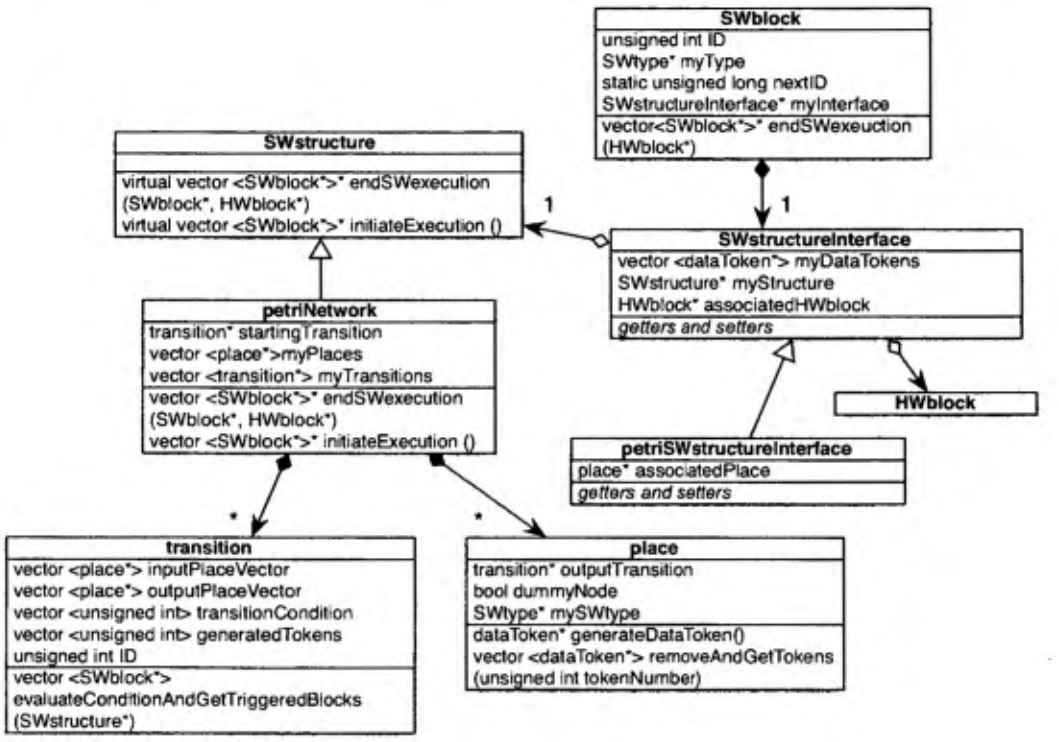

Figure 5.44: UML diagrams representing functional structure related classes

composed out of their own building blocks (nodes, arrows, places, transitions, conditions, etc.) but from the outer side, they communicate using *SWblocks* which are instances of functional primitives (also called functional blocks).
The two main polymorphous methods of *SWstructure* are *initiateExecution* and *endSWexecution* that respectively return the first *SWblocks* to execute after initialization of the structure and the *SWblocks* that may be triggered after the execution completion of a particular *SWblock* on a given *HWblock*. These functions are totally independent from the underlying model of computation execution semantics and enable therefore an easy communication interface with classes using it. However once the *SWblocks* issued by a *SWstructure* have been executed on the platform, we need in return to send additional information that are specific to this particular *SWstructure* subclass: this is why *SWblocks* contain another object inherited from the *SWstructureInterface* class.

SWblocks and SWstructureInterface The *SWblock* objects represent instances of functional primitives tagged with an ID upon their creation by a *SW-*

structure. Additionally it has an attribute called *interface* that is an instance of the *SWstructureInterface* superclass. This class contains attributes providing information enabling the *SWblock* communication with the functional structure that created it and its execution on the platform. Aside from the *SWstructure* that produced the *SWblock*, this classs defines a vector containing all the data tokens required by the *SWblock* for execution and the *HWblock* that the *SWblock* is mapped onto. Since *SWstructureInterface* is a superclass, additional information about the specific model of computation can be added in the subclass allowing the *SWblock* to be used again by the stucture once it has been executed. In our case we can see that a *petriSWstructureInterface* has been defined as a potential subclass containing the *place* inside the petri newtork where a petri token should be generated upon *SWblock* execution completion.

SWstructure communication: the big picture Now that we have described the different individual mechanisms for functional structure implementation, let us describe point by point the different steps used from the creation of *SWblock* until its execution completion for the example of a petri net functional structure:

- A *SWblock* is created by the petri network
- A new *petriSWstructureInterface* is attached to the this *SWblock* and its place attribute points towards the petri net that will receive a token upon *SWblock* execution completion
- This *SWblock* object is returned to the *mapper* class for execution on the platform
- Once the execution is finished, the polymorphous *endExecution* method is called using the *SWstructure* object reference of the *SWstructureInterface*: the method will thus be called in the *petriNetwork* class.
- Before the *petriNewtork* can generate the tokens, test the transition conditions and generate resulting *SWblocks* if required, we need to extract the information related to the petri network out of the *SWstructureInterface* of the *SWblock*. The trouble is that we have a pointer referring to the superclass instead of the subclass *petriSWstructureInterface*: to solve that issue, we make a dynamic cast of the object. If this method is often quite risky since it may generate run-time errors, we have no problem in our situation since we know that a *SWblock* starts and ends its life in the scope of the same *SWstructure* subclass instance.

Petri network The *petriNetwork* offers different methods for the execution of petri networks and is built of *transition* and *place* objects.
The *place* class has attributes indicating the functional primitive of the operation linked to this place, a link to the output transition and a flag to make the

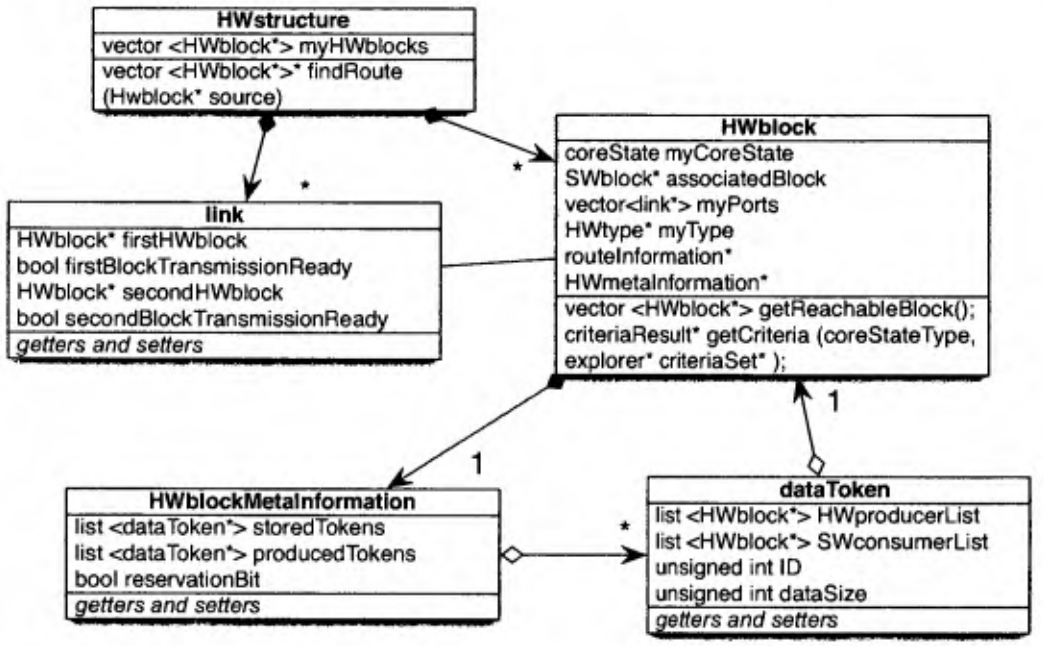

Figure 5.45: UML diagrams representing platform structure related classes

difference between normal and dummy nodes (see Sec.5.3.3 for more details about dummy nodes).
The *transition* class contains a vector of input and output places, the number of tokens required for each place to cross the transition and the number of tokens generated in return.

5.6.4 Platform structure

Contrarily to functional structures setting up implementation mechanisms to support the interoperability of different MoC's inside the performance estimation core, there is only one single possible representation for the platform structure: its implementation is thus much simpler. The diagram for the platform structure related classes is represented in Fig.5.45

HWstructure The platform structure is represented by the *HWstructure* class containing a vector of *HWblock* objects whose ports are connected together by the mean of *links*. Apart from getters and setters, the *HWstructure* class contains all the methods used to explore the platform structure and find routes from a source block to a destination block.

HWblocks The *HWblock* objects represent instances of a platform primitive and include all dynamic information about the block. The attributes contain, apart from the reference to the platform primitive, a list of all the instantiated ports and the state of the core. In addition, the *HWblock* contains a *HWblock-MetaInformation* object containing a list of all the data tokens stored in the memory of the block or available at the input ports but also an object *route-Information* holding several useful information for the routing process. The *HWblock* contains methods related to routing and estimation of the performance criteria based on the state and Yeti performance model of the different parts of this block (the core or the ports).

Links The *link* class contains all the information about the connected blocks and the bi-directionality of the connection.

5.6.5 Mapping

The mapping process concentrates a large part of the total code inside Nessie and describing it in details would be a waste of time. In this section, we will try to highlight the most important aspects related to the time management in the performance estimation process. Fig.5.46 illustrates the different classes involved in the mapping part of the implementation namely the *mapper*, the *timeLine*, the *timeNode* and the inherited *event* classes.

Events The *event* superclass main role is to define how an event reacts when triggered. Since there are many different possible events, we have define for the four different types of events subclasses inheriting from the superclass *event* and redefining the polymorphous method *triggerEvent*. Each subclass has its own attributes to characterize the nature of this precise event and contains getters to access to it. We also need to mention that the *event* class includes a reference to the mapper that issued it: its use will be justified in the following part devoted to the description of the mapper.

The mapper The *mapper* is one of the most complex classes because it contains all the methods related to the mapping and manages the three different stacks of *SWblocks* used during the mapping process (more details about these stacks can be found in Sec.5.4.3). Among the methods, we have the *map* function used to trigger the estimation of the performance criteria, the *route* and the *scheduleAndAllocate* methods.
Since the mapper manages the performance estimation from an event-centric point of view, each event will have an impact on the state of the different *SWblock* instances and their position in the three stacks. However these attributes (and many others) are part of *mapper* class so that the *triggerEvent* method of the events should have access to all these attributes. From the implementation point of view, this is not a good idea for two different reasons.

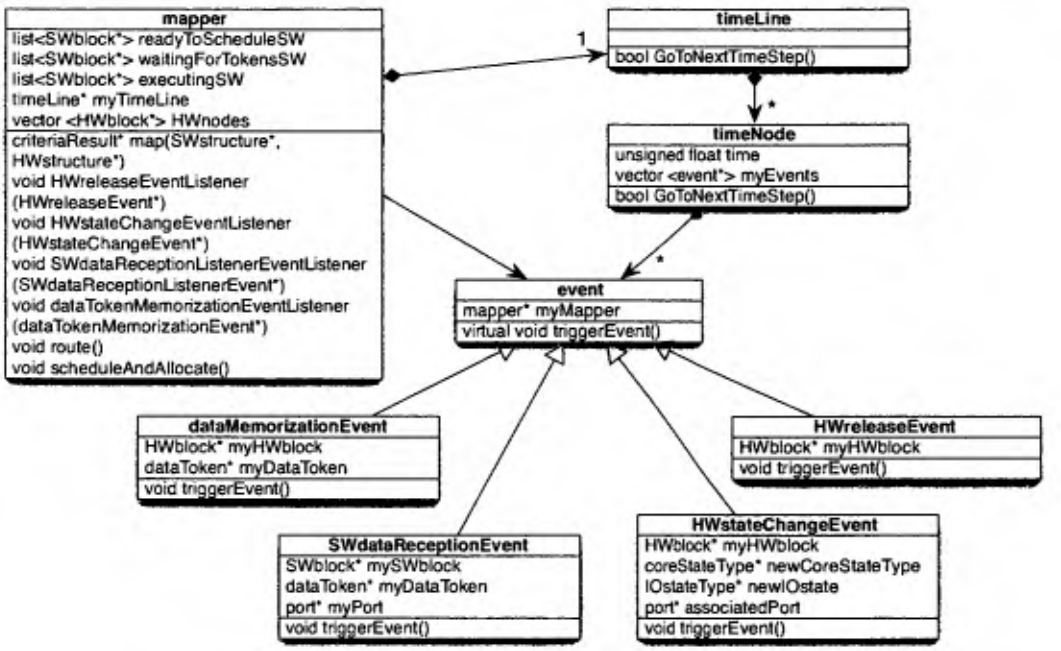

Figure 5.46: UML diagrams representing mapping related classes

First, almost all these mapper attributes should be passed to all the events which is neither efficient neither safe in terms of attribute encapsulation. Second, the process triggered by an event is part of the mapping so that it should not belong to events and be split among all the event subclasses.
To avoid that problem, we have implemented an *event listener* inspired mechanism. Each time an event is produced by the mapper, it is stored inside a timeline keeping track of the progression of the simulation. Once this event is popped out of the stack to be triggered, the *triggerEvent* method is called in the related event subclass: this function will then call in return a method of the mapper class (using the mapper reference of the event superclass) called an event listener and specific to the particular event that called it. This listener function will receive as parameter the event subclass that called it so that the mapper is able to extract the information contained in this event by invoking specific getters. This mechanism has the double advantage of putting back all the mapping related code inside the mapper and partitioning the event information inside dedicated subclasses. Adding a new type of event thus consists in defining a new subclass inheriting from the *event* superclass and adding the corresponding event listener method inside the mapper class.
Many other attributes and methods are also present in this class but we advise

the reader to refer to the code to have more details about it.

timeline To give the support to events management, we have defined a class called *timeLine* allowing a very dynamic and simple use of events. The timeline is based on an internal time reference holding the current time value of the simulation and the last triggered event that has been accessed. The main advantage of the timeline over a simple array management is that it will automatically order in time an event when it is added and provide a very easy way to explore all the contained events. Furthermore it allows forward and backwards traversal and holds all the past events which is quite convenient to generate reports related to the event activity.

5.7 Conclusions

In this chapter, we have discussed all the concepts, algorithms and implementation issues related to our performance prediction tool called Nessie. Compared to other performance prediction tools defined in the state of the art of the previous chapter, Nessie tends to offer more flexibility regarding the design space exploration exploration policy, the mapping process and the platform/-functionality representation.
In summary, we can say that Nessie provides the user with the following features:

- Automatic design space exploration based on a user-defined policy. This mechanism relies on an interface hiding the performance evaluation core to only expose inputs (degrees of freedom describing the design choices) and outputs (performance criteria) to the design space exploration policy.
- Flexible performance criteria specification: the user defines how to calculate each criterion over platform blocks and over time
- Separate hierarchical description of the functionality and the platform for easier mapping
- Automatic mapping of a functionality onto any generic platform made out of computation nodes, memories and communication blocks. The mapping policy is performed in three steps (scheduling, allocation and routing) and is driven by user customizable models defining an objective function used to optimize the allocation and the routing.
- Interfacing with Yeti, our closed-formed based models description and evaluation library. These analytical and dynamically built models bring a lot of flexibility to Nessie and provide an alternative to complete functional/platform mapping at any abstraction level.
- Automatic generation of result files including resulting performance criteria values files, activity reports and time lines reports defining in details the different steps performed by the mapping process

- Strict verification of the user input files relying on an XML schema grammar combined with run-time error management to make simulation initialization and running easier for the user

Now that we have presented in details Nessie, we still need to validate its different features on practical cases to demonstrate the proper working of Nessie and show how it can model realistic systems and estimate their performances: this is the topic of the next chapter.

Bibliography

[1] A. Schärlig, *Décider sur Plusieurs Critères, Panorama de l'Aide à la Décision Multicritère.* Lausanne: Presses Polytechniques et Universitaires Romandes, 1985.

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *ISCA*, 2000, pp. 83–94. [Online]. Available: citeseer.ist.psu.edu/brooks00wattch.html

[3] J. Laurent, N. Julien, E. Senn, and E. Martin, "Functional level power analysis: An efficient approach for modeling the power consumption of complex processors," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe.* Washington, DC, USA: IEEE Computer Society, 2004, p. 10666.

[4] W. Reisig, *Petri nets: an introduction.* New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[5] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=24143

[6] G. D. Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, 1994.

[7] K. Bazargan, "Ee 5301 - vlsi design automation i : High level synthesis," University of Minnesota, Tech. Rep., 2003.

[8] W.-T. Shiue, "High level synthesis for peak power minimization using ilp," in *ASAP '00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors.* Washington, DC, USA: IEEE Computer Society, 2000, p. 103.

[9] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level synthesis: introduction to chip and system design.* Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[10] G. N. Mangalam, S. Narayan, P. van Besouw, L. Avra, A. Mathur, and S. Saluja, "Graph transformations for improved tree height reduction," in *VLSID '03: Proceedings of the 16th International Conference on VLSI Design.* Washington, DC, USA: IEEE Computer Society, 2003, p. 474.

[11] C. Chang, C. Chen, and C. King, "Using integer linear programming for instruction scheduling and register allocation in multiissue processors," 1997. [Online]. Available: citeseer.ist.psu.edu/article/chang97using.html

[12] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961.

[13] R. Walker and S. Chaudhuri, "High-level synthesis: Introduction to the scheduling problem," 1995. [Online]. Available: citeseer.ist.psu.edu/walker95highlevel.html

[14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available: http://jmvidal.cse.sc.edu/library/dijkstra59a.pdf

[15] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

Chapter 6

Nessie: Case Studies and Applications

Abstract

In this chapter, we illustrate Nessie features through different case studies and demonstrate how it can be used to perform design space exploration by simultaneously exposing degrees of freedom of the application, the platform and the mapping.

In the first part, we propose different design studies relying on fairly simple and hypothetic architectures and applications to enable easy interpretation of the results and confront them with intuition. Even with such simple examples, we will be able to draw very interesting results and demonstrate how Nessie performs an automatic exploration of the solutions based on an automatic application/platform mapping in order to allow the designer to graphically compare the different design trade-offs on a multicriteria performance basis.

In the second part, we show how we can use Nessie flexible representation to model an existing H.264/AVC decoding application and its network-on-chip based platform. We compare our results with several experiments to validate them and extend them to the case of a 3D stacking architecture for a very small effort. We discuss the power consumption reduction achieved and compare it to the different costs related to the use of 3D stacking techniques.

Finally we define the strengths and weaknesses of Nessie approach based on the experience relative to the two previous case studies.

6.1 Introduction

In the previous chapter, we have extensively discussed the different features of Nessie and explained all the related algorithms and underlying mechanisms. In order to validate our framework, this chapter focuses on different case studies to test and demonstrate the most important features which are:

- Separate description of the functionality and the platform
- Automatic mapping of the functionality on the platform
- Multicriteria performance estimation
- Definition of multiple design degrees of freedom (parameter-based, platform/functionality structure)
- Automatic plotting of each possible DoF/performance combination

The case studies will be divided into two different parts:

- We first map a simple hypothetical application on different architectures in order to illustrate the basic mechanisms of Nessie and demonstrate how easy and straight-forward it can be to perform design space exploration using our tool. Even if we tried as much as possible to use realistic values for the different input parameters, we wanted to keep examples quite simple so that we easily interpret the results demonstrating that Nessie is doing what we expect from it.
- To prove that Nessie is able to handle more complex and realistic examples, we took a research case studied in our team and tried to formalize it inside Nessie. The case study consists in a H.264/AVC functionality mapped onto a MPSoC platform for which power consumption of the network-on-chip has been determined experimentally. In this example we will try to reproduce these power consumptions values and see in which extent we can apply the case study to 3D-satcking architectures to evaluate Nessie flexibility.

In this chapter we won't demonstrate design speed up resulting from the use of Nessie compared to a design flow without performance prediction. Indeed such a demonstration would require to exhaustively study a real design case (including its design time) to establish a fair comparison of our method with a classical approach which requires too much time to be studied in this thesis: another ongoing PhD is currently addressing this particular problem.

6.2 Design space exploration with Nessie

6.2.1 Introduction

This first part of the chapter focuses on the demonstration of Nessie features and design space exploration capabilities. As an illustration, we start from a

simple application and architecture that we progressively make more complex by adding degrees of freedom and testing more complex structures. Through these different examples we examine the evolution of three design criteria (the computation time, overall energy and surface) and show how Nessie can help the designer to take decisions based on the results delivered by the simulation. We will try to explain the conclusions resulting from the interpretation of the automatically generated curves and see how they match intuition.
Since we only try to demonstrate Nessie functionalities and prove that it delivers the expected results in simple cases, we don't want to define too complex case studies where interpretation would be too difficult. Therefore we have chosen to define fictive structures both for the application and the architecture but tried as far as possible to use realistic values for the parametric degrees of freedom in order to deliver significant results.

Methodology For this case study, we will proceed in two different steps:

1 We describe a single processing node with internal memory that has to execute an application exhibiting some parallelism. We then remove the internal memory and replace it by an external one for the computing node to communicate with and examine the impact on performance.

2 We use the previous application and map it onto a parallel platform to measure the performance gain on computing time and see the impact on energy. We compare different platforms and topologies and discuss the results.

To initialize a Nessie simulation, we first have to define the application and platform structures, the performance criteria used to compare the different solutions and the degrees of freedom that we can play with.

Performance criteria Through these following case studies, we will always use three different performance criteria:

- The *energy* criterion is additive on the surface and based upon the integration over time of the power consumption value of the different instantiated platform blocks (both static and dynamic)
- The *computation time* represents the total time for the functionality to complete its execution
- The *surface* criterion is independent on time but additive on the different blocks which gives at the end the total silicon area of the die.

These criteria of different nature are interrelated and dependent on each other so that their simultaneous optimization will probably require some compromises hence offer some interesting discussions.

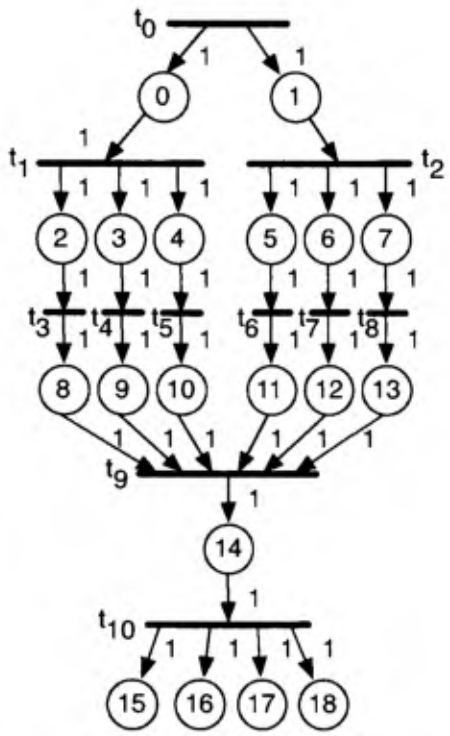

Figure 6.1: Petri newtork modeling the application used for our case study

Application For the purpose of our case study, we define an hypothetic application whose operation execution dependency is defined by the Petri network represented in Fig.6.1. As we can see it contains 19 places and all transitions require only one token to trigger so that the completion of the Petri network execution will require 19 operations. Examining the topology of the network, we can see that its major part is composed out of two identical operation flows that meet at transition t_9 where all the tokens are sent to place 14. These previous flows offer some very interesting parallelism possibilities with a maximum of 6 potential operations being performed at the same time. This application shows a good mix of different execution dependency patterns and will thus be used as a benchmark for our different architectural candidates.
Concerning the different functional primitives, we will only have to define the size of the data tokens produced by the associated operation. Our only parameter related to the functional primitive will be the number of enclosed instructions to perform called and is Op.

Architecture The architecture (hence the platform structure) will be specifically defined for each case that will be examined in the coming sections. However we have to specify models associated to platform primitives in order to be able to estimate the performance criteria of each atomic instance using Yeti (see Sec.5.1.3 for more information about atomic blocks). Since each functional primitive is defined by the number of instructions to execute, the platform primitive will offer a certain execution throughput expressing how many instructions per second can be processed, in other words the IPC parameter. The performance criterion t, the time required to complete the execution of the associated operation, will thus be simply defined by Eq.6.1.

$$t = \frac{Op}{IPC} \tag{6.1}$$

To calculate the energy criterion value, we have to define the power of each platform primitive in the different possible states: the dynamic power will correspond to the power consumed in the computing state while the static power will be associated to the idle state. The dynamic power P_{dyn} will be explicitly defined for each block while the static power P_{stat} will be calculated using Eq.6.2 where $Area$ is the performance criteria corresponding to the platform primitive area and $Coef_{stat}$ is a multiplication coefficient.

$$P_{stat} = Coef_{stat} * Area \tag{6.2}$$

All these different models will be defined in Yeti and instantiated for each platform primitive using global or local parameters when required (see Sec.5.1.4 for a detailed explanation about the locality of parameters).

Degrees of freedom Since degrees of freedom depend on the architecture, application and mapping, we will define in time the different available degrees

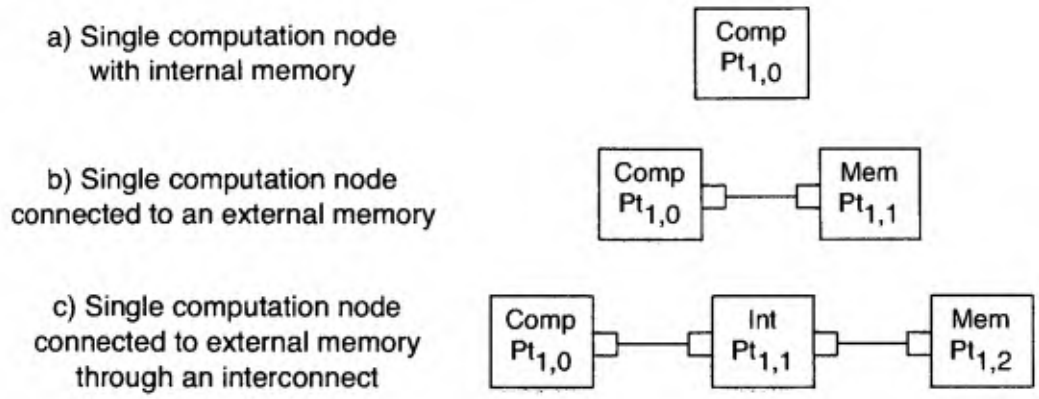

Figure 6.2: Single computation node architecture (a), with an additional external memory (b) and its communication bus (c)

Primitive	Data token size (bits)	Number of instructions
$Fc_{1,0}$	1.0e+5	1.0e+8

Table 6.1: Value of the functional primitive $Fc_{1,0}$ parameters

of freedom.

Now that we are done, let us start with the first case study: the single computation node.

6.2.2 A single computation node architecture

The first considered architecture is a single and isolated computation node with internal memory (as represented on top of Fig.6.2). The first question that we would like to answer is how and in how much time will this simple architecture be able to execute our application?

Degrees of freedom and parameter values

To simplify the problem and ensure an easier interpretation of the results, we first assume that all places of the Petri network (see Fig.6.1) refer to the same functional primitive $Fc_{1,0}$ with parameters values defined in Table 6.1. The platform primitive $Pt_{1,0}$ used for the definition of the computation node has parameters values expressed in Table 6.2. These numbers were chosen so that they fit realistic embedded processors performances for a $180nm$ technology:

- The absolute area of ARM7 based cores ranges from half a square mil-

Primitive	area(mm^2)	$Coef_{stat}$	IPC	P_{dyn}(mW)
$Pt_{1,0}$	1	0.01	1.0e+8	100

Table 6.2: Value of the platform primitive $Pt_{1,0}$ parameters representing the computation node

limeter to a few square millimeters[1]. Therefore we choose $1mm^2$ as a realistic value for the area of a processor of that generation.

- A realistic value of 100 MIPS was chosen for this computation node.
- Embedded processors of that generation show usual numbers around 1 MIPS/mW regarding the power efficiency[2]: with the chosen IPC, we thus obtain a value of $100mW$ for the dynamic power.
- The static power coefficient $Coef_{stat}$ is defined so that the static power leaking when the processor is in idle state equals a tenth of the dynamic power consumption ($10mW$ in our case). This ratio between static and dynamic power consumption is a very realistic value for a 180nm processor[3].

As a result of these parameters values, the execution of one instance of the functional primitive exactly takes one second since the number of operations to perform equals the IPC of the platform primitive. This will highly simplify the interpretation of further results as we will see in the coming sections.

Case of an isolated node

Now that all the degrees of freedom have been determined along with their values, we are able to define our XML input files (the simulation file and the different Yeti model files) and test the mapping of our application on the single node.
The results from the simulation are the following: the application takes 19 seconds to complete its execution for a total energy of 1.9J. These numbers are of course very predictable and consistent with the computation paradigm and the mapping defined inside Nessie. Indeed this node is only able to execute one functional block at a time and then stores the resulting data token inside its internal memory for further use. In the case of internal memories, Nessie assumes that the read/write accesses add no penalty to the total execution time[1]: as a result the execution of the application simply consists in executing

[1] If memory accesses need to be explicitly defined, their contribution to the total execution time can be easily estimated in two different ways using either Yeti models or Nessie structural representations. First we can modify the Yeti model to add a term to the execution time model representing the time penalty due to read/write operations defined by a parameter. Second we can represent the single computation node and the functional instance at a lower abstraction level: on the platform side, we define one or several platform blocks devoted to pure computation exchanging data with data memories and on the functional side, we use a petri network based structure to describe the operation and memory accesses patterns.

Primitive	area(mm^2)	N_{bits}	Latency	Freq(MHz)	P_{stat}(mW)	P_{dyn}(mW)
$Pt_{1,1}$	0.3	64	0	1	1	30

Table 6.3: Value of the platform primitive $Pt_{1,1}$ parameters representing the memory

functional blocks one after the other and using the stored data tokens to trigger the following one. Running the application on this platform thus results in nineteen successive executions of a one second long operation leading to a total execution time of 19s. Since the processing node is always computing, it always stays in the computing state where power equals 100mW during 19s leading to an energy consumption of 1.9J.
To move towards a more realistic architecture, let us now examine how the introduction of an external memory modifies the performances.

Single computation node with memory

In this case, we decided to add a memory to the architecture and remove the internal memory from the computation node to observe the impact on performances. This architecture is represented in the middle of Fig.6.2 and is composed out of the memory node directly connected to the computation node through ports. To simplify the problem, we have supposed that all ports have zero latency and an infinite bandwidth so that the transmission rate is only limited by the memory itself. This memory is represented by the new platform primitive $Pt_{1,1}$ and is, as previously, defined with all the appropriate Yeti performance models. The different parameters for this functional primitive are summarized in Table 6.3: again we tried as much as possible to keep reasonable and realistic values.
As we can see from this table, the bandwidth of this memory equals the product of the word width N_{bits} by the write/read frequency $Freq$ which gives 64Mbits/s. As a reminder, the data token produced by the $Fc_{1,0}$ functional primitive equals 1Mbit.
Launching a Nessie simulation with these parameters gives a computation time of 19.54s and an energy consumption of 1.97J. Again these results were quite predictable: each time an operation is completed, the computing node with no internal memory first tries to send the resulting data token to a nearby memory. Since the only node that has memorization capability is the second platform block, the computing node will transmit its resulting data token to the latter. The computing node is now in idle mode and ready to execute another operation which will require a previously saved data token: these data will be fetched from the memory and sent to the computing node. Given this mechanism and the relatively small data token size/bandwidth ratio, data transmission time will be small compared to the computing time leading to performance criteria values close to the single computing node case.
If this external memory doesn't change a lot the computation time compared

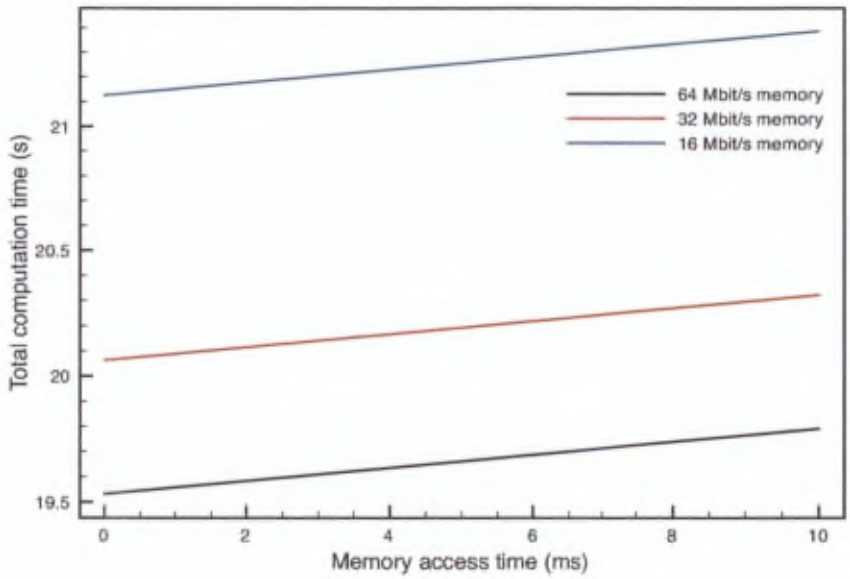

Figure 6.3: Impact of memory access time on total computation time for three different memorization bandwidths

to a single computing node case, it would however be interesting to quantify how much different values of memory latency and bandwidth would. Therefore we made the latency vary from 0 to 10ms for three different bandwidth corresponding to 1, 0.75 and 0.5 Mhz/word bit: the resulting graph is plotted in Fig.6.3. This plot was obtained using the plot file generation feature of Nessie: we just had to specify in the input simulation file that we wanted performance criterion $Time$ versus access time Lat_mem to be represented and the resulting plot was automatically generated after the simulation.

At first sight, we see a clear influence of the memory bandwidth on the computation time: the delay penalty seems to grow much than the bandwidth decreases. This is of course related to the hyperbolic relation linking the bandwidth and the transmission time. We saw that in the case of the 64Mbits/s bandwidth, the computation penalty is quite low making further memory bandwidth increase not that interesting. However decreasing that bandwidth too much quickly begins to have a significant impact on performance beyond that value.

Looking at the computation time dependency with memory access latency, we can observe that it is similar for all the bandwidth and quite small (a maximum penalty of 1.3% compared to a 0-latency memory in the 64Mbits/s bandwidth case). This result is again easy to explain: latency only delays the data emission/reception of a fixed time and is thus independent of the data

size contrarily to the bandwidth. In this precise case the computation time penalty is very small even for a high latency of 10ms, meaning that slow memories to access are still suited in the case of our current application/platform combination.
As we can see from this experiment, it is very easy to explore the impact of one degree of freedom inside Nessie: all we need to do is to change the value of the required degrees of freedom, tell to the framework which variables should be represented on the graph and run the simulation. In a matter of seconds, it is thus possible to obtain a completely new result from the previously defined simulation with very changes in the input file.

Single computation node with memory and interconnect

In the previous example, the computing node was directly connected to the memory which allowed us to focus on the memorization interaction. In this case, we would like to examine the impact of a bus transmitting the data on the performance criteria: therefore we insert between the existing computation and memorization node an instance of a newly defined primitive $Pt_{1,2}$ representing the interconnect.
To derive the performance criteria of this platform primitive, we used specific Yeti models relying on the following parameters whose value is given in Table.6.4:

- The wire length L_{wire} set to 1mm like the side length of the computing node
- The wire capacitance C_{wire} that is associated with a value of $1pF$ for a 1mm long global wire in 180nm. This value is perfectly consistent with our estimation in the chapter devoted to Yeti results (see Sec3.3.2) and with numbers taken from the literature[4]
- The wire frequency F_{wire} whose value impact will be explored in the following
- The number of wires N_{wire} set to 64 which matches the word width of the memory
- The voltage swing over the wire V_{wire} that we chose equal to 2.5 V
- The area of the wire is neglected since it will likely be routed above the two previous components
- The latency is also set to 0 since it will likely be negligible compared to memory access time

Based on these different parameters we can easily calculate the bandwidth B_{bus} defined by Eq.6.3.

$$BW_{bus} = F_{wire} * N_{wire} \tag{6.3}$$

Primitive	area(mm^2)	L_{wire}(mm)	Latency	$Freq_{wire}$(MHz)	N_{wire}	V_{wire}(V)
$Pt_{1,2}$	0	1	0	Variable	64	2.5

Table 6.4: Value of the platform primitive $Pt_{1,2}$ parameters representing the bus

Regarding the power consumed by the wire when it is transmitting data $P_{bus,wire}$ we use Eq.6.4 to make a worst case estimation[2].

$$P_{bus,wire} = F_{wire} * N_{wire} * C_{wire} * L_{wire} * V_{wire}^2 \tag{6.4}$$

Running a simulation with these parameters gives exactly the same results as the previous case: computation time penalty is simply null since we decided to neglect the latency and defined a total wire bandwidth value equals to the memory bandwidth. Furthermore if we calculate the resulting switching power when the wire works at 1MHz, we obtain a tiny value of 0.4μW which makes bus power highly negligible compared to the static or dynamic power consumption of the two other blocks.
However this case is interesting to see how Nessie performs the effective bandwidth calculation for the data token transmission rate. To illustrate that, we have plotted the total computation time versus the bus bandwidth for different output data token sizes of $Fc_{1,0}$ and did the same again while setting the bus bandwidth to a very high value (over 1Gbits/s)[3]. The resulting graph (plotted in Fig.6.4) shows us three sets of superposed curves that diverge beyond a wire bandwidth of 64Mbits/s. This effect results from the minimum wire bandwidth matching performed inside Nessie: the data token will be sent at a rate that is equal to the smallest bandwidth of all the encountered elements bandwidths along the path. In our case, this mechanism prevents the wire from sending the data faster than the memory is able to store: this explains that the curves become flat when the memory bandwidth of 64 Mbits/s is exceeded. This performance limitation is not really problematic since we are already close to the lowest computation time value reachable in the single computation node case: the difference between the red and the black curve remains very small after the bandwidth threshold value. However in the case of bigger data tokens, the difference between the two curves grows more past that 64Mbits/s value offering some potential optimization for the global computation time: memory bandwidth progressively becomes a bottleneck. We may also notice by looking at the different set of curves that the performance gain for a fixed bus bandwidth increase becomes more important as the data token size grows. One

[2]This model supposes that each wire capacitance of the bus will be toggled each time a chunk of 64 bits is sent over the bus. The model of Eq.6.4 is thus often multiplied by an additional factor representing the average activity factor when transmitting data, factor that is around 0.5 in practice but that we implicitly consider equal to 1 to take the worst case.

[3]At the moment, if it is possible to disable the effect of latency by setting its value to 0, Nessie doesn't allow the user to define infinite bandwidth. Instead we simply define bandwidths values so that the transmission time of a data token becomes negligible compared to other operations.

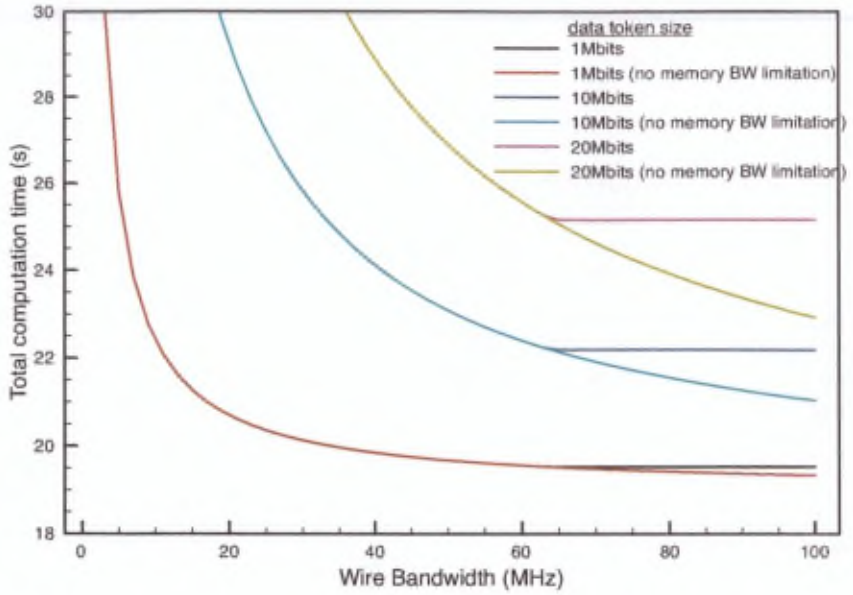

Figure 6.4: Illustration of the minimum bandwidth matching mechanism performed inside Nessie

last observation that can be made on this graph is that the computation time grows very fast for small bandwidths: this is nothing more than a visualization of our consideration on bandwidth hyperbolic relation with data transmission time made in the previous architectural case.

Summary

Through these different examples, we tried to illustrate the basic features of Nessie on a simple hypothetic case that we progressively made more complex. When the input simulation files have been defined, it is only a matter of seconds to modify the degrees of freedom values so that other experiments can easily be performed for another set of values. To get an approximate idea of the input simulation files complexity, it approximately required one hour to get all the input files defined and initialized for the different examples presented in this section.

Now that we have examined how Nessie performs for a single node, it would be interesting to know how it is able to exploit the parallelism offered by the application: the next section will be devoted to the study of multiple computation nodes cases.

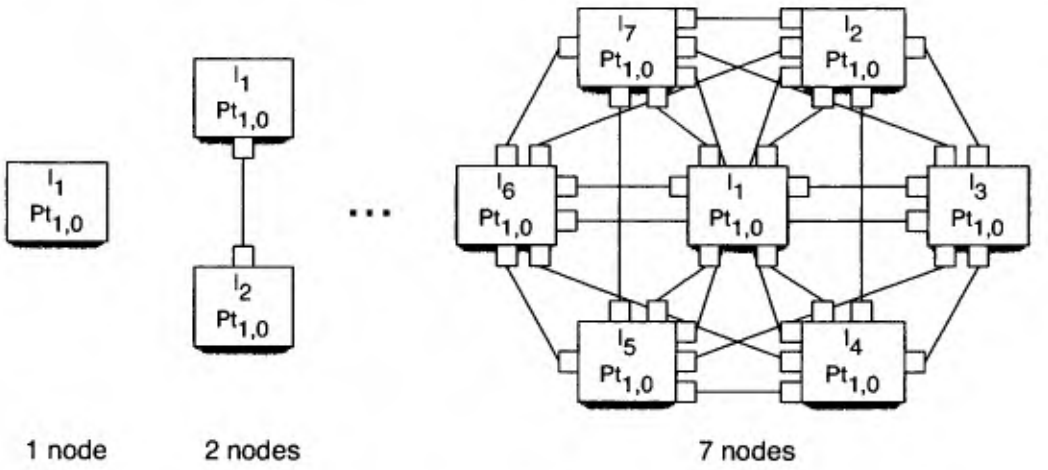

Figure 6.5: The seven fully connected homogeneous architectures competing for the best mapping of functionality $Fc_{0,0}$

6.2.3 Multiple computing nodes architecture

In this section we will quantify the impact of multiple computation nodes architectures on performances and examine the trade-off between computing time and consumed energy. All results were obtained using the same application (functional structure) as described in the previous section and all functional/-platform parameters values are kept the same unless explicitly mentioned.

Improving computation time

The first interesting experiment that could be performed is to evaluate how the parallelism offered by a platform can be used to speed up the execution of our application described by Fig.6.1. Therefore we will define several competing architectures with a growing number of computation nodes and see how they convert this additional platform parallelism into smaller computation times. As depicted in Fig.6.5, we have built seven architectures ranging from 1 to 7 computing nodes derived from platform primitive $Pt_{1,0}$: the communication architecture is fully connected meaning that there is always one path (bidirectional virtual link) between each pair of block.
From the Nessie point of view, we define seven different platform structures for the primitive $Pt_{0,0}$ and map the application for the seven values of this new structural platform based degree of freedom. The resulting graph is plotted in Fig.6.6: we can see that the computation time decreases with the number of computing nodes meaning that Nessie indeed converts these additional computation nodes into faster application execution. However this improvement is

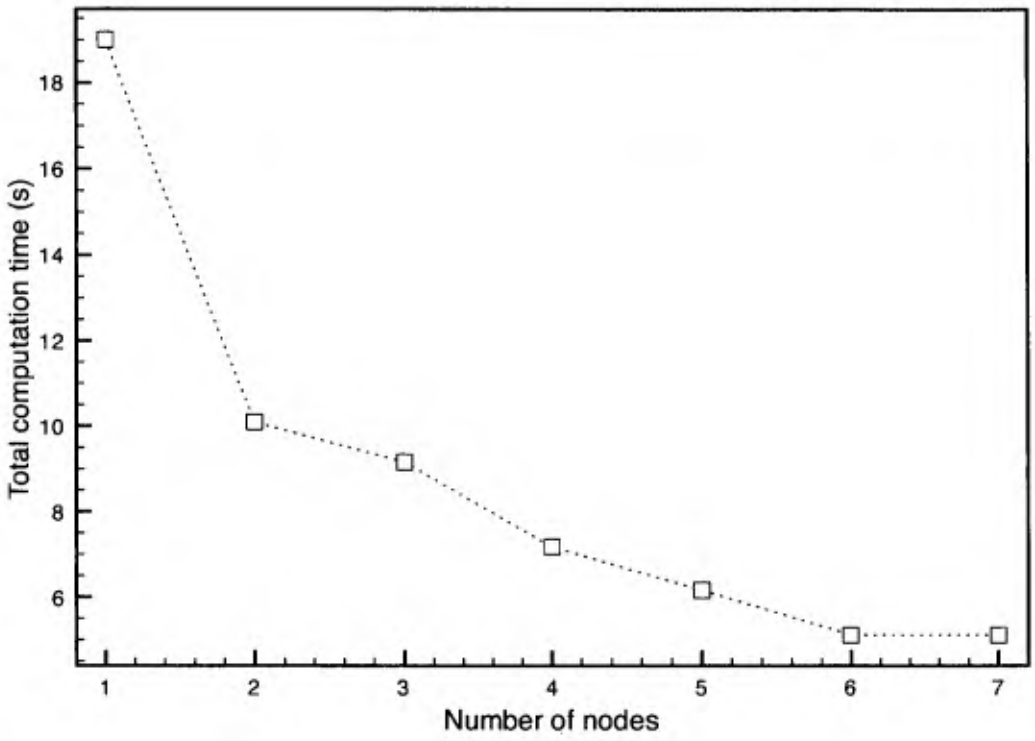

Figure 6.6: Evolution of the total computation time with the number of platform blocks composing a fully connected architecture

far from being linear: while two computation nodes bring the execution time from 19s down to 10.1s, the gain from additional blocks beyond that value tends to decrease until the addition of a seventh node doesn't improve the computation time anymore (still requiring 5.1s). This can be justified quite easily looking at the application described by the Petri network: indeed at any possible stage of the application execution, we have only a maximum of six functional blocks that can be processed at the same time (parallel places 2-7 and 8-13). Therefore our application doesn't benefit from the addition of a seventh node that is never used due to the limited available parallelism: the execution time thus doesn't decrease after the sixth computing node. However the reason for the two nodes architecture to perform so well compared to the single node solution is that there is almost all the time a set of two potential operations that can be simultaneously executed. The efficiency of the different architectures can be confirmed and further investigated by having a look at the activity reports[4] generated during the previous simulation at the same time as the plot and timeline files.

In our case we have used these activity reports information to represent the cumulated relative time spent in the computing state: this metrics results in something very close to the speedup factor defined in Amdhal's law (see Sec.2.2.6). In other words it quantifies how much the application benefits from more parallel platform blocks:. a.cumulated-relative-activity-value-of-N-for-a-N-nodes architecture means that each of the platform block is computing 100% of the time. Looking at the graph, we can see that the cumulated relative computation time increases with the number of nodes until six where it keeps the same value: the seventh node is in idle mode 100% of the time so that it doesn't improve the total computation time at all. The 2-nodes architecture is however far more efficient: node 1 and node 2 compute respectively 99% and 89% of the time leading to an effective conversion of the platform parallelism into decreased computation time. Furthermore this graph shows us that when the number of nodes grows, the average efficiency of each node decreases and the workload is split among them so that the total execution time keeps on decreasing after the addition of a second node but at a slower pace. Interesting to mention is the fact that, inside each architecture, the computing activity always decreases for platform blocks with growing ID values. This reflects the way Nessie proceeds to allocation using the default policy: a functional block is mapped onto the first compatible platform block that is found among the pool of available blocks. For instance, block 1 relative computing activity is always close to 1 since it is the first candidate in line for allocation.

If the previous analysis is interesting, it focused only the sole computation time aspect neglecting other possible performance criteria like energy. Let us now evaluate and discuss this compromise between consumed energy and

[4] As a reminder the activity report describes for each node of the architecture the absolute and relative time spent in each state (more details can be found in Sec.5.5.2).

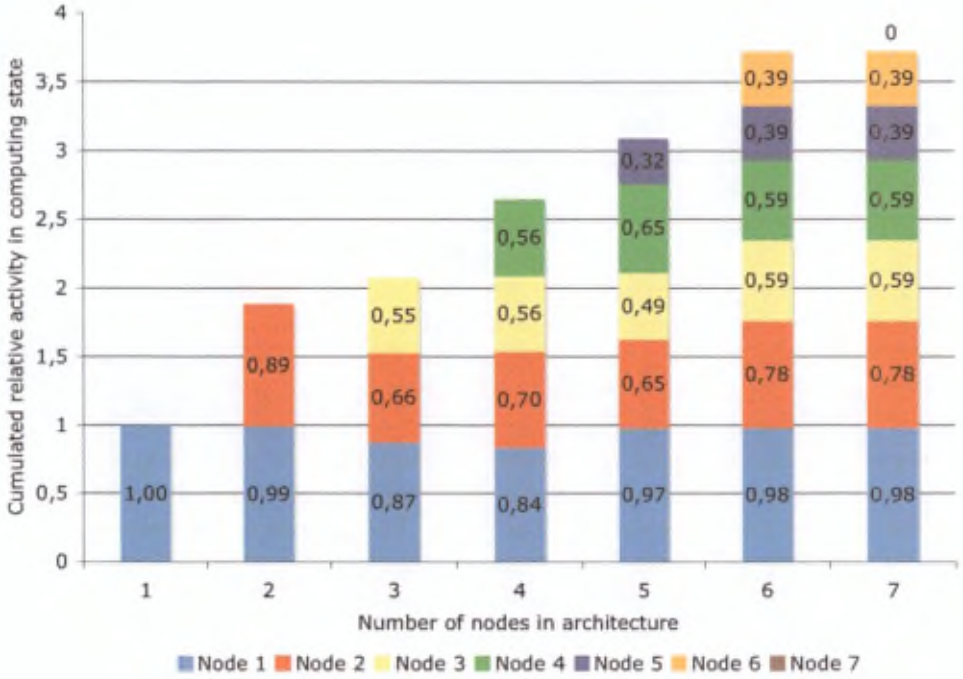

Figure 6.7: Cumulated relative computing activity for architectures with a growing number of computing nodes

computation time.

Energy VS computation time

Now that we have set the simulation parameters values in our previous experiment, we just need to change the input and output parameters of the plot result file to express the total computation time versus the energy. The graph representing this compromise is depicted in Fig.6.8 where each point is tagged with its corresponding architecture[5]. Based on this graph, we can draw very different conclusions compared to the previous ones: if it always good in terms of computation time to increase the number of computation nodes, it is certainly not when we take energy into account. However more nodes doesn't necessarily imply more energy consumed to complete the application execution: the 5-nodes architecture is for instance simultaneously less energy efficient and takes more time to execute the application than the 6-nodes architecture. In fact the energy efficiency simultaneously depends on the execution time and activity of the different platform blocks: the longer they stay in idle state, the more static power they draw increasing in turn the total energy required to complete the execution. This wasted energy thus grows with the number of blocks in idle mode and the proportion of the time spent in this state. Taking back our cumulated relative computing activity graph of Fig.6.7, it becomes thus clear that architectures 5 and 6 score much worse than architecture 2 in terms of energy efficiency since their mean activity is much lower for a higher number of nodes.
Looking back at our time-energy graphs, we can easily find the pareto optimal solutions by discarding all the dominated solutions[6]: architecture 1, 2, 3, 4 and 6 thus remain. Depending on the constraints and requirements, one of these solutions could be selected by the designer as the solution to further implement. The architecture composed out of 7 nodes has the same execution time as architecture 6 for an increased energy since the seventh node is never used due to limited parallelism of the functionality and is only consuming static power 100% of the time. We could be tempted to prefer the 2 nodes solutions to the 1 node solution since a small increase in energy almost divides the computation time by a factor 2 but again this is only true if we don't take the area criterion into account for which architecture 1 performs better.

[5]Contrarily to parameters, degrees of freedom that represent functional/platform choices have a limited set of values since they represent different structures: plotting the solutions as a continuous curve thus makes little sense. However this can be useful to easily distinguish the different estimated points (in our case based on the number of nodes of the architecture) and to describe the shape of the solutions envelope to see how the points evolve with a given parameter. To emphasize the fact that intermediate values have no meaning, we will thus use in the rest of this work dashed lines to link this type of solutions.

[6]A solution is said to be dominated if there is at least one other solution that has all criteria scoring better than the latter. This solution is thus of no interest (at least considering the examined criteria) and is always discarded.

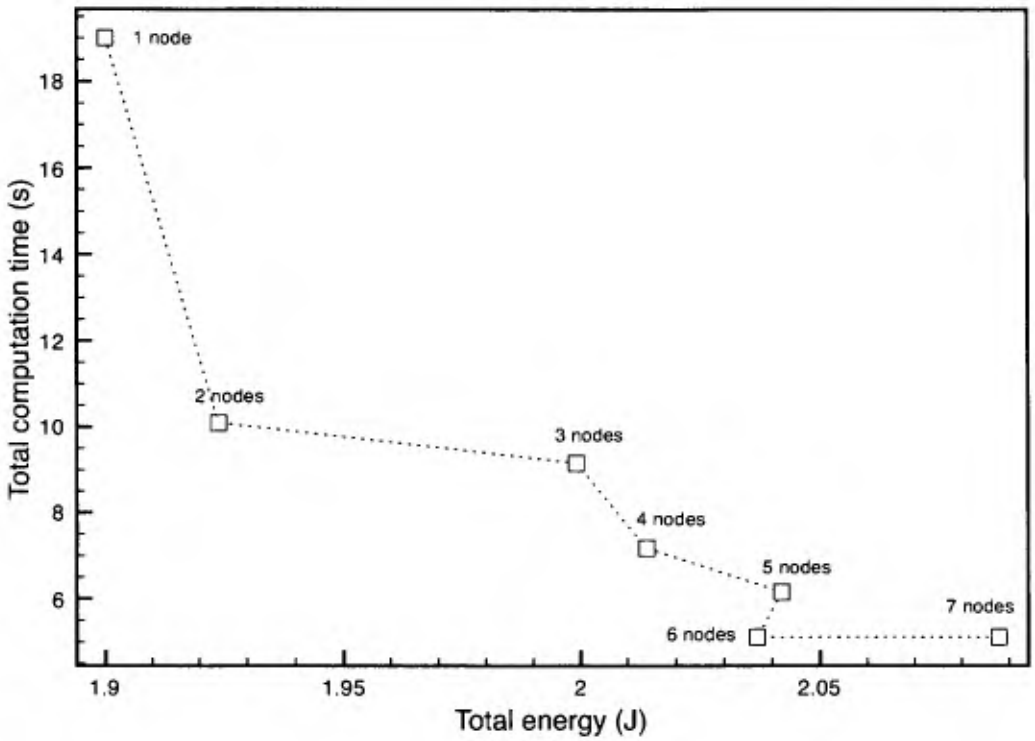

Figure 6.8: Computation time versus energy for architectures with a growing number of computation nodes

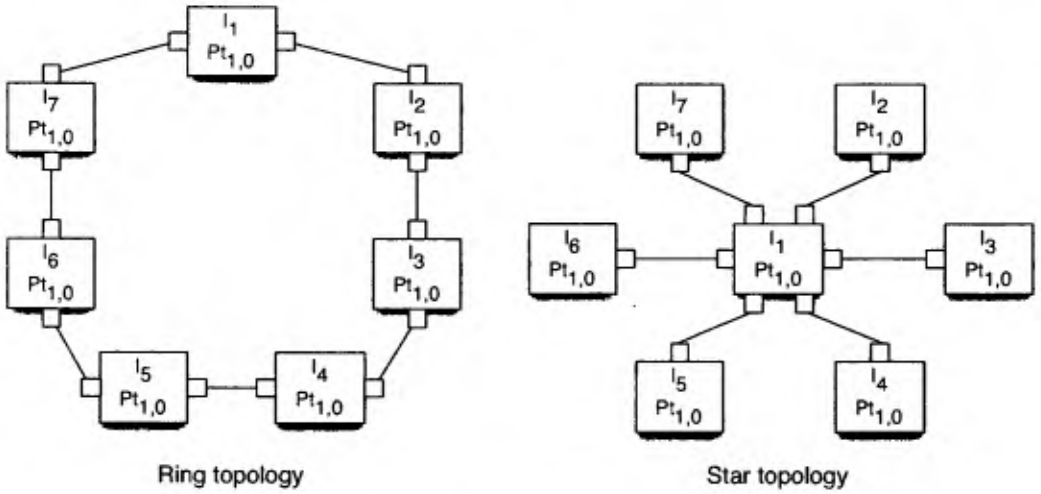

Figure 6.9: Ring and Star topologies for 7-nodes architectures

Topology	Fully connected	Star	Ring
Computation time (s)	5.11	8.41	9.97
Energy (J)	2.04	3.01	3.43

Table 6.5: Energy and computation time for the mapping of a functionality on six nodes based platforms with different topologies

Alternative topologies

While sweeping the number of computation nodes present in the architecture, we only used fully connected communication network topologies: in this section we measure the impact of other topologies on performances. We therefore choose a ring and star topology (both depicted in Fig.6.9) to compare with our fully connected topology. Simply initializing Nessie with these new topologies however provides the user with two invalid solutions meaning that Nessie was not able to find a suitable solution to the problem. This is explained by the fact that, contrarily to the fully connected topology, ring and stars topologies don't have connections between all pairs of nodes: since the $Pt_{1,0}$ block doesn't have any interconnect capability, data tokens are only able to flow from one block to a directly connected one. This restriction combined with the greedy allocation policy of Nessie leads to deadlocks as explained in Sec.5.4.3. To avoid these problems and enable Nessie to be able to find valid solutions, we added the communication state defined in $Pt_{1,2}$ to $Pt_{1,0}$ to allow data tokens to flow through the different blocks. The valid solutions are represented in Table6.5.

As we can see, the penalty from using these topologies in terms of energy and computation time is far from being negligible. This decrease in performance can be explained by the fact that some computation blocks are sometimes ready to execute but lack a free path to receive the data token required to begin the execution. The situation is worse for the ring topology: when a data token needs to be sent to all blocks, it has to be separately sent to all the blocks while, in the other case, it can be sent once and for all to all the blocks from the centre node: this explains the difference in terms of execution time.

Static/dynamic effect on total energy

As part of the sensitivity analysis of Nessie results to input parameters changes, we have performed an additional experiment to evaluate how energy evolves with the static/dynamic power ratio. To this end we performed the mapping of the functionality $Fc_{0,0}$ on the different candidate platforms while making the static power vary from 0.1% to 30% of $Pt_{1,0}$ dynamic power: the resulting plot is represented in Fig.6.10.
As we can see, the total energy scales very differently with increasing static power from one architecture to another. While the 2-nodes architecture reacts with a very slow global energy increase, the larger architectures see their global energy scale much worse. This difference in the slope of these different lines can be explained by taking back our previous activity graph of Fig.6.7 combined to the total computation time of each architecture: the cumulated absolute time spent in idle state grows with the number of nodes. Increasing the static power will thus have a bigger impact on the total energy for larger architectures.
That said, the mapping is still sufficiently efficient in our precise case to prevent the static power from dominating (a 30% static/dynamic power leads in the worst case of the 6-nodes architecture to a 18% increase in energy compared to a zero static power situation). However we can see that dynamic/static power proportion however has some impact on the choice of the architecture: for instance, the 6 nodes architecture is very interesting for static/dynamic values under 4% while it becomes the most power hungry over 15%. Finally we can also notice that the different architectures tend to energy values very close to 1.9J for small values of the static power: this is because the total dynamic energy is constant for any architecture. The small difference for static power values close to 0 is only due to different memory and communication activation patterns for the different architectures.

6.3 Modeling an H.264/AVC application inside Nessie

6.3.1 Introduction

While the previous case study demonstrated the different features of Nessie on basic examples, this section focuses on the modeling of a system that has

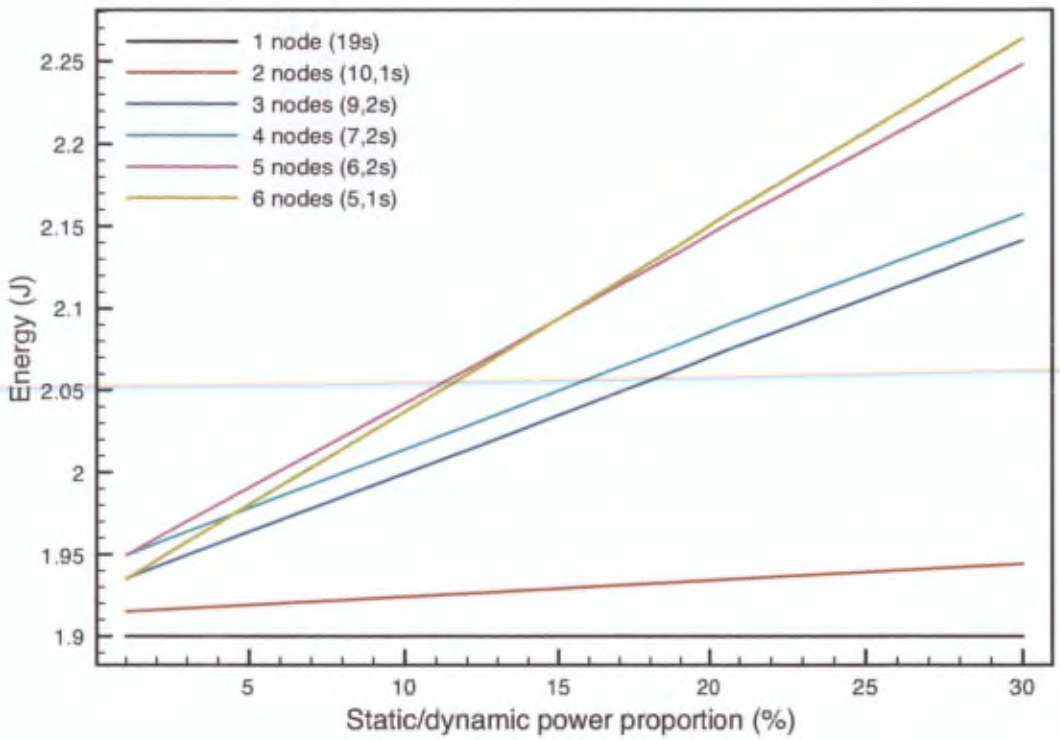

Figure 6.10: Energy consumed versus the static/dynamic proportion

already been studied and whose performances have been estimated and used as a reference for comparison with our framework. This case study relies on a work done by Dragomir Milojevic and published in[5], a reference that will be used as the main source of data for our study. This case study has been entirely performed by a student under our supervision: with no particular knowledge in microelectronics, he learned to use Nessie and Yeti, modeled and integrated the system of this case study into Nessie and finally performed the different experiments that are presented in this section in only three weeks of work. This gives an idea of the time required to get into grips with our tools and produce significant results.

The basic idea of this paper was to evaluate the power dissipated in the different components of a network-on-chip (interconnect links, network interface units (NIU) and switches) for a realistic traffic pattern generated by a multimedia application and compare it to the overall power consumption of the platform to prove network-on-chip efficiency in terms of energy. Different mapping scenario of the application on the platform have also been studied to measure the impact on the total power consumption.

The aim of our case study is to see in which extent it is possible to model such a realistic system in Nessie and try to reproduce the results. Based on this modeling, we will then extend the previous results for an architecture layout based on 3D stacking[7] and evaluate the expected power reduction achieved through the mean wire length reduction.

First of all we will briefly define the functionality, the platform and their mapping.

6.3.2 Description of the system

Functionality

The functionality used in this case study is a simple profile[8] H.264/AVC real-time video decoding application[7] studied for three different resolutions (CIF, 4CIF and HDTV)[8]. Resolution has a significant impact on the data amount exchanged between the different decoding operations since it is directly proportional to the number of macroblocks[9] required to code the video stream. The higher the resolution, the more macroblocks per frame will be exchanged between the different operations leading to an absolute increase in data bandwidth for a fixed framerate.

[7] 3D stacking basically consists in stacking different active silicon layers to reduce the average wire length: more details will be given in Sec.6.3.4.

[8] H.264 offers a lot of different capabilities regarding the inter- and intra-prediction: *profiles* including packages of capabilities have therefore been defined to suit the needs of different types of applications. The simple profile is primarily used in the case of systems with heavy resources constraints and mobile applications: more information about profiles can be found in [6].

[9] Each frame of a video stream is divided into 16x16 pixels blocks called *macroblocks*.

Platform

The H.264/AVC decoding is performed on the 3MF MPSoC platform developed by Imec and designed to support different video coding standards (such as MPEG4, AVC, SVC)[9]. The architecture depicted in Fig.6.11 is composed out of thirteen nodes:

- The six computational nodes are instances of the ADRES processor[10] with onboard L1 data and instruction memory. This architecture is based on a VLIW processor connected to a matrix of reconfigurable functional units and is meant for low power, high computation performances applications.
- Two instruction memory nodes L2Is1 and L2Is2
- Two data memory nodes L2D1 and L2D2
- One FIFO memory that buffers the compressed and uncompressed data stream
- One external memory interface (EMIF) connecting one L3 off-chip memory to the system
- One ARM microprocessor responsible for the MPSoC control and the audio subsystem.

To interconnect the different nodes, an Arteris NoC[11] is used allowing different transaction based communication protocols to be used. A separated communication architecture is implemented for data and for instructions as illustrated in Fig.6.11:

- The data network-on-chip uses a fully connected NoC with a 2x2 mesh topology to minimize the latency so that a maximum of 2 hops is required to reach a destination node. Each of the four switches is connected to a specific cluster of nodes (two clusters of ADRES, one containing the data memories and the last one to the FIFO).
- The instruction network-on-chip is only made out of one switch connecting the different instruction memories to the six ADRES nodes so that any communication takes one single hop. Additionally it is connected to the data network-on-chip to enable the communication between instruction memories and any other node of the architecture.

Each communication architecture is actually split into two different network-on-chips (see Fig.6.11): one request NoC carrying the data payload and a response NoC used for acknowledgement only. Finally arbitration is based on a Round-Robin algorithm and the routing is performed statically at design time so that only route between initator/target[10] pair is allowed.

[10]The initiator is simply the source node of the data while the target is the destination node for that same data.

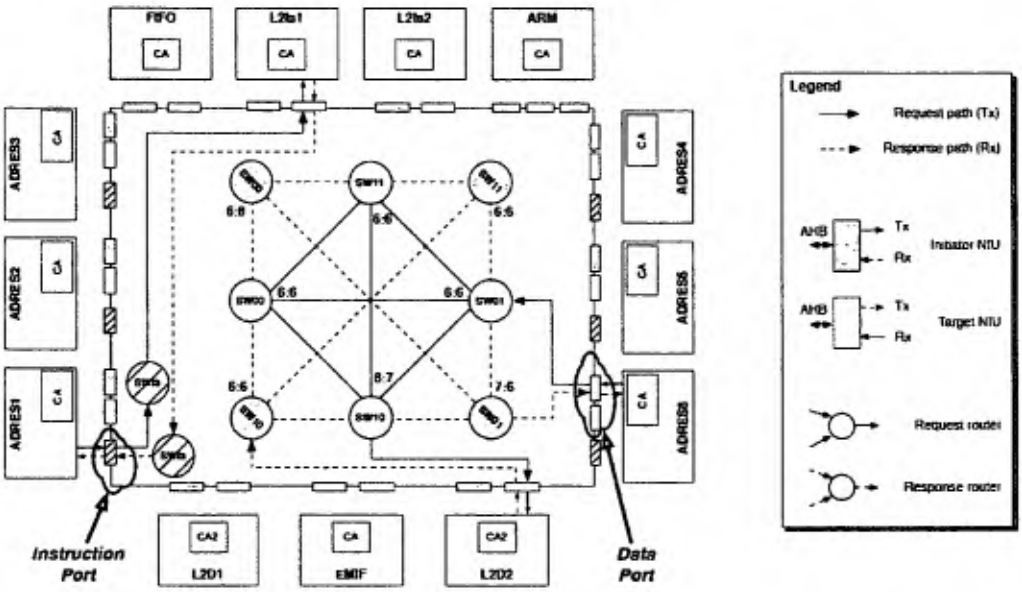

Figure 6.11: The 3MF platform for multi-standard video decoding based on six ADRES computation nodes[9]

Original mapping

As explained in the original paper, the functional block diagram is depicted in Fig.6.12 where the different operations (memorization and computation) are represented with the amount of exchanged data. The units of these data are expressed in byte per macroblock so that they represent the amount of data exchanged between the different operations for one simple macroblock.
Many possibilities exist to map the encoding application on the 3MF platform but three specific *mapping scenario* have been selected and compared in [5]:

- The *data split* consists in dividing the video stream in six equal parts that will be processed in parallel by the six different ADRES nodes. This scenario puts the stress on the instruction NoC due to the encoder code size.
- The *functional split* strategy relies on the distribution of the different operations over the six ADRES cores. Three ADRES are dedicated to motion estimation (ME), one to intra prediction, DCT, IDCT and motion compensation, one to the computation of the deblocking filter and finally the last one to entropy encoding. Compared to the data split, the functional split is more friendly with the instruction NoC but doubles the amount of data flowing on the data NoC.
- The *hybrid* scenario is a compromise between the previous solutions: the heaviest computational task (the motion estimation) is mapped onto three ADRES nodes using a data split while the remaining tasks are mapped onto the three other ADRES nodes according to the functional split scenario.

For each mapping, the paper contains a table with the data throughput of the different platform nodes communicating to execute the application described by the functional diagram of Fig.6.12.

Power dissipation modeling

To estimate the power dissipation of the complete network-on-chip, we model the individual power consumption of each element that the NoC is composed of. Basically power consumption can be separated into a dynamic P_{dyn} and an idle P_{idle} (when the element is not transmitting data) contribution as described by Eq.6.5.

$$P_{tot} = P_{idle} + P_{dyn} \tag{6.5}$$

The switch and NIU power consumption are respectively modeled by Eq.6.6 and Eq.6.7 where c_i are experimentally determined coefficients representing the idle and dynamic power consumption. The A term of the equation represents the activity of the element, in other words the percentage of time during which the element is active and transmitting data.

$$P_{switch} = c_1 + A * c_2 \tag{6.6}$$

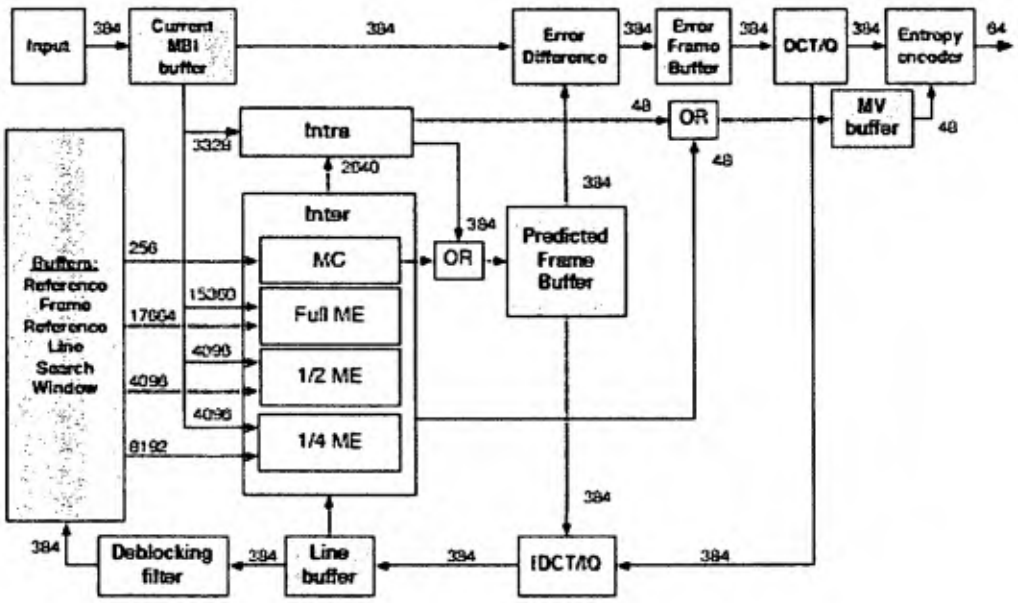

Figure 6.12: Functional block diagram for the H.264/AVC application including the different memorization and computation operations with their data transfers (in bytes per macroblock)[5]

$$P_{NIU} = c_3 + A * c_4 \tag{6.7}$$

In the Arteris NoC, each link is composed out of several *fixed length* segments, each segment containing a certain number of wires with their associated repeaters. The link dynamic power consumption P_l is thus the product of the segment power consumption P_s by the length of the link l as described by Eq.6.8. The contribution of the segment to the power dissipation involves the number of wires ω, the NoC frequency f_{NoC}, the capacitance per segment unit C (including the contribution from both wires and repeaters), the activity of the wire A, the supply voltage V_{dd} and the average toggle probability of a wire p_{toggle}.

$$P_l = l * P_s = l * \omega * C * A * f_{NoC} * V_{dd}^2 * p_{toggle} \tag{6.8}$$

To feed these models with input values, a three step methodology was used in the original paper:

1 The *Arteris NoC explorer tool* is used to generate traffic patterns based on the three mapping scenarios described earlier.

2 Based on these traffic traces, *Synopsis VCS* is used to perform a functional RTL simulation that will result in SAIF files (Switching Activity Interchange Format). These files contain all the detailed activity of the different components that will be required to compute the power consumption.

3 Based on the SAIF, the power analysis is performed using the *Magma Blast Power* tool operating at a gate-level.

The different numerical values for the coefficients of the power model of the link, switch and NIU resulting from these experiments can be found in the original paper. To compute the total power consumption of the NoC, all the initiator-target consumptions are summed up (results for the different mapping scenarios are detailed in the paper).

6.3.3 Formalization of the problem

Introduction

Now that we have completely defined the system, we are ready to discuss how it can be translated into the formalism accepted by Nessie. We will then use this model to calculate the power consumption using Nessie and see if these estimations match the results of the original paper.
From the description that was made in the previous section, we can already make several straight-forward considerations about the general methodology that will be used to integrate all these data inside Nessie.

Abstraction hierarchy In the previous section, we have made a structural description of the functionality and the platform with their mapping. This

means that this problem will be translated into two abstraction levels inside Nessie: application/chip will both be described by structures based on functional/platform primitives of the abstraction level just below. To evaluate the performance criteria of the different functional/platform blocks pairs determined by the mapping we will use the different power consumption models described in the paper and integrate them inside Yeti.

Mapping In the original case study, three different mapping scenarios (the data split, the functional split and the hybrid scenario) have been tested, each one defining which operation (a functional block from the Nessie point of view) is supposed to be mapped onto which platform node (a platform block). However Nessie is a tool meant for dynamic and automatic mapping in the context of HW/SW co-design and assumes that both functionality and platform are described separately. In the original case study, the mapping has however been performed manually and the functionality is not described independently from the platform. To make the problem description compatible with Nessie formalism we will thus have to "unmap" the functionality from the platform and successively associate each functional block with one platform block to properly describe the mapping.

Performance criteria The original paper computed the power consumption of each individual element based on its activity factor and summed them up to get the total power consumption of the NoC. Nessie however performs an online event-based mapping that will estimate the activation order and time of each operation mapped onto the platform. To calculate the power consumption, we will thus define an additional criterion representing the energy consumed by each block and integrate that value over time to get the total power consumption. Additionally we will also estimate the area of the platform based on the individual area of each block. This gives us a total of three performance criteria:

1 *Computation time* : one mandatory criterion in Nessie used to derive the value of all the other criteria

2 *Energy*: an integrate time criterion that is additive over the surface

3 *Area*: a time-independent criterion that is additive over the surface

Platform and associated Yeti models

To represent the 3MF platform inside Nessie, we have modeled each separate element (switches, links and computation nodes) as a platform block. As we can see in Fig.6.13 the connection topology between the different elements is the same as the original structure presented earlier in Fig.6.11. The sole difference is that we only included the request NoC carrying the data payload and not the response network that is much less used and for which we don't have sufficient

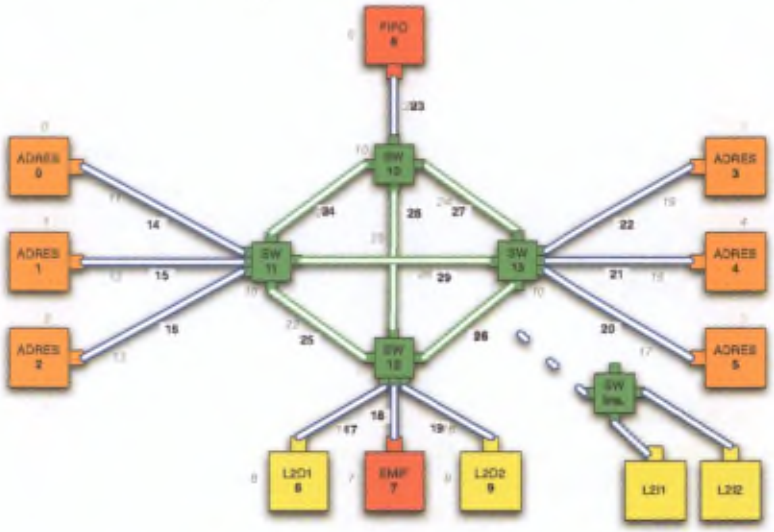

Figure 6.13: Platform structure used inside Nessie for the representation of the 3MF platform

information to establish a correct model. Since the request network carries much more data than the respond network, neglecting the latest will have a very small impact on the overall power consumption.

Each platform block (memories, ADRES nodes, switches and links) derives from a given platform primitive associated with a model to estimate performance criteria (time, energy and area). *Energy* is however a particular criterion because it is the sum of three different criteria: the NIU energy, the wire energy and the switch energy. By separating the contribution of these different elements, we are able to establish a more detailed analysis of the power consumption budget.

For each platform primitive, we define two different states with different values for the criteria: an idle state and a memorization/transmission state (depending on the block primitive nature: a memory or a link/switch). The time spent in the memorization/transmission state divided by the total computation time corresponds to the activity factor A defined in the previous equations of the power consumptions Eq.6.7 and Eq.6.6. Nessie will change the states of the different platform blocks according to the performed mapping so that it will reproduce their respective activity factors.

The different parameters used as inputs for the Yeti models are those previously defined by power consumption equations (except the activity factor A). We tried as much as possible to reduce the number of parameters ending up as degrees of freedom by making a clever use of the parameter local/global definition (see Sec.5.1.4). Each parameter that needs to have a single value for

the whole chip is defined as a global parameter: this is the case for the supply voltage, the toggle probability p_{toggle} etc. Using this technique, the value will only appear once in the degrees of freedom so that changing this value will affect all the models that use this parameter. On the contrary, some parameters like the wire length take values that are dependent of the instantiated platform block: therefore we have assigned this parameter with local values so that each link can be associated with its own length. Other combinations are also possible to group several links together in order to define their length by a unique degree of freedom value.

Application

In our problem, it is not possible to directly reuse the description of the functionality inside Nessie but we have all the necessary information to enable its transformation into the format required by Nessie. Indeed the functional diagram depicted in Fig.6.12 details the different operations, the amount of data exchanged and the memories towards which the data produced by the different nodes will be sent. Based on this diagram, we have manually extracted the data dependency to expose the maximum parallelism of the operations and convert this information into a petri network: Fig.6.14 and Fig.6.15 respectively depict the resulting petri nets representation for the data split and the functional split. Each operation requiring to send data is represented by a place (with the required data bandwidth) and the operations that consume these produced data are connected through a transition representing the data dependency. If these petri networks are fairly easy to interpret, we have to mention two particularities regarding the explicit memorization and the data split that will have an impact on the resulting petri network: both questions are further discussed in the following paragraphs.

Explicit memorization As explained in chapter 5 devoted to its complete description, Nessie is primarily meant for automatic mapping of a functionality on a platform. To perform this mapping, the communication and memorization needs are implicitly defined by the data dependencies of the petri network describing the functionality so that it should only contain computation operations. Based on this petri network, Nessie maps the different operations onto the platform, automatically transmits data tokens between platform nodes when required, memorizes data tokens inside the platform block that has produced it when memorization ability is available or transmits it otherwise to a distant memory.
In our case however, the mapping is completely fixed for each scenario and the data resulting from the different operations are assigned to specific memories so that Nessie is not allowed to choose them arbitrarily as it would have done normally. To force manual mapping, we thus have to make data memorization operations explicit in the petri network: this can be illustrated by transitions

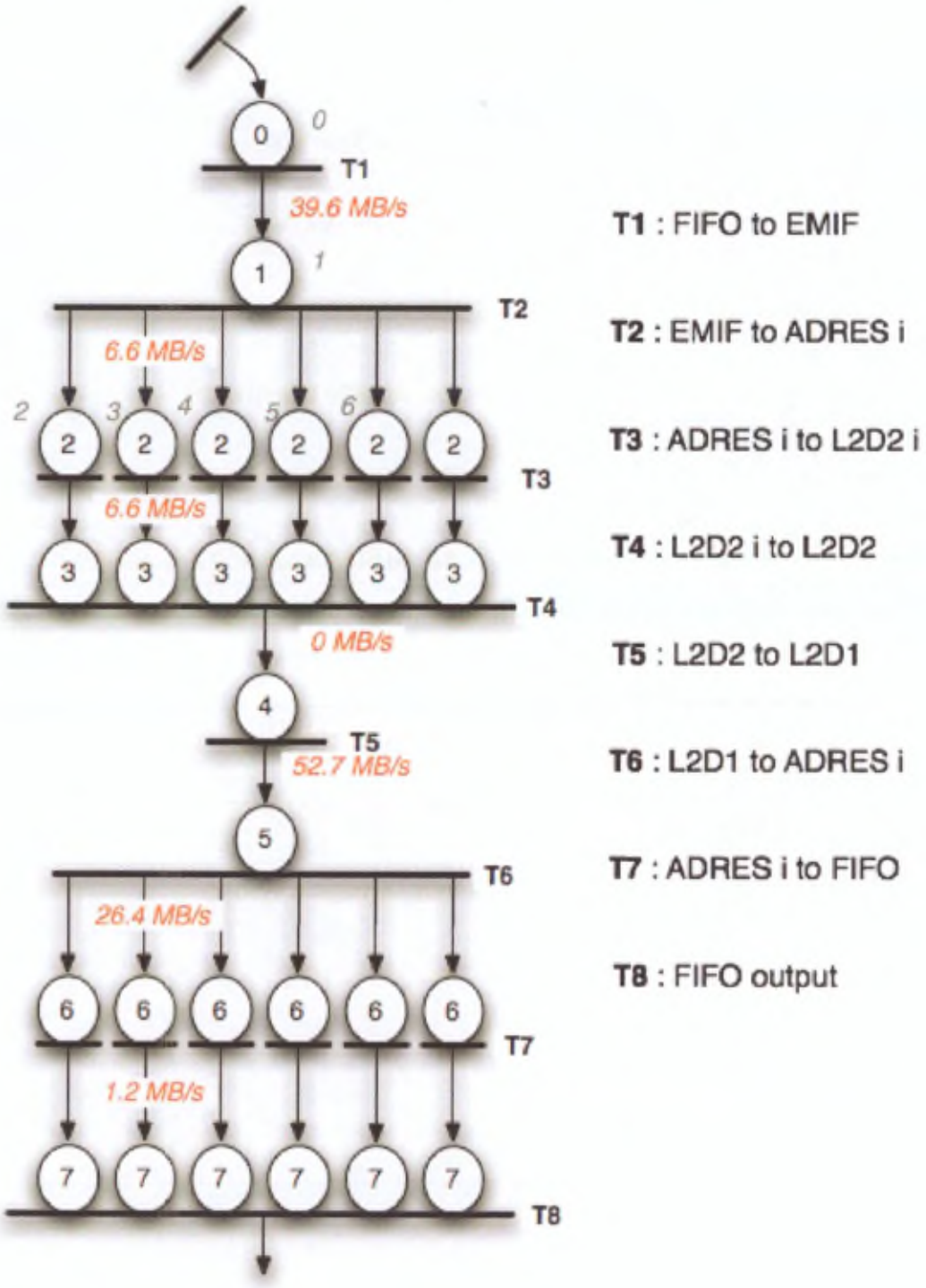

Figure 6.14: Description of the H.264/AVC functionality with a Petri network for the data split scenario

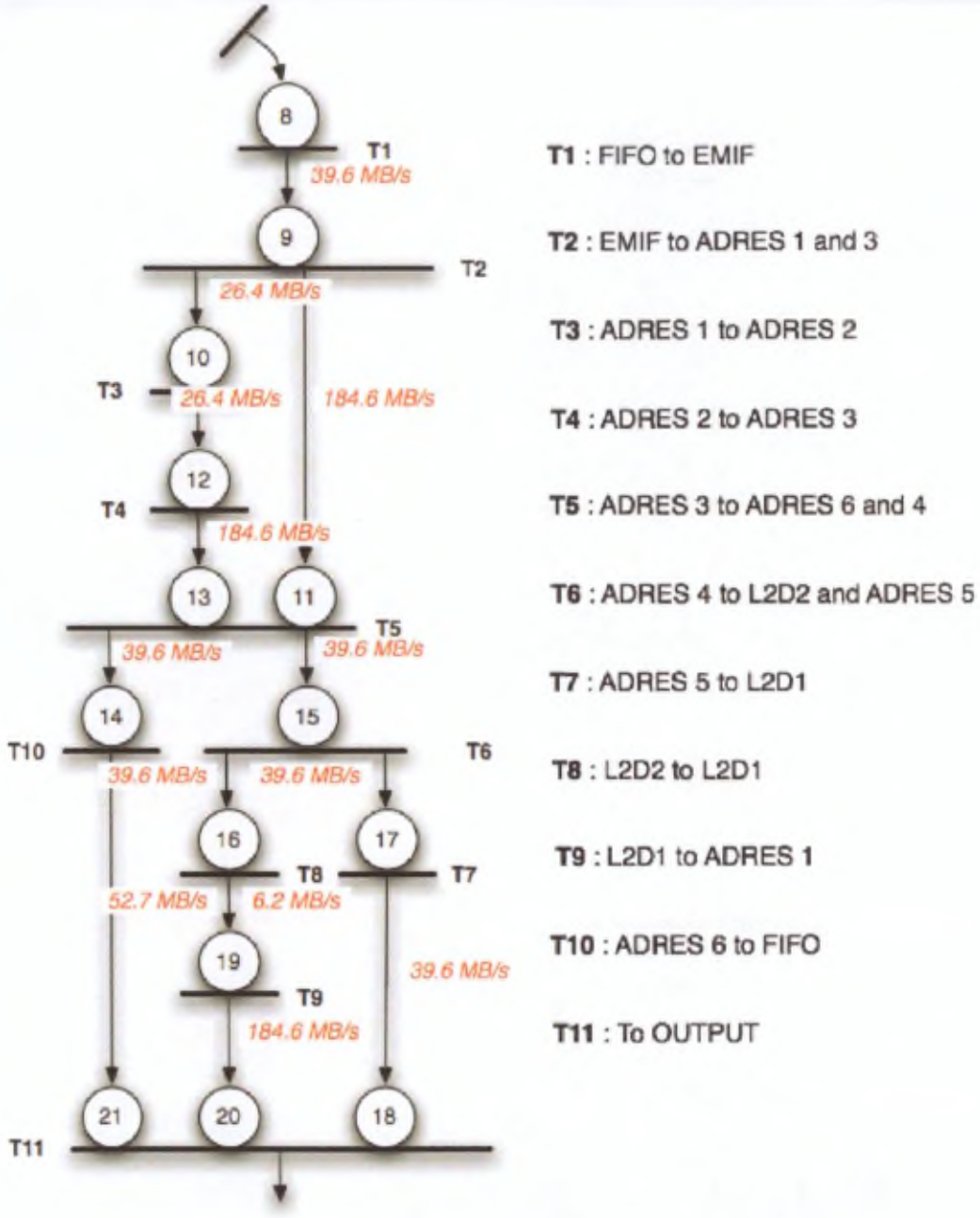

Figure 6.15: Description of the H.264/AVC functionality with a Petri network for the functional split scenario

T_1, T_2 and T_4 for the data split scenario of Fig.6.14 for instance. However this petri network gives a simplified view of what is really happening in order to prevent the figure from being overwhelmed with too many places. There are two distinct procedures depicted in Fig.6.16 that can be used to force the memorization (fetching) of a data into (from) a specific memory:

- *Data fetching* from a specific memory can be achieved by inserting a *dummy* place before the place representing the operation that needs to consume the stored data token (see left of Fig.6.16). Place P_1 contains a dummy operation (with zero execution time) what is only compatible[11] with the specific memory that should contain the data token required by place P_2. Due to this unique compatible platform block, Nessie will thus map the operation associated to P_1 onto this memory which will require a zero execution time and produce the data token that will immediately be memorized. Operation associated with P_2 that requires this token will thus tell Nessie to route the data token from the memory to the platform node where operation P_2 has been mapped. This mechanism can thus be used to define data tokens stored in a specific memory which can be useful for instance to set where first data are physically available on the platform for a functionality to start.
- *Data memorization* to a specific memory can be achieved by inserting a *dummy* place after the place representing the operation producing the data token to be stored (see right of Fig.6.16). Place P_4 contains a dummy operation (with zero execution time) which is only compatible with the specific memory that should store the data token produced by place P_3. Once place P_3 has produced its data token, Nessie will route and transmit it to the specific memory compatible with operation P_4. Because it is a dummy operation, P_4 will instantaneously produce a token of the same size as the token it received and memorize it. This mechanism is very useful to force the result of an operation to be stored in a specific memory used as interface with the outside of the system for instance.

Data split Looking at the functional block diagram in Fig.6.12 we can see that some parts of it ($EMIF$ to $ADRES_i$ for instance) require a data split. When mapped onto the platform, this means that the data coming out of the platform source block (the EMIF) are equally divided into six data chunks individually sent over to the different ADRES nodes. However such a data split cannot be represented in a straight-forward manner in our petri network because each place may only produce one single token. Enabling places to produce more than one token as a result of a functional block execution completion doesn't solve that issue since these tokens would be sent to all the places connected to the output transition anyway.

[11] As a remainder, a compatible list establishes the different platform primitives that can execute a

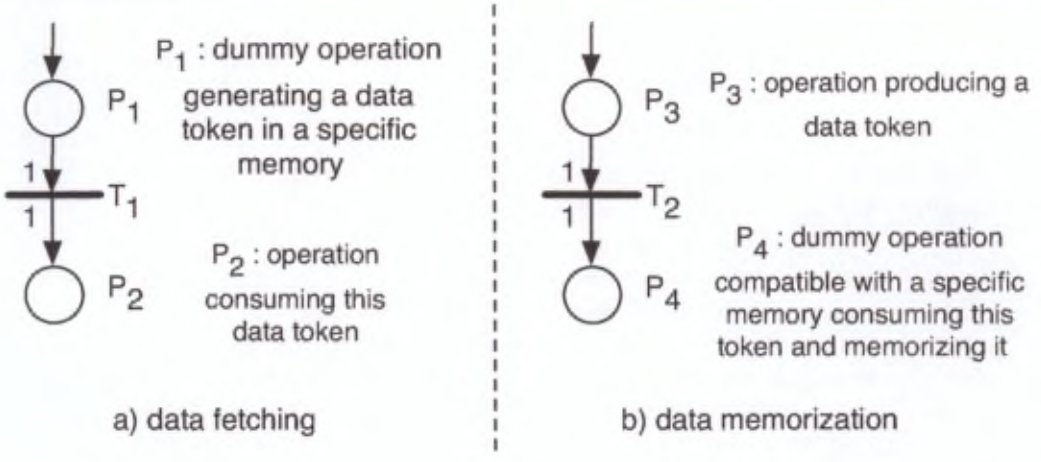

Figure 6.16: Modification of petri network for manual mapping: a) procedure to force data fetching from a specific memory and b) method to force the memorization of data inside a specific memory

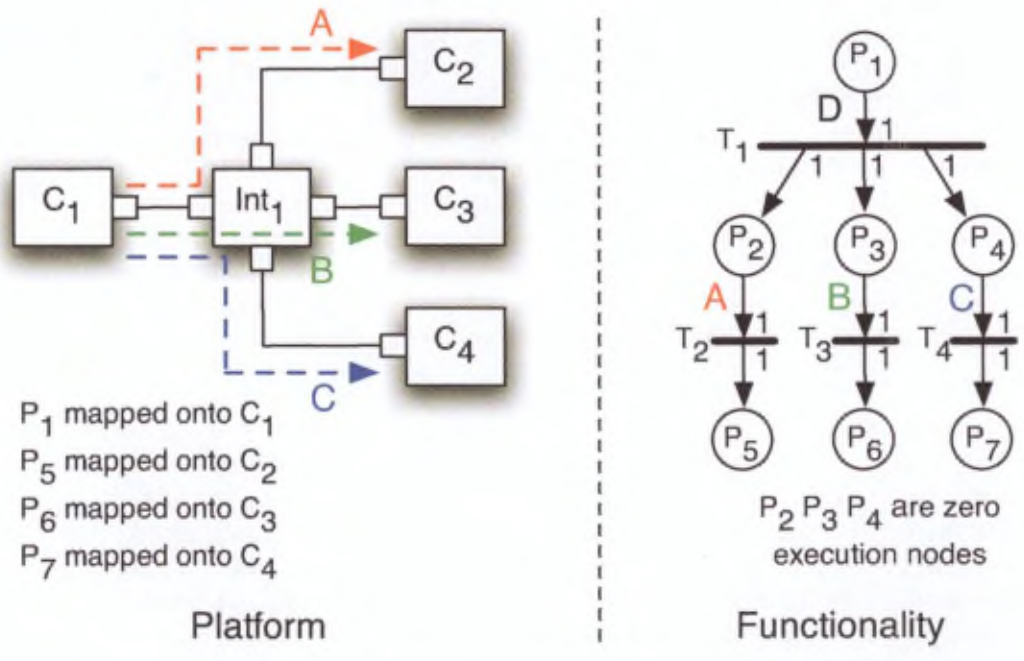

Figure 6.17: Modification of petri network to enable the functional description of data split

To overcome this limitation and enable a given token to be sent towards a specific place, we can make use of *zero-execution places*. Fig.6.17 presents a case where we would like a place P_1 to send one individual token of size A, B and C respectively to places P_5, P_6 and P_7. To do so, three additional places P_2, P_3, P_4 have been inserted between the original places and the destinations: these new places are zero-execution places, all of them only being only compatible with platform block C_1. When C_1 ends the execution of the functional block related to place P_1, transition T_1 is fired and places P_2, P_3 and P_4 are ready for execution. Since these three places relate to zero-execution operations compatible with platform node C_1, they will consume data token D and execute instantaneously on C_1: as a result, three tokens A, B and C are generated and stored in C_1. As each place P_2, P_3 and P_4 are connected to individual transitions T_2, T_3 and T_4, each token will be sent individually to the platform block onto which the functional block related to one of those three next places P_5, P_6 and P_7 will be mapped (respectively C_2, C_3 and C_4 in our case).
This technique has the advantage of being quite simple, doesn't change the execution time calculation and is sufficiently flexible to solve any issue regarding the number of generated tokens and the functional blocks that require to use it. Indeed we just need to insert as much *zero-execution* places as required tokens and link them to transitions connected to all the places requiring each individual token. We have used this mechanism as often as it was necessary to describe data split in our AVC functionality inside the petri network formalization.

Mapping

As we have already explained earlier, the mapping operation is reduced to its simple expression in our case since we went for a manual allocation of the functional blocks on the platform blocks to represent the case study as accurately as possible. Some important remarks are however to be made to detail the conditions required for Nessie mapping to be relevant in our AVC case study.

Explicit memorization To perform the manual mapping according to original paper, each operation needs to be made explicit: not only computation operations are specified in the petri network based functional description but also data memorizations. This is required to explicitly define inside Nessie all the pairs of platform blocks that will exchange data which will then enable proper routing.

Routing In the original paper, routing is performed statically at design time with the objective of enabling a maximum number of nodes to communicate

---

particular functional primitive.

simultaneously. Only one path is allowed between any pair of nodes so that the traffic is entirely deterministic over the network-on-chip. In Nessie, routing is however performed online so that the mapping core will determine a route dynamically whenever a platform block needs to send data to another node. We could solve this problem using the same technique as the memory manual mapping: each platform block belonging to the route should be compatible with a dummy operation in order to force the data token to take a given path from the source to the destination. Adding these information to the petri network would however lead to a severe overhead in terms of description complexity and wouldn't make much sense. Furthermore Nessie always performs a greedy routing to simultaneously use as much routes as possible: the difference with the optimized static routing of the paper should thus be quite small in terms of wire activity.

6.3.4 Results

In order to validate the previous modeling of our AVC case study, we will first try to reproduce the results of the original paper and see to which extent Nessie can accurately predict the NoC power consumption. In a second step, we will extend the 3MF model to the case of a 3D stacked architecture and see if the average link length reduction can be efficiently converted into power consumption savings.

Results based on the original 2D architecture

This first experiment consists in estimating the NoC power consumption of the AVC application running on the 3MF platform thanks to their modeling inside Nessie (see Sec.6.3.3). The input values of the analytical models of Sec.6.3.2 have all been taken from the reference paper [9] except for the length of each link that has been estimated based on the 3MF floorplanning.
For the three scenario and three possible resolutions, Table 6.6 represents the original results for the total NoC consumption, the estimations made by Nessie based on the model and finally the relative difference between them. As we can see Nessie performance predictions for the total power consumption are very close to the original values with a mean absolute error of 0.77% when averaged over all the experiments. This error is only related to link approximations but not to NIU or switch since input parameters, activity (defined by the functional diagram) and models used to estimate their power consumption estimation do not differ from the original paper. Indeed if we estimate the power consumption contribution from links only (representing on average 8% of the total power), the mean error reaches 11.8% demonstrating that only links contribute to the total error. This error on wire power consumption estimation comes from two different sources:

- The dynamic routing perfomed inside Nessie may differ from the static

Resolution	Power	Data split	Functional split	Hybrid
HDTV	Original results [mW]	21.37	21.27	17.01
	Nessie results [mW]	21.19	21.19	16.87
	Relative error [%]	-0.82	-0.38	-0.80
4CIF	Original results [mW]	19.35	17.73	14.77
	Nessie results [mW]	19.15	17.91	14.92
	Relative error [%]	-1.05	1.02	0.98
CIF	Original results [mW]	18.16	15.64	13.46
	Nessie results [mW]	17.99	15.74	13.50
	Relative error [%]	-0.93	0.65	0.33

Table 6.6: Comparison of the original NoC power dissipation with Nessie estimations for the 2D case study

routing used in the original paper. The data tokens taking a different path will then change the total activity of the wires hence their power consumption

- Due to a lack of precise data, we used approximations of wire lengths which is directly proportional to the power consumption.

The combination of both these sources of error explains why we don't have the exact results and proves that Nessie is able to accurately estimate the performances of a system as long as we use precise models and data. Furthermore the error is small enough on average to reasonably extend this model to the case of a 3MF architecture implemented on a 3D stacked die and measure the overall impact of this architecture change on the NoC power consumption.

Extension to a 3D stacking based architecture

In this experiment, we try to evaluate the power consumption gain achieved through the use of a 3D stacking architecture for the 3MF platform to replace its original 2D layout.
To cut to the chase, 3D die stacking is a technique consisting in stacking different active silicon layers to increase the integration density[12]. The main advantage lies in the decrease in the average wire length: different parts of the chip are able to communicate through short interconnects by vertically crossing the thin active layers rather than having to run along the whole chip side length as it would be the case in a 2D layout. This average wire length reduction leads to smaller latencies, higher throughputs and wire power consumption savings.

Formalization This modification of the architecture is very easy to represent inside Nessie since we only need to change each individual link length which is a parameter of the Yeti models used to estimate the performance criteria. Each link length was already associated with a degree of freedom so that we only

needed to change its value in the previous simulation file: no other modification is required to move to the 3D case[12].
To compare with the 2D layout, we have defined ten possible variants of the 3D stacked 3MF architecture composed out of two or three active layers. For each variant, we detail the nodes that have been floorplanned on layer 1 (and layer 2 if the architecture contains three layers) while the remaining nodes that have not been explicitly mentioned are placed on the upper layer 1:

1) Layer 2 contains L2I1 and L2I2 memories
2) Layer 2 contains L2I1 memory while layer 3 contains L2I2 memory
3) Layer 2 contains L2I1, L2I2, L2D1, L2D2 memories and EMIF
4) Layer 2 contains FIFO and EMIF
5) Layer 2 contains L2I1, L2I2, L2D1, L2D2 memories and EMIF while layer 3 contains ADRES1 and ADRES4
6) Layer 2 contains ADRES1, ADRES2, ADRES3 and FIFO while layer 3 contains L2I1 and L2I2 memories
7) Layer 2 contains L2I1 and L2D1 memories while layer 3 contains L2I2 and L2D2 memories
8) Layer 2 contains L2I1 memory, ADRES1, ADRES2 and ADRES3 while layer 3 contains L2I2 memory, ADRES4, ADRES5 and ADRES6
9) Layer 2 contains the switch of the instruction NoC, the EMIF, L2D1 and L2D2 memories
10) Layer 2 contains ADRES1, ADRES4 and and L2D1 memory while layer 3 contains ADRES3, ADRES6 and L2D2 memory

The nodes of each variant have been manually floorplanned on the additional layer(s) to minimize the overall wire length compared to the initial architecture. We chose a vertical distance of $15\mu m$ between each layer (consistent with the current technology[13]) so that reasonable link power savings could be expected from moving the initial 48.66 mm^2 die to a 3D stacked architecture. Listing the length of all the individual links for each variant would be way too long and not of great interest since they can be easily found in the simulation files.

Power results For the three different resolutions we have evaluated the performance criteria of the ten 3D stacked architectures running the H.264/AVC application: wire power consumption for HDTV, 4CIF and CIF are respectively represented in Fig.6.18, Fig.6.19 and Fig.6.20. From these figures, we can see that some interesting wire power reductions can be achieved by switching from a 2D to a 3D layout for the 3MF architecture. The average

[12]Since the connection topology between the different nodes of the 3MF architecture remains the same for the 2D and 3D case, we can keep the platform structure intact inside Nessie which is a great benefit from our approach.

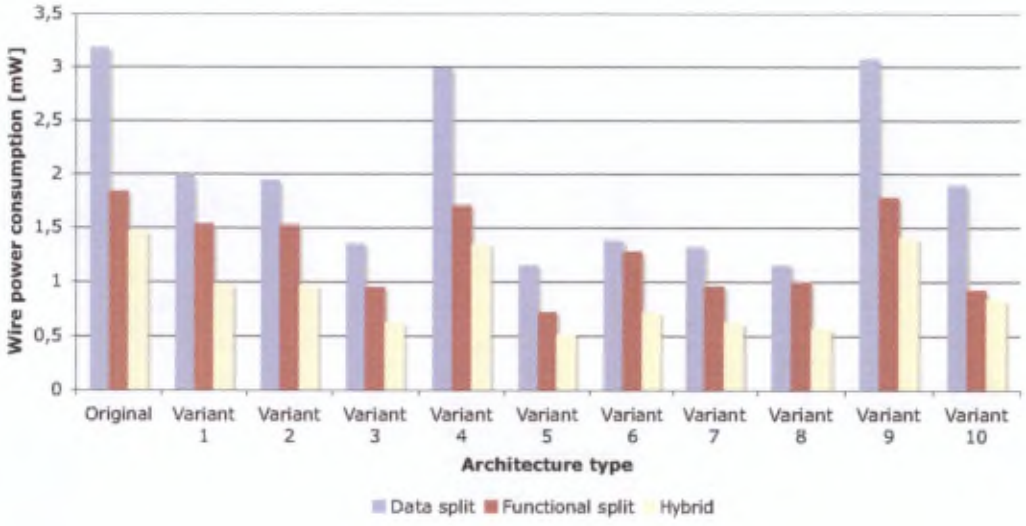

Figure 6.18: Contribution of wires to the power consumption of the 3MF architecture running the AVC application for ten 3D stacked variants in the case of an HDTV resolution

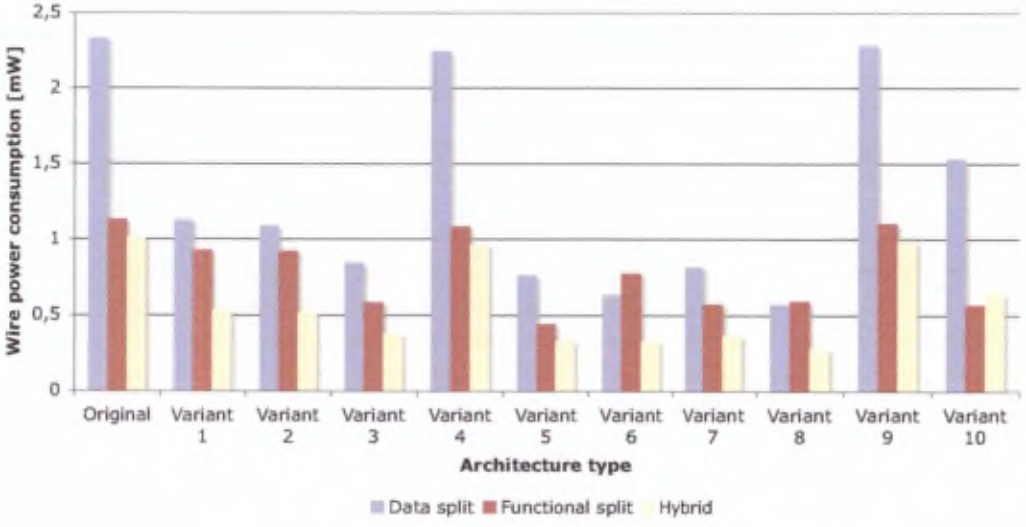

Figure 6.19: Contribution of wires to the power consumption of the 3MF architecture running the AVC application for ten 3D stacked variants in the case of a 4CIF resolution

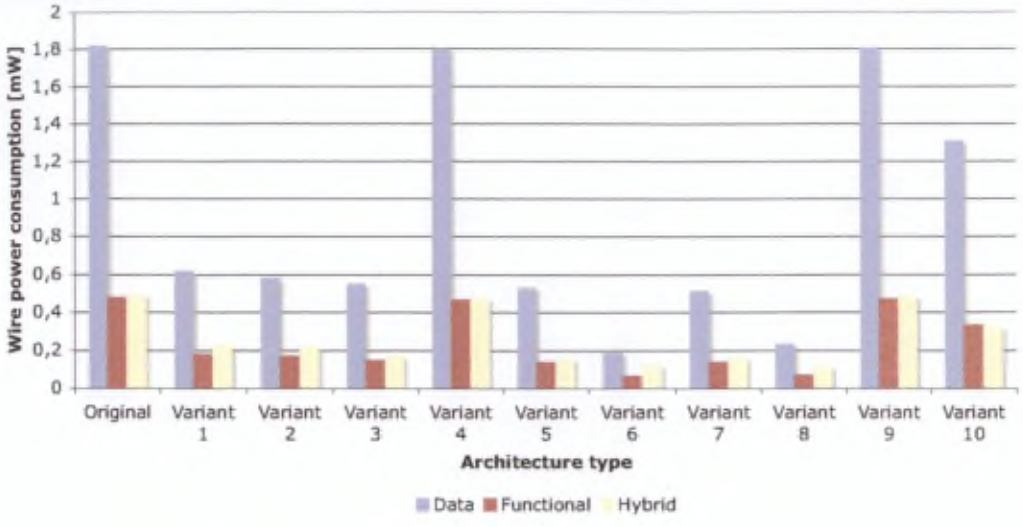

Figure 6.20: Contribution of wires to the power consumption of the 3MF architecture running the AVC application for ten 3D stacked variants in the case of a CIF resolution

Power gain (%)	Data split	Functional split	Hybrid
HDTV	43.0	32.8	41.4
4CIF	54.3	36.5	52.6
CIF	55.3	54.2	49.7

Table 6.7: Average wire power gain achieved by the use of 3D stacking for different mapping scenarios and resolutions

gain for each resolution and mapping scenario can be found in Table 6.7. With values ranging from 32.8% to 55.3%, we can state that 3D stacking efficiently converts the average wire length reduction into wire power savings. However it is possible to push the analysis a bit further and observe several interesting things from the three graphs:

- Variants 4, 9 and 10 perform not very well compared to the original 2D layout particularly for the functional data split scenario. If we look at the 3D layer distribution of the different elements of these three different variants, we can see that the FIFO and EMIF are placed on dedicated layers in order to decrease the wire length used to connect them to the different ADRES nodes. However [5] tells us that less than 5% of the total data bandwidth is implied in data transfers between/to the EMIF and the FIFO: this explains why reducing their average wire length to the

other nodes does not lead to great power savings.

- If we observe the three figures one after the other, it is striking to see that the overall power gain *relatively* increases compared to the 2D layout as the resolution decreases. Since the resolution is directly proportional to the amount of data exchanged over the data NoC but does not change the size of the code transmitted over the instruction NoC, lower resolutions tend to have on average higher instruction/data ratio to transmit. This entails that, based on our ten different variants, the instruction NoC tends to benefit more on a power consumption point of view from a move of the architecture towards a 3D layout than the data NoC does.
- Let us now have a look at the hybrid mapping scenario compared to the others. For the HDTV resolution, the hybrid solution is better for each of the ten variants and so is it in the case of the 4CIF resolution even if the difference becomes smaller. However for the lowest resolution, the functional split slightly beats the hybrid and becomes the best solution. As we explained earlier, the instruction/data relative bandwidth ratio decreases when the resolution decreases so that the instruction NoC activity contributes more and more to the total power consumption budget. The functional split has the advantage over the data split and the hybrid scenario of minimizing the instruction bandwidth by avoiding to send the same piece of code multiple times over different ADRES nodes. When the resolution decreases, the instruction bandwidth optimization thus becomes more and more important relatively to data bandwidth to the point where the functional split beats the hybrid solution. The optimal choice of the mapping scenario is thus dependent on the size of the code and the resolution so that any change in the AVC functionality implementation is to be taken into account to efficiently minimize the wire power consumption.

Based on these different observations and the analysis of each 3D stacking variant, we can define guidelines for the floorplanning and distribution over the different layers of the platform nodes. In order to optimize the wire power consumption by shortening the wires, we should follow this priority to bring the different nodes closer:

1 The ADRES nodes and the instruction memories

2 The ADRES nodes and the data memories

3 The ADRES nodes between each others

4 The ADRES nodes and the EMIF/FIFO

Until now we have seen that the 3D stacking technique used to modify the original 3MF architecture has shown very interesting reductions of the wire power consumption but how does it perform regarding the whole communication architecture? To answer that question, let us examine Fig.6.21 that represents

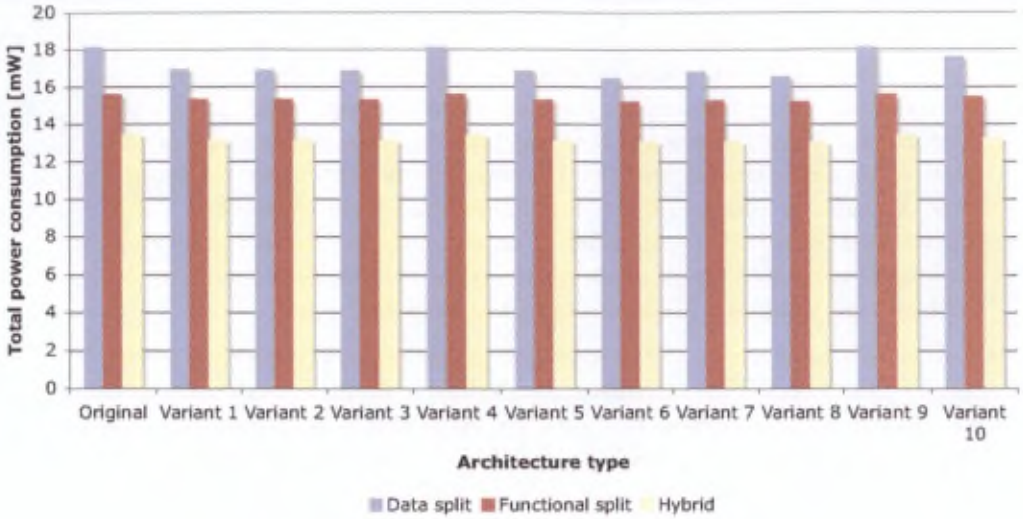

Figure 6.21: Contribution of the NoC to power consumption of the 3MF architecture running the AVC application for ten 3D stacked variants in the case of a CIF resolution

the complete network-on-chip power consumption for the CIF resolution for which we reached the best results in terms of wire power saving. As we can see the power consumption gain achieved through wire length reduction thanks to 3D die stacking is very small when we consider the entire network-on-chip. Indeed choosing the most power friendly 3D stacked architecture in the data split scenario only leads to a 9% overall power gain for the complete NoC compared to the original 2D layout while the same gain is only around 3% for the functional split and hybrid scenarios. These same power consumption savings would even look smaller when compared to the power of the different platform nodes.

3D stacking and manufacturing yield Now that we have discussed the power gain achieved through 3D stacking, we will try to give some insight into the cost associated with the use of this technique. Since 3D stacking processes are still under research and not ready to be fully transferred to fabs, it is difficult to define realistic manufacturing costs: however we can relatively rank our ten 3D variants in terms of efficiency and yield.
To do so, we have calculated the *wasted area* that we define as the difference between the cumulated total area of each layer in the 3D layout and the area of the 2D layout. Since all layers don't have the same number of components, some silicon area may be wasted while it could have been used to etch useful

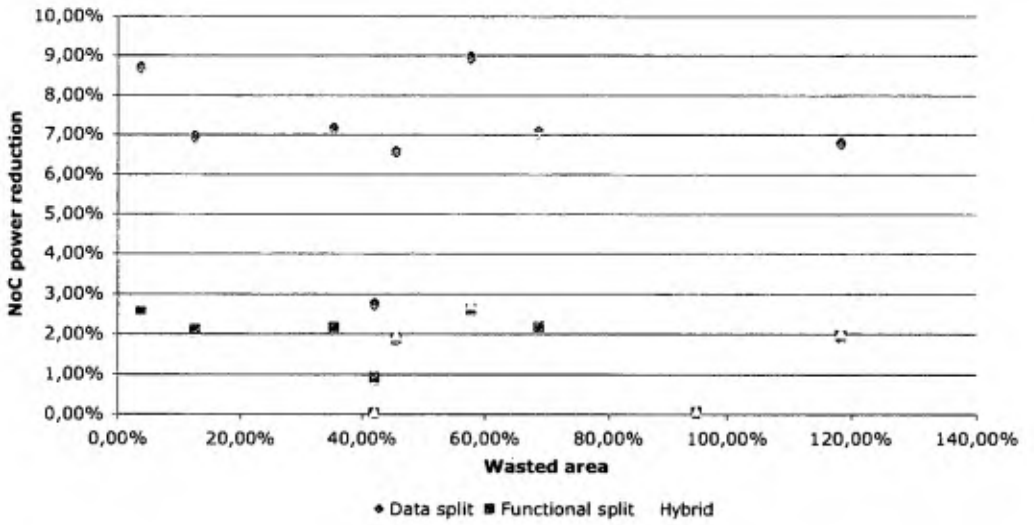

Figure 6.22: Power consumption reduction VS wasted area compared to the original 2D layout of the 3MF architecture

components instead. In other words for a same design, wasting some silicon area will thus -for a same amount of manufacturing time and resources[13]- turn into a smaller number of manufactured dies compared to a single active silicon layer architecture: this leads to a smaller manufacturing yield hence to more expensive dies to produce.

Fig.6.22 quantifies this potential yield loss by comparing the total NoC power consumption gain for the CIF resolution achieved by using the different 3D variants with the wasted area in percentage of the original 2D layout area. We assume that the floorplanning is 100% efficient so that each layer total useful area is just the sum of the different components area that it is composed out of.

On this graph, we seek the solutions that simultaneously minimize the wasted area while maximizing the power consumption savings: the more interesting solutions are thus those situated on the top left of the graph. At first sight, it is striking to see on this figure how spread is the wasted area value ranging from 2.6% to 118% for the the worst solutions. On the contrary, the power consumption reduction of the whole network-on-chip vary in a much smaller range from 0.25% up to 8.74%. Taking into account the wasted area is thus

[13] We implicitly assume that a 3D stacked die composed out of N active silicon layers requires more than N times the duration needed to manufacture a single active silicon layered die of the same area. This assumption is more than reasonable since all layers of the 3D layout will anyway require to be etched separately so that their assembly is an unavoidable overhead.

much more crucial in the optimization process than looking at the single power savings.

With the absolute numbers being underestimated by the perfect floorplanning assumption, our example demonstrates that the wasted area (hence manufacturing yield) is much more sensitive to 3D stacking organization than the power consumption reduction is. The stress is put on design tools and designer work to prevent additional design and manufacturing costs from making the achieved power savings insignificant.

Based on these different results, we may thus wonder if 3D stacking is really worth the effort? The answer to that question is however not so simple: from a pure energy and design/manufacturing cost, it is clearly not. Indeed the power dissipated in the communication wires will probably always represent a very small part of the global platform power budget so that the global power reduction will always be outweighed by the design time and money spent in it (more etching masks, more complex design, heat dissipation issues etc.). On the other hand using 3D stacking has some side advantages that have not been quantified in our comparison:

- One important effect of wire length reduction achieved through the use of 3D stacking is the reduction of the communication latency and the increase of its bandwidth: two benefits that we didn't evaluate because it was out of the scope of this study focusing on power consumption.
- Instead of simply assigning the platform components to the different layers, using 3D stacking could allow the designer to perform the floorplanning of a single component on several layers improving both their individual performance and the platform performances.
- By using different silicon processes for each layer, it is possible to optimize the technology for the component type that is etched. For instance, memory and pure logic are often implemented using different technologies as highlighted in the ITRS roadmap[14].

In summary, we could answer that even if moving to 3D stacking is not very effective in reducing the power consumption of an existing platform, it has other side advantages that we could benefit from at the expense of putting the stress on EDA tools to maintain reasonable cost and efficiency.

6.4 Discussion of the use Nessie

Through our two previous case studies we have used different functionalities of Nessie and performed several experiments with it. Based on these results we are now able to discuss what Nessie is good for and its current limitations:

- Fast to execute but long to initialize: Nessie executes fast but first requires a lot of informations from the designer to model the system and its performances. For instance, our AVC case study only takes 10 seconds

(when disabling the console information display) to generate the 30 functionality/platform combinations that we explored even if the main XML input file is ten thousand lines long. We could lower this data entry complexity by helping the designer with a graphical user interface although it will not lower the modeling effort that will anyway be required to estimate performances.

- Automated: Design space exploration and automated plot generation makes Nessie very comfortable to use thanks to the concept of degrees of freedom and performance criteria. In our first case study, this allowed us to successively sweep on a parameter and automatically perform for each of them a new simulation. In the AVC case study, the automation provided by Nessie gave us the opportunity to perform a batch of 30 simulations at once while other tools might have required 30 successive simulations to be initialized and launched.
- Strict: Nessie uses a strict definition of the system i.e. a hierarchical description of both the functionality and the platform with Yeti performance models associated to the atomic platform primitives. This clear separation between functionality and platform is inherited from the HW/SW co-design approach: any system described differently from our convention will thus be more difficult to represent. That's the case of our second case study that already had a single unified description of the functionality mapped onto the platform and forced us to "unmap" them. Nessie intentionally discourages the user from using such descriptions since we believe that separating the functionality from the platform should be the best way to describe a system in a design space exploration perspective.
- Extendable: Nessie could not at the moment pretend to be the top notch design space exploration tool because there is still so much to do in the domain. It introduces many different concepts (flexible mapping, automatic exploration of the design space, degrees of freedom etc.) that have all been implemented but could be extended in many ways. The current implementation has been thought as flexible as possible so that these different aspects could be extended easily without having to start from scratch: these extensions will be further discussed in the next chapter devoted to future work.

6.5 Conclusions

In this chapter we have demonstrated the proper working, the use and the different features of Nessie based on two different case studies respectively focusing on design space exploration for an hypothetical yet realistic design and on the modeling of the power consumption of a platform running an H.264/AVC application.

In the first case study we started from an hypothetical application and mapped it onto several architectures ranging from a single computation node to multiple nodes architectures with different communication topologies. We have performed different experiments including the estimation of the computation time, the compromise between energy and computation time as well as the influence of the static power on the total energy. These different experiments illustrated some of the different features (input parameter sensitivity analysis, automatic exploration of the solutions, flexible performance criteria definition) applied to examples helping the designer to quantify design performance compromises and enabling an easier dimensioning of the different design parameters.
In the second case study we have demonstrated how Nessie is able to model a real system based on the 3MF platform running an H.264/AVC decoding application. We have then performed several experiments to estimate the power consumption of the whole communication architecture for three different mapping scenarios of the application and compared it to the results of the original paper resulting in a very reasonable average error of 0.77%. For ten possible variants of the 3MF architecture based on different 3D layouts, we have reached a maximum 9% power reduction for the communication architecture but highlighted potential yield losses making the use of 3D stacking subject to caution.
Looking at the different aspects that we investigated during our two case studies, we can state that Nessie is able to represent the most important aspects of a system by representing explicitly the application and the platform. Automatic exploration of the design space based on the degrees of freedom combined to the flexible definition of performance criteria could make such a tool very useful for a designer. In counterpart Nessie pays its flexibility by an important modeling effort to get to that result.
First, XML files are quite heavy and may become painful to define for very complex systems but this issue can be partly addressed by using a graphical user interface (this topic will be further discussed in the next chapter devoted to future work). However once a simulation has been initialized, it is very easy to change the degrees of freedom, generate new plots based on all performance criteria and degrees of freedom values, swap a platform for another and so on: the initial effort may be high but performing new simulations require much less time.
Second, the designer needs to capture the whole application/platform and the models for performance estimation in a strict way which may be a time-consuming task. If this could be seen as an overhead, I personally think that it has the merit to make this modeling task mandatory which should be part of any design: it's the price to pay to benefit from accurate and flexible performance estimation.

Bibliography

[1] ARM corp., "Arm doi 0035-4 - arm7 familiy guide," ARM, Tech. Rep., 2005. [Online]. Available: http://www.arm.com/pdfs/ARM7_thumb_flyer_35_4.pdf

[2] O. Sentieys, "Gestion intelligente de l'énergie : Gestion intelligente de l'énergie : aspects matériels et logiciels," ENSSAT - Université de Rennes 1, Tech. Rep., 2002.

[3] H. Hanson, "Static energy reduction techniques for microprocessor caches," 2001. [Online]. Available: citeseer.ist.psu.edu/article/hanson01static.html

[4] M. Horowitz, "Circuits and interconnects in aggressively scaled cmos," in *proceedings of ISCA 2000*, 2000.

[5] D. Milojevic, L. Montperrus, and D. Verkest, "Power dissipation of the network-on-chip in multi-processor system-on-chip dedicated for video coding applications," *Journal of Signal Processing Systems*, p. 15, June 2008.

[6] G. Sullivan, P. Topiwala, and A. Luthra, "The h.264/avc advanced video coding standard: Overview and introduction to the fidelity range extensions," in *SPIE Conference on Applications of Digital Image Processing XXVII*, vol. 5558, Aug 2004, pp. 53–78.

[7] I. Richardson, "White paper: An overview of h.264 advanced video coding," Vcodex: Video compression design, analysis, consulting and researsch, Tech. Rep., 2007.

[8] R. L. Myers, *Format and Timing Standards published in Display Interfaces.* Wiley-SID Series in Display Technology, 2003.

[9] D. Milojevic, L. Montperrus, and D. Verkest, "Power dissipation of the network-on-chip in a system-on-chip for mpeg-4 video encoding," in *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian.* Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 392 – 395.

[10] F. Bouwens, M. Berekovic, B. D. Sutter, and G. Gaydadjiev, "Architecture enhancements for the adres coarse-grained reconfigurable array," in *HiPEAC*, 2008, pp. 66–81.

[11] Arteris TM, "Network-on-chip network-on-chip : The future of soc power management," in *CDN Live*, June 2006.

[12] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3d) microarchitecture," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–479.

[13] E. Beyne and B. Swinnen, "3D System Integration Technologies," *Integrated Circuit Design and Technology, 2007. ICICDT '07. IEEE International Conference on*, pp. 1–3, May 2007.

[14] "International technology roadmap for semiconductors 2007." [Online]. Available: http://www.itrs.net/

Chapter 7

Future work

Through this dissertation we have presented our contribution to prediction performance tools with the combination of two original tools called Nessie and Yeti. We have successively explained how we built them from scratch, the principles they rely on and eventually demonstrate their abilities to solve electronic systems design problems. During this work we have highlighted new problems and questions that could do some very interesting new research topics. We decided to create a single chapter entirely devoted to the presentation of these new ideas that arose from this thesis in order to draw a complete picture of the future work.
This chapter will be divided in three main parts:

- *Concepts* describe the new ideas that still have to be largely explored before being implemented.
- *Implementation* details the different new features that do not require any further conceptual work and that we would like to appear in the coming versions of Nessie and Yeti. Most of these changes are almost ready to be coded but were either out of scope either highly time consuming and not crucial for the purpose of this work.
- *Design flow integration* discusses the benefits and methods of integrating our tools inside existing design flows.

Let us start with the new concepts that we would like to implement.

7.1 New concepts

7.1.1 Improving Nessie exploration layer

As explained in the chapter related to Nessie (see Sec.5.1.1), we provided our tool with a simple interface defining a solution as a set of inputs (the degrees of freedom gathering the different design choices) and outputs (the performance criteria representing the different figures of merit of a given solution). The advantage of such an interface lies in the abstraction of the inner mapping

and performance evaluation core so that we could implement any exploration policy above this interface to define how to explore the design space. At the moment, we have only implemented a full factorial solution exploration policy as a demonstrator: all combinations of degrees of freedom are tested and the resulting performance criteria are then explored for each of them. If this policy is suited for a small number of solutions, more clever solutions could be used to explore larger design spaces.

As part of a prospective work, we supervised a student work[1] that consisted in the study of methods and tools for the exploration of large solution spaces and its application on a practical case taken from the computer science literature. We selected the model of Codrescu (see Sec.3.2.2 for more details) that we completed with some new models in order to define power consumption as an additional criterion to computation power. With the help of a department specialized in multicriteria analysis in our university, we surveyed several methods and selected two metaheuristics methods: the *taboo list* and the *simulated annealing*[1]. The experiments were performed on a design space composed out of 200k solutions generated by the value sweep of three inputs parameters feeding our model and showed very promising results. Exploring the entire design space took 20 minutes using Matlab while the simulated annealing method only took 20 seconds to generate the whole set of Pareto optimal solutions while discarding suboptimal solutions. The resulting Pareto optimal curve is represented in Fig.7.1 where each point represents a different solution with a different compromise between power consumption and computation power.

This prospective work has put into light very interesting methods for the implementation of new design space exploration policies above Nessie's interface. Such metaheuristics could drastically reduce the exploration time of large design spaces while finding out *good* (meaning not too far from the optimum) solutions.

7.1.2 Decreasing the estimation time

While Nessie is able to estimate performance criteria based on an explicit mapping of the functionality on the platform in reasonable times, it could be desirable to provide the user with the ability to find the same values without performing the whole mapping and for a smaller computation time. This could be done using a Yeti model that would be able to extract the essential information out of functional/platform structures and the mapping policy and use them in return to estimate the performances as accurately as possible. Such a process depicted in Fig.7.2 is expected to offer reduced estimation times hence enable the exploration layer to explore wider portions of the design space in a given amount of time.

The calibration of the models used inside Yeti could however be a real problem

[1] More information about multicriteria analysis and metaheuristics can be found in [2] and [3]. We won't further detail these topics since it goes out of the scope of this discussion.

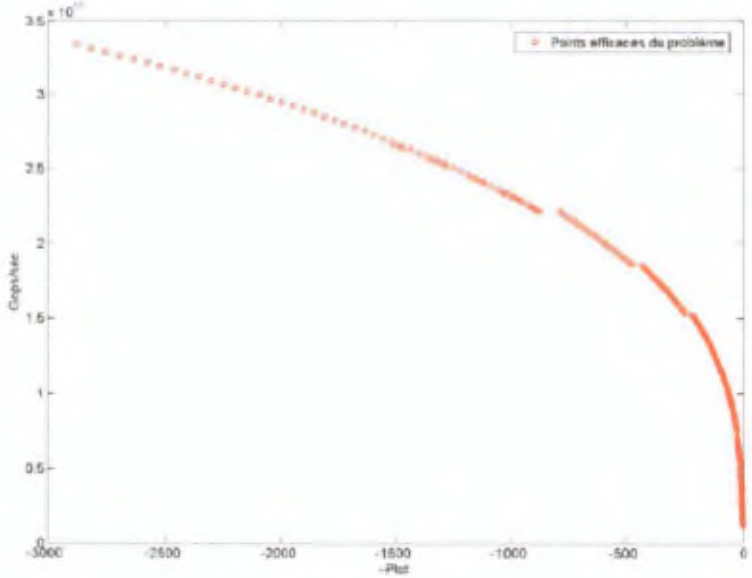

Figure 7.1: Pareto optimum curve of the computation performances versus power consumption for the extended Codrescu model[1]

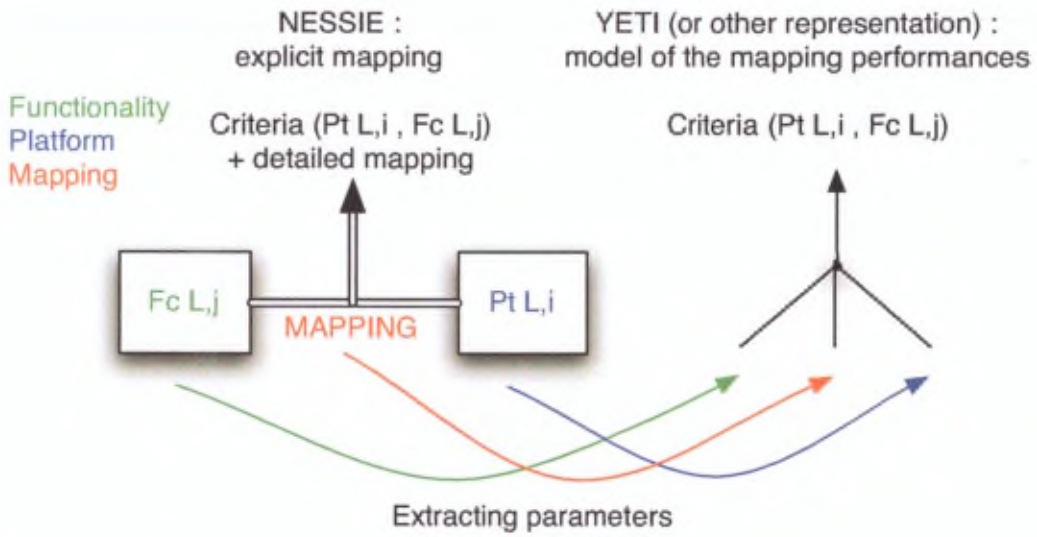

Figure 7.2: Modeling of the performance criteria resulting from Nessie explicit mapping by functional, platform and mapping parameter extraction

to guarantee a sufficient level of accuracy. To tackle this issue, we could imagine to add some learning abilities to Nessie by replacing the fixed Yeti model with a black box that would progressively build a model learning from the previously performed experiments. The general mechanism of such a learning based model would be the following: each time Nessie performs an explicit mapping, the functional, platform and mapping parameters are extracted and associated with the output performance criteria. This new experience result set can then be fed back to the learning tool that will use it to refine its previous performance model to match the predictions with the new dataset. The more we use Nessie, the more accurate the model of the mapping performances will become. The exact underlying learning mechanism could be chosen among the state-of-the-art methods of intelligent learning (for instance expert system[4], neural networks[5], data mining[6], etc.).
To accelerate the learning process and calibrate the models faster, we could imagine to submit different automatically generated cases to Nessie and let it learn as long as required. Several practical limitations could however be anticipated:

- It is difficult to predict how many parameters from the functionality, platform and the mapping should be used and if they would all significantly influence the resulting performance criteria. If we need too many parameters to characterize these three, we could end up with a tremendous number of experiments to perform to build up a sufficiently large sample of cases to train the model: this would limit the practical use of this approach due to limited computation power. A study should be made to anticipate this problem and demonstrate whether or not this method could be feasible: design of experiments[7] could therefore be of great help.

- We could also wonder if randomly generated applications and platforms could represent a relevant set of training cases to build a model that would afterwards be used on realistic applications and architectures. It's probably up to the user to wisely determine the parameter range and define some limitations on their values and possible combinations/patterns in order to avoid the generation of structures that will never be encountered in practice and therefore bias the model

Whether or not we use models based on progressive learning, it would be very interesting to explicitly represent the compromise between the accuracy of the results predicted by the model and their estimation time in order to take this information into account at the exploration layer.

7.1.3 Developing metrics for application and platform characterization

As we described in the above section, extracting the parameters of a functional/platform structure is a crucial task before being able to train the learning tool. Aside from this particular use, parameter extraction from structures (particularly for the functionality) would also be a nice feature to quantitatively compare different competing implementations based on objective criteria. These different extracted parameters could be used as input parameters in our automatically generated plot which would give more precise and quantified information about a functionality and platform than just its type.
We will not try to set an exhaustive list of all the possible parameters but we could think of the following ones for the platform: the number of blocks, a measure of their heterogeneity, the mean number of connections between the blocks, a possible regular topology etc. Regarding the functionality we could enumerate the critical path[2], the mean data/control dependence of functional blocks, the average data token size, the proportion of the different functional types etc.
The determination of relevant and non-redundant parameters capturing the essential information of functionality or platform is still a wide research topic that has been partly investigated in a master thesis on application profiling [8].

7.1.4 Introducing design cost related issues inside nessie

At the moment, Nessie focuses its performance evaluation on the properties of the designed chip (the silicon area, the energy, etc.) but does not take into account the cost of the design process that leads to this chip. Manufacturing, testing and design costs, yield, design time[9] are some examples of criteria whose values should be put in balance with the chip properties[10]: for instance, how many additional dollars per manufactured die am I willing to spend on an overall 2% power consumption reduction? The compromise between chip and design related performance criteria could thus provide the designer with new relevant information in the design choice: this strives for the explicit modeling of design related activities and costs inside Nessie.
Many points should still be investigated before being able to develop such an integrated framework, here are however some promising ideas:

- As a starting point we should be able to evaluate the design time and complexity of any functional/platform block mapping. We could be tempted

[2]The number of consecutive functional blocks is not sufficient to determine the critical path of a functionality since the different blocks have execution times both depending on the functional type and the platform block that it will be mapped on. This is exactly the reason why we previously emphasized that ASAP and ALAP methods make few sense in the context of our problem (see Sec.5.4.2). Instead of the number of blocks, we could use statistical distributions of the execution time of a given path.

to state that design time is directly proportional to the size of the designed block however, as demonstrated in [11], the number of transistors designed per man-month is very dependent on the type of architecture making the previous approximation quite inappropriate. A full survey of the related literature and the possible development of design time/cost models would probably be worth a dedicated reaserach.

- We could reuse our multiple abstraction levels formalization for the purpose of design flow modeling: this situation is depicted in Fig.7.3. Besides the previously defined abstraction axis (see Sec.5.1.2), we have added an horizontal axis representing the evolution of the design process in itself where each graduation is a design step (refinement, mapping, iteration etc.). Using this paradigm we can represent the hierarchical evolution of the chip description for each design step and use a model to define the absolute time separating two successive design steps by taking advantage of the available functional, platform and mapping information. Parallel design activities could also be represented by superposing different timelines above each other.
- There is a striking analogy between the functional/platform mapping problem and the design activity: as the different functional blocks are mapped onto the platform structure in order to optimize time or other performance criteria, the design process consists in allocating design resources (manpower, tools, etc.) to design tasks in order to meet the deadlines while minimizing design cost and time. Design planning is, in a certain extent, nothing more than a scheduling/allocation problem where design tasks require the expertise of designers offering a certain design productivity. If it is not guaranteed that methods used in the case of functional/platform mapping could be directly reused for design process modeling but some concepts could at least be transfered or used.

The modeling and study of design cost is thus relevant in the context of electronic system optimization and could therefore be an interesting subject of research.

7.2 Implementation and future tool evolution

7.2.1 Nessie

Nessie has now reached a stable implementation state but could in the future benefit from new or extended features to optimize its performances and make it more flexible. These different ideas are detailed in the following sections.

Improving mapping

As explained in Sec.5.4.3, the mapping is divided in three parts: the allocation, scheduling and routing. We have developed different methods to perform

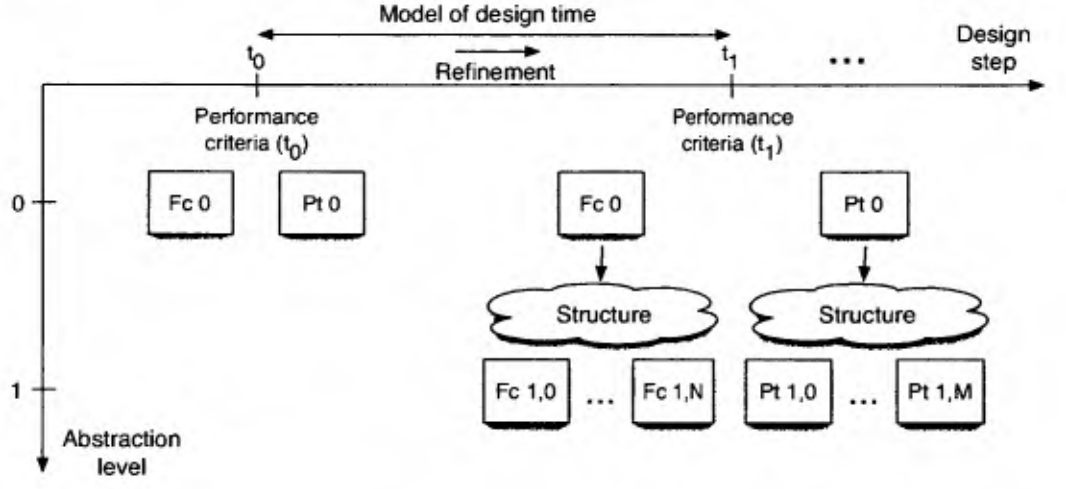

Figure 7.3: Simultaneous representation of the design activity and the chip hierarchy organized in different abstraction levels

these three mapping steps but new ones could be explored or tested to enhance the performances. For instance the allocation of a functional block is, at the moment, performed based on the minimization of a cost function defined by a mathematical combination of the different performance criteria of each potential platform block. A possible extension would be to define the allocation priority of the different functional blocks while taking functional structure into account (minimization of the execution time difference between parallel paths, defining higher priority for functional blocks whose execution completion will allow the generation of a maximum new functional blocks, etc.).
At the moment, we have only defined one single policy for allocation, scheduling and routing but in order to enable the use and choice of new mapping methods, we should think of a mechanism to add new degrees of freedom defining which exact method should be used for each mapping step. The modification in Nessie would consist in defining for each mapping step a super class that would have several derived classes associated with each particular method of a mapping step. This would enable the definition of degrees of freedom for the mapping and allow the user to define more flexible mapping policies instead of having them hard coded in the framework.

Automatic generation of platform structures

Instead of explicitly defining all the platform structures, it could be very useful to have a generator of regular topologies (mesh with different connection topologies, bus with N connected blocks, ring and star topologies etc.). From

a user perspective, this would reduce the time spent on filling XML files with repetitive content and avoid error-prone copy/paste operations.

Automatic generation of functional structures

As for the platform structure, the automatic generation of functional structures could be very useful to simplify the description of the application but also to provide Nessie with automatically generated cases for evaluating the performances of a given mapping method or performing model learning.

Memorization policy

Two interesting extensions still need to be added to the current version of Nessie concerning the memorization of data tokens.
First, each memory has, at the moment, an infinite size and the area is only fixed by the criteria of the platform block. To make it more realistic, we should turn the area mandatory criterion into an *area per bit* value so that we could calculate the total memory area from the maximum data amount that was ever contained in the memory (an information only available after the mapping process).
Second, we should define flexible memory policies to manage the data tokens stored in the different memories during the mapping process. At the moment, the hardcoded policy consists in keeping track of all the memorized data tokens and removing all the instances of a particular data token when there is no functional block that still needs to consume it: this policy acts a bit like a garbage collector in a software. More complex policies could however contribute to decrease the total required memory size or decrease the execution time by transferring a data token to distant memories when the activity of the chip is under a certain threshold in order to move it closer to its future consumers.

Modifying the routing block reservation policy

During the routing process, Nessie uses a mechanism to reserve all the platform blocks along the determined path in order to make sure that no other route using those blocks will be established during the data token transfer. When the token completely leaves a block belonging to the route, it is put back in the pool of available blocks for routing. If this method works perfectly, it has the drawback of reserving blocks that won't necessarily be used at once: the closer we are to the destination, the longer the block will have to wait for the data token to effectively pass through it. However these platform blocks could have been used instead to perform another operation meanwhile: transforming the bit reservation into a reservation timeline scheduling all the time intervals where the block is unavailable would solve this issue and optimize the platform block usage. If we are in a case where data transfers don't take much time

compared to computation operations, the benefit would however be small or even inexistent: that's why we decided to leave the implementation of this feature for future work.

Additional functional structures

As mentioned in Sec.5.3.1, Nessie has been implemented to enable the simultaneous use of different models of computation to represent the data/control dependency. Aside from Petri nets, functional structures based on DFG and KPN (see Sec.4.3.3) would give more flexibility to Nessie and enable its interfacing with a larger number of existing tools.

7.2.2 Yeti

Additional operations

At the moment, Yeti already knows a lot of basic mathematical operations but could still use some new analytical operations but also a set of boolean operations. As we have shown in Sec.2.5.1, it is very simple to add new operations to the current framework in a matter of minutes: if required in the future, any missing operation should be added without hesitation.

Statistical parameter handling

Currently Yeti only manages scalar values as input and output parameters of its models. The addition of statistical distributions based values for parameters and the creation of dedicated operations to handle them could provide the user with new features to model the uncertainty on fluctuating values instead of using input parameter sensitivity to figure out the model output variation.

Releasing Yeti

At several times, we were asked during conferences if Yeti was available for download and use. This demand could probably lead to a near future in the online availability of Yeti in two possible formats: an online repository of models or a standalone client tool. While the first option has been chosen by the GSRC for their GTX tool, we have to admit that, although the initiative was very encouraging, it didn't get the success it deserved and almost no models were added to this online repository. There is no reason why we could do better, the second option should therefore be preferred.
Making this tool available will however require a lot of support: XML input file documentation and user guide manuals will have to be written and some time should obviously be spent on user support and framework maintenance to take into account reported bugs or user comments. We felt useful to mention this opportunity in this chapter devoted to future work since it emphasizes a

demand from the community that could be profitable for our department to increase its reputation in the domain seen from the outside.

7.2.3 XML parser update

At the beginning of this work we chose the APACHE XercesC library to parse our XML documents using SAX but at the time of version 2.7, no support was provided for XPath and Xinclude. With the recent release of XercesC 3.0, these two XML technologies have been integrated in the library offering new interesting features that could be exploited in the context of our work:

1 *Xinclude* allows the inclusion of an XML file into other documents by simply referring to their names. This feature is very useful to avoid XML content from being duplicated whenever it is necessary enabling a better data consistency. In our case we could imagine to store relations, SW/HW structures, SW/HW primitives in separate files to build up a repository of models and representations and use Xinclude to include them as necessary.

2 *Xpath* allows the user to perform powerful queries on an XML document to extract the required information. Many functionalities are available to sort out the data of a document and make the manipulation of large files particularly easy: some ideas to use Xpath are given in Sec.7.3.3 of this chapter.

7.3 Integrating Nessie inside an existing flow

Nowadays there is a strong demand for languages enabling the description a functionality and a platform at very high abstraction levels while still allowing the designer to simulate the behaviour. If languages like SystemC TLM 2.0 follow that trend and offer very interesting features regarding the functional point of view, we could probably combine them with Nessie's capabilities in performance estimation of non-functional performance variables. In this section, we discuss the opportunities and possible issues for the integration of Nessie into existing design flows.

7.3.1 Integration method

Nessie is a standalone tool requiring an explicit description of the platform, functionality and degrees of freedom in order to deliver performance criteria estimations. At the moment all these information need to be gathered by the user and then entered manually into Nessie: this process requires too much formalization and work to seduce users in the perspective of an integrated design flow.
As depicted in Fig.7.4, two solutions are therefore possible to combine Nessie with existing design flows:

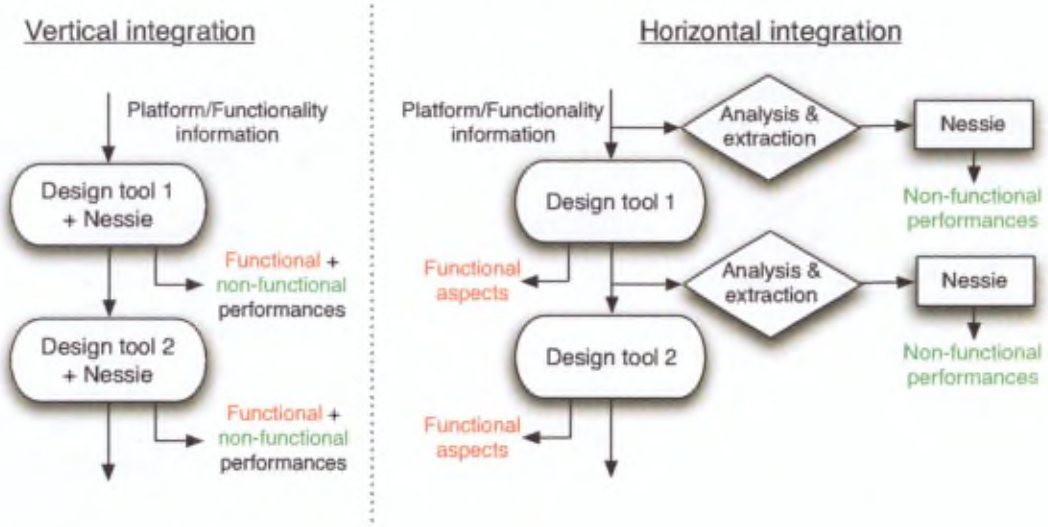

Figure 7.4: Comparison of vertical and horizontal integration of Nessie inside existing design flows

1 Vertical integration: Nessie engine or some of its performance evaluation methods and concepts are directly integrated inside existing tools to provide non-functional performance estimation. If this approach leads to the total integration of our framework into a seamless design flow, it has the drawback of requiring some heavy modifications to existing tools and languages which is not very realistic.

2 Horizontal integration: Nessie remains a standalone tool and works in parallel with the existing tools/languages of the design flow. To lower the overhead of entering manually models inside Nessie for non-functional parameter estimation, we can benefit from the information provided anyway to design flow tools, analyze and convert them into the input format required by Nessie.

In the context of a smooth integration of Nessie into an existing design flow, we think that the horizontal approach is more realistic and doesn't require both the existing tools of the flow and Nessie to be modified. In return we need to translate the existing functionali and platform descriptions into Nessie input format files. Regarding the functionality, different tools exist to extract data/control dependency from an existing C code (like for instance in Design Trotter, see Sec.4.3.4). Concerning the platform, the topological description format may largely differ from one tool to another: the extraction of these information to transform them into Nessie platform structures should however not be a problem at all. Finally the information about the different func-

tional/platform primitives and their performance models should need to be explicitly provided by the user since they are missing in functional verification focused tools: this is however the price to pay for non-functional performance estimation. The analysis and platform/functionality information extraction could also be reused in a pure design space exploration context to allow the user to define the functionality and the platform in other formats than the ones defined by Nessie.

7.3.2 Adding a GUI

The only way to specify the inputs of Nessie currently consists in writing the different required XML input files. If the task is simplified by an XML editor performing tag autocompletion based upon Nessie and Yeti XML schemas, this method isn't still very user-friendly. A graphical user interface could instead make this work much less fastidious for the user and provide him with a few interesting features:

- Graphical functionality and platform structure edition to build the operation dependency and the architecture topology
- Interactive edition to build Yeti model using parameter-based degrees of freedom
- Based on the previously defined graphical representations of the platform/functional structure, we could, once the simulation is over, use the resulting timeline files to illustrate the co-execution of the functionality on the platform. The idea is to allow the user to see how the different data tokens are exchanged between the different platform blocks, watch the evolution over time of the execution of different SW blocks, etc. By adding a filter, we could also monitor the activity of a particular platform block or petri place to give further understanding of the resulting scheduling, allocation and routing. All the information required to build this interactive simulation are contained in the generated timeline files and only need to be graphically translated.

The GUI should thus provide the user with the ability to enter all the information in a graphical way and then automatically generate the XML input files according to the previous descriptions. The user could then be able to launch the simulation and analyze the results or get a report in the case of run-time errors due to bad information provided to Nessie. We have to mention that the input data syntaxical and semantical verification is entirely left to Nessie that embeds the XML document validation using the different schemas of Nessie and Yeti provided. This makes the work of the GUI much easier since it only has to report the errors generated by Nessie to the user.
The development of such a GUI is a priority to simplify and accelerate the use of Nessie and is the subject of a currently ongoing master thesis.

7.3.3 Result analysis

Easy analysis of the output results is crucial in order to simplify the task of the designer: thousands lines long reports are often discouraging and worth nothing if they cannot be interpreted quickly and efficiently. At the moment, Nessie already performs with honor in that domain by providing the user with automatic plotting of any combination of the performance criteria/degrees of freedom along with the generation of activity reports providing information over the efficiency of the mapping for a given solution. However we could go a step further.
In the previous section, we described the premisses of an interactive simulator based on the timeline report files. This could be used to understand why a given platform/functionality combination performs better or worst than another solution.
Another improvement would be to provide the user with some statistical analysis features of the different solutions generated by the exploration policy. Using the Xpath support of the new XercesC 3.0 library in a separate C++ program, we could be able to query the resulting XML file summarizing the different solutions (with the performance criteria and associated degrees of freedom):

- How many and which solutions have a criterion X value greater than Y ?
- What is the distribution of criterion X value ?
- What are the top five solutions that maximize the X*Y criteria values product?

Such a feature would incredibly simplify the work of the user, allow him to classify the different solutions and to extract those which fulfill these conditions.

7.4 Summary

In this chapter we have identified different ideas and great opportunities for the future of Yeti and Nessie. We classified them in three categories: new prospective concepts, implementation improvements that could optimize the performances and flexibility of both frameworks and finally some hints for their possible integration inside an existing design flow.
If Yeti, in its standalone version, is sufficiently mature to be released, we will still need some efforts to make Nessie usable by the computer science community without having to read this entire thesis.
Sorting out the different presented options for immediate implementation, I personally believe that the most promising developments to enhance Nessie and Yeti are the implementation of a GUI, the integration of tools to convert existing functional descriptions from widely used languages to Nessie functional structures and finally the addition of advanced analysis methods of the results.

Regarding the more conceptual parts of the work to do, we could imagine to extract two research works that could lead to PhDs.
The first subject is about the *development of new exploration policies for fast design space exploration* and would consist of the following tasks:

- Extensive study of the different methods to efficiently explore spaces with a large number of solutions applied to the field of VLSI systems
- Development of methods/rule of thumbs to select the different parameters of exploration policies (total exploration time, specific values of the heuristics parameters, etc.)
- Adding some flexibility to the different mapping steps in order to make explicit all the degrees of freedom for mapping
- Defining several case studies and benchmarking the exploration methods to prove their efficiency to explore large design space and identify good solutions in reasonable times

The second subject would be related to the *development of cost models for VLSI design flows and their integration inside chip performance prediction tools* consisting of several tasks:

- Surveying of the literature about manufacturing/design time and cost, yield and all performance metrics related to the efficiency of the design flow
- Extracting useful information of project management techniques and investigate if we can reuse chip allocation techniques for the optimization of design time and resources
- Integrating the cost modeling of this design activity into Nessie to explicitly evaluate the compromise between chip and design activity performance
- Identifying several case studies by studying existing design projects and extracting relevant data
- Showing in which extent the combined study of chip/design related performance can lead to better compromises and help to meet project deadlines while improving overall chip design performances.

Bibliography

[1] N. Simons, "Optimisation des architectures numériques: formulatin du "design flow" sous forme d'un problème d'aide à la décision multi-critères," Université Libre de Bruxelles, Tech. Rep., june 2006.

[2] P. Vincke, *Multicriteria Decision-Aid.* J. Wiley, New York, 1992.

[3] J. Dréo, A. Pétrowski, P. Siarry, and E. D. Taillard, *Métaheuristiques pour l'optimisation difficil.* Eyrolles, September 2003.

[4] J. C. Giarratano and G. D. Riley, *Expert Systems: Principles and Programming*, 3rd ed. Course Technology, February 1998. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0534950531

[5] L. Francis, "The basics of neural networks demystified," in *Contingencies Workshop*, 2001, pp. 55–61.

[6] M. Kantardzie and A. N. Srivastava, "Data mining: Concepts, models, methods, and algorithms," *Journal of Computing and Information Science in Engineering*, vol. 5, no. 4, pp. 394–395, 2005. [Online]. Available: http://dx.doi.org/10.1115/1.2123107

[7] D. C. Montgomery, *Design and Analysis of Experiment*. Wiley, 2000, vol. 5 edition.

[8] P. Pilavachi, "Application profiling: state of the art and case study," Université Libre de Bruxelles, Tech. Rep., June 2006.

[9] D. Harris, "Introduction to cmos vlsi design - scaling and economics - course given at harven mud college - claremont usa," 2005.

[10] A. B. Kahng, "The cost of design," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 136, 135, 2002.

[11] Numetrics Management Systems Inc., "Design complexity and productivity," Numetrics Management Systems Inc., Tech. Rep., 2004.

Chapter 8

Conclusions

In this dissertation, we have developed a combination of two original tools, called Nessie and Yeti, in the context of design space exploration and performance estimation methods for VLSI systems. The main contributions of our work are:

- A characterization of VLSI performance estimation tools/methods based on five classification criteria. Based on this review of the literature, we have identified new opportunities to cope with the limitations of existing solutions and enhance them.
- The design of and implementation Yeti, an innovative C++ library/standalone tool for the flexible and dynamic calculation of mathematical relations. Yeti enhances the current state of the art tool in that domain by:
  - Providing the user with flexible scripting capabilities to make large simulation campaigns easier
  - Introducing the notion of model reversibility to facilitate and encourage existing model reuse
  - Relying on a very strict XML grammar to enable easy and automatic verification of user's input
  - Offering easy input/model sensitivity analysis features.
- The design and implementation of Nessie, a new tool for performance estimation that explicitly represents the functionality and platform and proposes the following features:
  - Automatic hierarchical mapping of a "functionality" onto a "platform" based on flexible allocation, scheduling and routing policies
  - Estimation of user-defined multicriteria variables to drive the design decision process
  - Flexible design space exploration policy relying on an interface gathering performance criteria and degrees of freedom describing all the

design choices in a generalized way (platform, functionality and mapping).
  - Integration of Yeti to enable fast performance estimation based on flexible closed-formed models
- The demonstration of Nessie's and Yeti's proper working, performance evaluation features and abilities to solve a first realistic set of design cases including:
  - The extension of a multi-processor model for computation power estimation taken from the literature
  - The optimization of clock frequency and bandwidth based on a stage delay model built from scratch
  - The successful power consumption modeling of an AVC decoding application running on an MPSoC platform and the reproduction of the results with an average error lower than 1%.

As a conclusion, if we have already explored some preliminary case studies to prove the proper working and practical interest of both our tools, some additional validations on larger case studies taken from industrial designs are still required to prove the concept and are currently being investigated in the context of an ongoing PhD. The main benefit of our approach certainly lies the ability to quantitatively test and compare different design solutions by simultaneously changing functional, platform and mapping related parameters. Estimating the impact on performances of several architectural or functional variants thus becomes incredibly fast and easy at the expense of a relatively modest modeling effort. If many opportunities to reduce this modeling cost (like the introduction of a GUI or the automatic extraction of modeling information from existing design descriptions) have been presented in our analysis of the future work, Nessie and Yeti intentionally encourage this process of modeling since we believe this is a key to performance estimation.

On a personal level, I believe the main contribution of this thesis lies in the gathering of existing concepts with new ideas of my own to build, what is to my knowledge, one of the most flexible framework for design space exploration.

Appendix A

A complement to Yeti implementation

A.1 The shunting yard algorithm

A.1.1 Introduction

The shunting yard algorithm has been developed in the early seventies by computer scientist Dijkstra. This algorithm, named after its creator because of its similarities with a railroad shunting yard, is mainly used to convert infix mathematical expressions into *RPN*(Reverse Polish Notation) and is nowadays widely used in any common calculator. However another interesting application of this algorithm is the ability to turn a closed-formed expression into an *AST*(Abstract Syntax Tree) whose nodes are operands and directed hyperedges represent operators. In the simple tree example of Fig.2.7, we can see how is represented the expression $A = B + C * 78$ where Int_1 is an intermediate node added to make the tree structure valid. Thanks to this representation, a closed-formed expression can evaluated at run-time without having to make it hard-coded.

A.1.2 Available operators

All yet available operators are defined in table A.1. They are classified by the following criteria:

- *Operator name*
- *Operator type* refers to the number of operands required for this precise operator: both unary and binary operators are available
- *Symbol* is the string associated with an operator and defines how it will appear in a closed-formed expression. It is interesting to mention that symbol - is used both for the binary *subtraction* and unary *opposite* op-

Operator name	Type	Symbol	Precedence	Associativity	Exception
Addition	Binary	+	1	Yes	No
Subtraction	Binary	-	1	No	No
Multiplication	Binary	*	2	Yes	No
Division	Binary	/	2	No	Yes
Logarithm	Unary	log	3	No	Yes
Opposite	Unary	-	4	No	No
Exponent	Binary	^	3	No	No

Table A.1: Table of available operators classified by name, type(number of operands required), precedence value, associativity and possibility of mathematical exception

erator. How confusing it may seem, we wanted to keep this symbol for both operators because of their common use.

- *Precedence* refers to the priority for the evaluation order of an expression: considering two successive operators (without brackets), the one with the highest precedence value will be evaluated first. For instance, the expression $a = b + c * d$ requires $c * d$ to be evaluated first rather than $b + c$ because of respective values of $+$ and $*$ operators (see table A.1)
- *Associativity* of the operator
- *Exceptions* may occur during expression evaluation for the logarithm (negative basis or argument) and division (null denominator). These exceptions must, of course, be detected at run-time.

A.1.3 The algorithm

Algo.7 presents in details the way the shunting yard algorithm is performed. To illustrate it, we took the expression represented by the textual string A.1. The different steps are detailed in Fig.A.1 while the explanation follows.

$$a = b * (c - \log(83)) + 5 \tag{A.1}$$

1 The *OperandStack* and the *OperatorStack* are created. At the beginning, they are empty. The string is then parsed: the first expected element is the destination operand a and then the $=$ operator. At this point, everything is initialized for the algorithm to start: the string is parsed and each element is read and classified into two categories *operands* and *operators*.

2 Operand "b" is parsed and pushed onto the *OperandStack*

3 Operator "*" is parsed and pushed onto the *OperatorStack*

4 Element "(" is parsed and pushed onto the *OperatorStack*

5 Operand "c" is parsed and pushed onto the *OperandStack*

Algorithm 7 Shunting Yard Algorithm turning an infix into an AST representation

1: Define *OperatorStack*
2: Define *OperandStack*
3: **while** String is not empty **do**
4: Parse next *element*
5: **if** *element=operand* **then**
6: Push *currentOperand* onto the *OperandStack*
7: **else if** *element=Operator* **then**
8: **if** *currentOperator* precedence > *OperatorStack*[*lastElement*] precedence **then**
9: Push *currentOperator* onto *OperatorStack*
10: **else**
11: Pop the last operator off the *OperatorStack* and the required number of operands off the *OperandStack*
12: Push this newly created operand onto the *OperandStack*
13: **end if**
14: **else if** *currentOperator* = ")" **then**
15: **while** *OperandStack* ≠ "(" **do**
16: Pop the last operator off the *OperatorStack* and the required number of operands off the *OperandStack*
17: Push this newly created operand onto the *OperandStack*
18: **end while**
19: **end if**
20: **end while**
21: **while** *OperandStack* and *OperatorStack* aren't empty **do**
22: Pop the last operator off the *OperatorStack* and the required number of operands off the *OperandStack*
23: Push this newly created operand onto the *OperandStack*
24: **end while**

Ensure: The *OperatorStack* must be empty and the *OperandStack* must only contain the final operand value

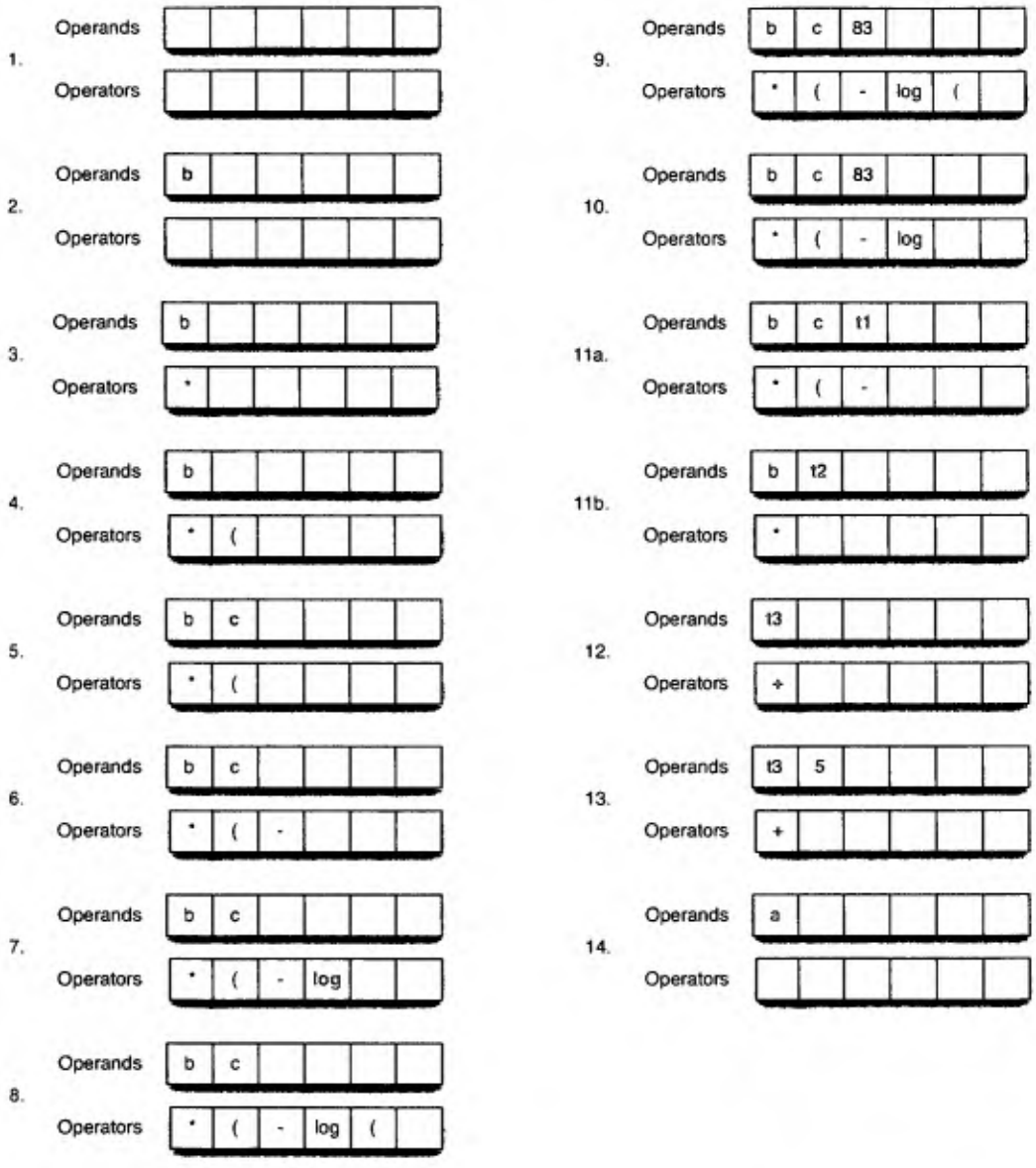

Figure A.1: Detailed procedure for the extraction of the AST out of the textual string $a = b * (c - \log(83)) + 5$ using the shunting yard algorithm

6 Operator "-" is parsed and pushed onto the *OperatorStack*

7 Operator "log" is parsed and pushed onto the *OperatorStack*

8 Operator "(" is parsed and pushed onto the *OperatorStack*

9 Operand "83" is parsed and pushed onto the *OperandStack*

10 Operator ")" is parsed and all the operators are popped off the *OperatorStack* until a "(" is found. As "(" is at the top of the stack, it is only popped off and nothing else is done.

11 Operand ")" is parsed:

(a) Operator "log" (unary operation) is popped off the *OperatorStack*, operand "83" is popped off the *OperandStack* and the resulting operand $t1 = \log(83)$ is created and pushed back onto the *OperandStack*

(b) Operator "-" is popped off the *OperatorStack* while operands "t1" and "c" are popped off the *OperandStack* and the resulting operand $t2 = c - \log 83$ is pushed back onto the *OperandStack*

Finally the creation of operands ends with "(" being popped off the *OperatorStack*.

12 Operator "+" is parsed, its precedence is compared to the one of operator "*" on top of the *OperatorStack*: because it is smaller, it means that operator "*" has to be executed prior to "+". A new operand $t3 = b * t2$ is created and pushed onto the *OperandStack*.

13 Operand "5" is parsed and pushed onto the *OperatorStack*

14 There is no element to parse anymore from the expression string: all remaining operators are popped off *OperatorStack*. Operand $t4 = t3 + 5$ is created and pushed onto the *OperandStack*. As we can see, the *OperatorStack* is empty as it should be and the *OperandStack* only contains the destination operand (therefore renamed with the operand destination name a memorized at the beginning of the algorithm).

A.2 Mutual class inclusion

Many times during our C++ implementation we have run into the same compiling problem related to *mutual class inclusion*. To overcome once and for all this recurrent and naughty issue that prevents the compiling from succeeding, we present here a systematic solution to solve it.

This particular problem occurs when two classes define mutually dependent instances of the other class in their attributes. The code sample of List.A.1 illustrates this particular issue. Two classes A and B are defined: A contains an instance of class B and B an instance of A. In a classical way, one would tend to include the header file definition of class A (B) inside class B (A) so that they know about each other (line 2 and 19 of List.A.1). However this will cause the compilation to fail because the compiler will start reading *Class_A.hpp* (line

Figure A.2: Graph representation of a mutual inclusion class problem

2), jump to *Class_B.hpp* definition (line 19), jump to *Class_A.hpp* and so on until it gives up. The problem is that we have created a deadlock due to the mutual dependence of classes A and B: A requires B to be completely defined before going on but B also requires A to be completely defined before going on. If we represent this dependence between classes as a loop, the solution simply consists in breaking this cycle so that one of the two classes can be entirely defined before the other one. To do so, we modify *Class_A.hpp* by replacing line 2 by a prototype of the class B (see List.A.2). This mechanism provides the compiler with a temporary definition of class B so that it can proceed with the compiling process of class A and afterwards finish the compilation of class B: the reference to the temporary class B is finally overwritten with the new definition during linking.

Since the solution to the mutual class inclusion issue is to find a way to break the mutual class dependence, it can be represented using a graph where nodes are classes and edges represent the dependences: solving our problem then just consists in breaking graph cycles (Fig.A.2). If the graph transposition can seem a bit complex to solve a problem with so few classes, it comes at hand for multiple class dependences where complexity becomes quickly overwhelming. In a funny but ironic way, we already solved that cycle detection problem using our previous behaviour hypergraph exploration algorithm (see Sec.2.5.3) ...however our algorithm was not usable yet because we precisely needed to answer the mutual class inclusion problem to get $YETi^3$ compiled. The UML class diagram of the whole framework (see Fig.2.6.1) shows a lot a mutually referring classes: hence we had to graph all the classes and manually break the cycles to get our framework compiling.

An alternative method to graph cycle breaking is *incremental compiling*. This method, somewhat more brutal but paradoxically very subtle, can be combined to the previous one when graph complexity becomes too overwhelming. It simply consists in commenting the mutually referring part of the code of one of the class that triggers a mutual inclusion problem: since the dependence is broken (actually hidden inside the comments) the compiler and the linker get the work done properly. Afterwards we release the previously commented portion of the code, recompile the entire code ...and it works! This can be explained through the fact that only modified classes need to be recompiled while the others have already been linked hence their methods/attributes are already part of the symbol table. This incremental compiling technique may

Listing A.1: Essai

```
1  //Definition of "Class_A.hpp"
2  #include "Class_B.hpp"
3  class A
4  {
5  public:
6          A(B* _myB);
7  private:
8          B* myB;
9  };
10
11 //Definition of "Class_A.cpp"
12 #include "Class_A.hpp"
13 A::A(B* _myB)
14 {
15         myB = _myB;
16 }
17
18 //Definition of "Class_B.hpp"
19 #include "Class_A.hpp"
20 class B
21 {
22 public:
23         B(A* _myA);
24 private:
25         A* myA;
26 };
27
28 //Definition of "Class_B.cpp"
29 #include "B.hpp"
30 B::B(A* _myA)
31 {
32         myA = _myA;
33 }
```

seem a bit magic but works perfectly when the previous method gives up.

Listing A.2: Essai

```
1 //Definition of "Class_A.hpp"
2 class B;
3 class A
4 {
5 public:
6         A(B* _myB);
7 private:
8         B* myB;
9 };
```

Appendix B

XML for Nessie/Yeti data support

B.1 Some insight into XML

B.1.1 Introduction

XML (eXtended Markup Language) is a W3C standard defining a generic syntax used to mark up data with simple, human readable tags. Usually associated with web technologies, XML is also a powerful language for anyone wanting to define its own file structure to manipulate data. An example of XML document is shown in List.B.1 representing a simple behaviour element containing two empty relations element with their own attribute.
In the context or our work, XML offers a lot of advantages compared to user-defined file structures:

- Standard interfaces (namely SAX and DOM) have been defined for object-oriented languages to manipulate XML documents easily and extract data from the structure.
- XML grammar is completely independent of the C++ framework so that we can easily verify the validity of a document using an external *validating parser*
- Since it is a standard, many useful surrounding technologies have been defined and many tools are available to manipulate and create XML based content
- Turning XML files into browsable content is very easy and offers very promising opportunities for the possible perspective of building an online model repository.

The main drawback of using XML is the size of the resulting documents. Producing human-readable documents implies a lot of structural information to establish a clear data structure hence a lot of unnecessary and redundant information. So in terms of memory occupancy, XML documents score very poorly

Listing B.1: Example of an XML code containing one behaviour element and two nested relation elements with their name attribute

```
1 <behaviour>
2         <relation name="Rel1"></relation>
3         <relation name="Rel2"\>
4 </behaviour>
```

compared to raw data files which can become a problem when the amount of information increases.
There are many XML related technologies and describing them in details would be a waste of time: for the purpose of our talk we will only introduce the very essential notions that will be used through this work:

- *XPath* is a language able to select parts of an existing XML document by performing searches on the tag structure and the contained data. In other words this is very similar to querying the document (like in a database) and particularly useful to sort or extract data from large documents.
- *Xinclude* is a technology recently developed by the W3C to combine different independently valid documents to form a new valid file. This is especially useful when we have different existing documents whose content must be included into another XML file: it avoids to unnecessarily duplicate document parts hence to maintain data consistency.

XML document verification

If XML enables the specification of almost any type of data, some verification mechanisms are obviously required to track structural and content errors.
Two types of verifications have been set up to respectively check the syntax and the semantic of an XML instance:

- First a document is said to be *well-formed* if the structure respects a few constraints (all data have to be put inside tags, no closing tag is allowed if it isn't preceded by its corresponding opening tag, nested tags must be ordered properly, etc.). This first *syntaxic* verification is performed to check whether or not the file is an XML document regardless of its data.
- Second the semantic of the XML file is verified to make sure that the document is *valid* against a defined grammar. The data types, elements occurrence and number, attributes number are verified to guarantee that the document has a meaning as intended by the grammar.

There are many different technologies to describe the grammar of an XML document: we only considered the two W3C standards to guarantee as far as possible tool compatibility.

- DTD (Document Type Definition) is the first W3C standard grammar. It has the particularity to be very concise allowing to specify in a few lines

the structure of a document at the expense of being very cryptic. Since DTD's are quite old now, they suffer from a lot of limitations: very few possible constraints on data (no value enumeration, no predefined types) and no element occurrence restrictions.

- Schema is the most recent W3C standard grammar developed to cope with all the limitations of DTD's. While extending the data and structure constraints capabilities, XML schema is much more strict and deal with the lack of DTD expressivity. However, schemas pay their price for extended functionalities by being much more verbous and complex than DTD's especially because a schema is in itself a valid XML document.

We finally chose schemas rather than DTD's for their enhanced grammar specification capabilities particularly regarding the data constraints. To simplify the implementation of the schema, we used XML Oxygen 8.0 that provides the programmer with a graphical environment to build its schema's along with a lot of other available XML technologies.

B.1.2 Implementation

Once the schemas have been defined and XML files instantiated, we have to convert the contained information into C++ classes to make it usable inside our framework and produce afterwards XML result files.

Reading an XML file

To read the information from the files, the best solution is to take profit of the standard API interfaces named DOM3 and SAX2:

- DOM parses the whole document and completely stores it in the memory: when this is done, the user can invoke methods to extract the desired data out of it by querying the memorized structure.
- SAX works very differently: it parses the document step by step and each time an XML related entity is identified (opening tag, attribute, closing tag etc.) an event occurs calling the corresponding method. The programmer has just to implement what has to be done when such a method is invoked.

DOM is probably easier to use but introduces a certain memory overhead (that can be as big as the document itself) while SAX consumes very low memory. Since we have to parse the complete document, SAX becomes the obvious choice. It is however important to note that depending on the object structure, the order of appearance of each XML element or complex type may greatly complicate the programmer's task forcing him to store important parts of the XML file if some further information are required in order to build the objects. Hence the order of tag appearance has been chosen to simplify

the programming while keeping it simple for the user to enter the XML file information.
XML schemas are parsed thanks to the XercesC 2.7 API [1] using SAX2 to extract data. Despite being one of the widest spread and most complete XML parser for C++, Xerces does not implement all XML features and technologies[1]. For instance, Xinclude which enables easy data inclusion from one XML document into another one is currently not supported. This feature is however particularly well adapted to our purpose: Yeti relations might be defined in separate documents and a behaviour could then just consist in an enumeration of XML documents containing relations. Apart from making behaviour building much more easy than relation cut and paste operations it further avoids redundancy of information and potential errors. To cope with that lack, we have implemented our own Xinclude processor thanks to a simple Perl script that replaces all occurrences of Xinclude tags by their linked XML file content while preserving the validity of the resulting file.

Creating an XML document

XML documents an be created in two different in the context of Yeti and Nessie: either manually (input files for the program) or automatically (as a result of the model estimation).
Creating documents manually may quickly become boring for the user due to the complex structure that may be needed to support the data. A nice way to solve this problem is to use an XML editor that guides the user when building the document thanks to the schema definition (embedded feature in XML Oxygen). For a new document, the editor automatically inserts the minimal tag structure required to have a valid document and leaves the user with the only task of entering the data. To complete the document by adding elements where it's necessary, the editor proposes interactive choices to the user and then performs auto-completion of the structure. With such a tool, the user doesn't need to enter any single character related to the structure and only fills in the document with useful data while respecting the grammar since it is added by the editor itself.
Creating documents automatically (using C++ in our case) has to be done using string manipulation because, contrarily to document reading, no standard API has been defined to do that. Anyway this task remains much more simple than the XML file reading operation even with the help of SAX2 and DOM3 dedicated API's.

[1] At the time we programmed Yeti, only the 2.7 XercesC version was available but, with the very recent release of the 3.0 version, some new XML technologies have been introduced inside the library.

B.2 Yeti schema's

Introduction

Each element/attriubute meaning and content are explained and eventual restrictions on data are presented. We refer to elements and attributes using the standard Xpath language with a *path relative to the root element* of the schema file. All the elements are included into the same namespace *http://beams.ulb.ac.be/avdbiest* using the prefix *yeti* in schema's. Besides the path of the element and its description, we also provide information about the type since we defined a lot of types for our own usage that may sometimes be very particular. To guarantee the validity of the document, we made an extensive use of *xs:key*, *xs:unique* and *xs:keyref* elements when it was possible. However to supply for the limitations of the schema's data constraints mechanism, we need to differ part of the verification until parsing at run-time. That's why we clearly define during element description which data constraints violations will trigger run-time errors or schema validation errors.

B.2.1 Schema's organization

In this section, we describe the different schema's to provide the user with all the necessary details to understand the input and output XML files used inside Yeti. In order to simplify the conversion of XML files into Yeti classes, we wanted to keep our 3-layered hierarchical model representation inside our XML schema description. Therefore we encapsulated each of the Yeti element (parameter, generic rule, relation, behaviour, etc.) inside a *<xs:complex type>* schema element composed out of a collection of more simple types (like it it the case in each class). We can afterwards use each XML element by including its definition (using the *<xs:include>*) inside a new schema to build new and more complex types. Besides the improved file organization, this mechanism enables the propagation of the changes made to one schema to all the schema's that use it so that type consistency is always preserved.
The *YETi*[3] schema design is described in Fig.B.1[2] where each box represents a schema defining a particular element type while arrows link a schema to another one using it. At the top of the figure we may recognize our three basic model entities separately modeled in the XML schema architecture (*associationType* approximately corresponds to the generic rule). Let us also notice that only top schema files are instantiated types (no "type" at the end of the name) while all other schemas contain complex type elements. That's because the *<xs:incldue>* tag simply copies the content of the referenced file (with some minor changes in header tags): the included schema can then be instantiated in the higher level schema.

[2]Unlike UML for object-oriented languages, schema don't have a standardized representation format thus the way we are describing the schema organization is of our own and has not to be mistaken for a standard

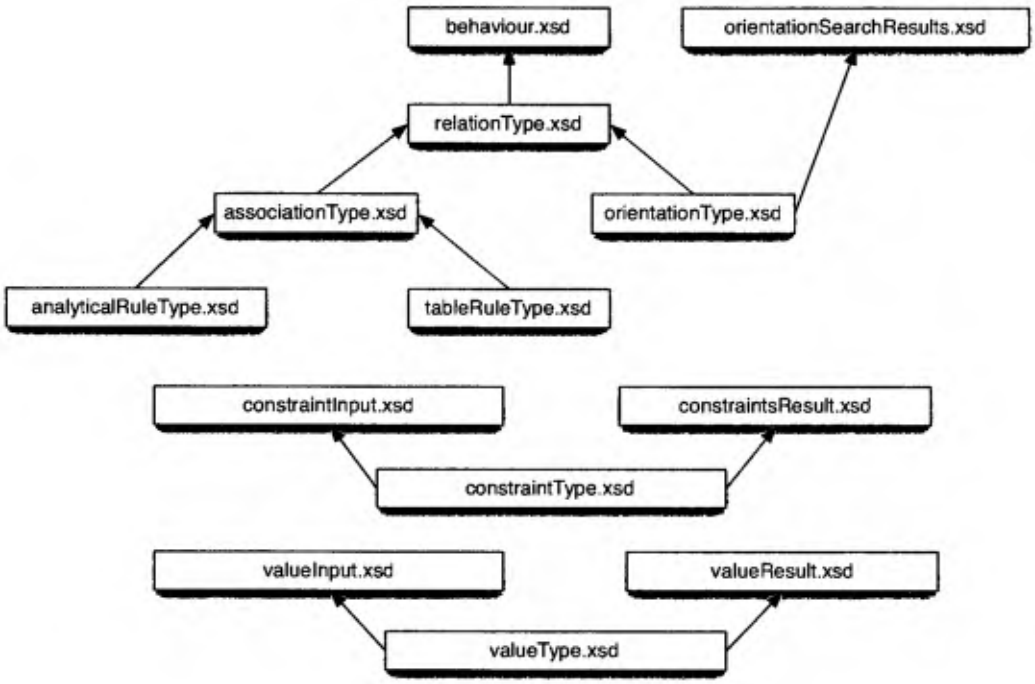

Figure B.1: Schema organization for the $YETi^3$ framework

B.2.2 Schema's description

In the following sections, we describe one by one of the different schemas, explain their role and detail each element and the meaning of the data that it contains.

behaviourType.xsd

The behaviour schema (Fig.B.2) contains all the useful data's required to build a behaviour class. It is composed out the following elements:

- ./@name (*yeti:nonEmptyString*): name of the behaviour
- ./relationList (*xs:complexType*): list of all the relations present in the relation
- ./relationList/relation (*xs:complexType* named relationType): description of the content of a relation. A key guarantees the uniqueness of *./relation@name* attribute values.
- ./orientationList (*xs:complexType*): contains the list of all possible orientations for the behaviour
- ./orientationList/orientation (*xs:complex type* named orientationType): orientation of the behaviour

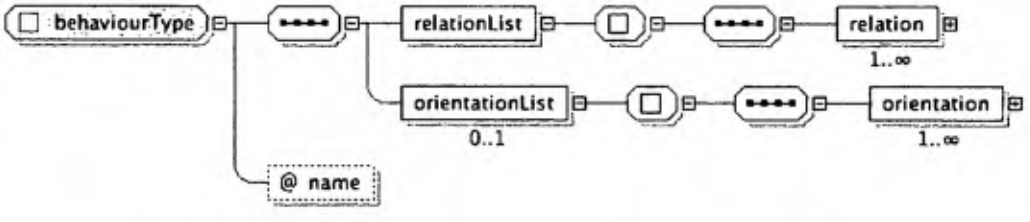

Figure B.2: Schema for the behaviour type - behvaviourType.xsd

orientationType.xsd

The orientation related schema (Fig.B.3) contains all the information required to specify the orientation of all the relations of a given behaviour. $YETi^3$ triggers a run-time error whenever the XML parser comes across a parameter/relation name that doesn't exist in the considered behaviour.

- ./@orientationName (*yeti:nonEmptyString*): name of the orientation
- ./inputParameter (*yeti:nonEmptyString*): the set of all these elements represent all the input parameters of the current behaviour orientation

- ./outputParameter (*yeti:nonEmptyString*): the set of all these elements represent all the output parameters of the current behaviour orientation
- The sequences of ./relationName (*yeti:nonEmptyString*) and ./oriented-Parameter (*yeti:nonEmptyString*) elements define for each relation (relationName) the corresponding parameter (orientedParameter) that it will be associated with for the current orientation.

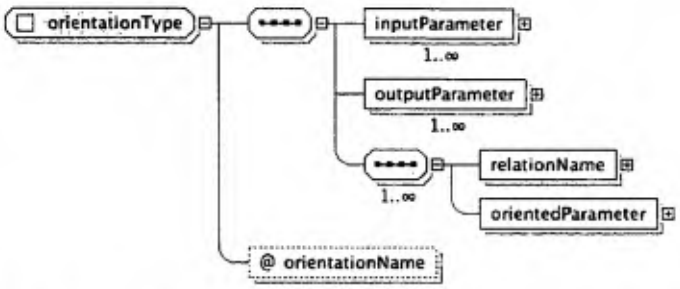

Figure B.3: Schema for the orientation type - orientationType.xsd

relationType.xsd

The relation schema (Fig.B.4) describes the structure of a relation and provides the support for different types of generic rules. A relation is composed out of the following elements:

- ./@name (*yeti:nonEmptyString*): name of the relation
- ./content/parameterList (*xs:complexType*): list of all the external parameters of the relation (except the constant parameters which may absolutely not be referred in that list, see Sec.2.5.1).
- ./content/parameterList/parameter (*yeti:nonEmptyString*): name of the parameter
- ./content/association (*xs:complexType* named associationType): defines the possible associations for the previously defined parameters. At least one association must be defined (otherwise the relation would not be executable) and the maximum number of possible associations equals the number of defined parameters (since each parameter can be associated with only one single generic rule). $YETi^3$ triggers a run-time error whenever the XML parser comes across a parameter name that doesn't exist in the considered relation.

associationType.xsd

The association schema (Fig.B.5) defines the association for one parameter and enables a kind of inheritance mechanism allowing the user to mix heterogeneous

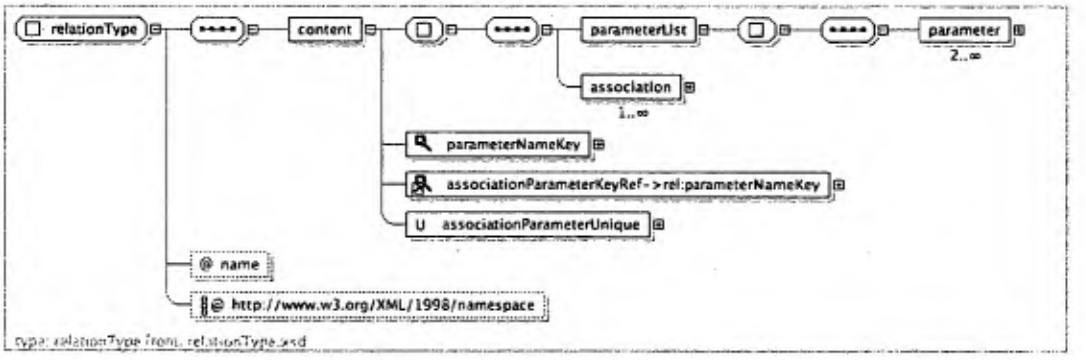

Figure B.4: Schema for the relation type - relationType.xsd

generic rules inside the same relation.

- ./associatedParameter (*yeti:nonEmptyString*): name of the parameter. $YETi^3$ triggers a run-time error whenever the XML parser comes across a parameter name that doesn't exist in the considered relation.
- The association points towards a generic rule that may either be an analytical rule (element ./analyticalRule) or a table-based rule (element ./tableRule).

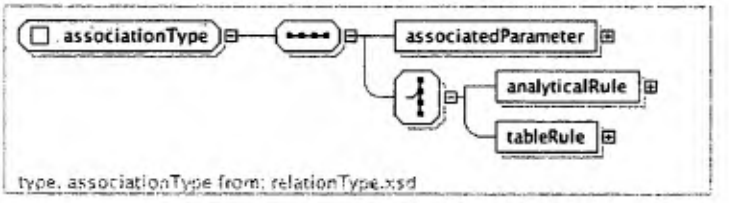

Figure B.5: Schema for the association type - associationType.xsd

analyticalRuleType.xsd

The schema (Fig.B.6) related to analytical rules specifies all the information required to build them based either on a simple textual string capturing a closed-formed expression or on a collection of basic operations forming together the final analytical rule.

- ./collection (*xs:complexType*) defines a collection of basic analytical elements that can be gathered to form the complete analytical rule.
- ./collection/analyticalElement (*xs:complexType*) describes basic operation elements (see Sec.A.1.2) forming a tree and may contain either:

1 A single root parameter (./collection/rootParameter) in case of a leaf parameter

2 A unary operation is represented by a root parameter, one sub-parameter (./collection/subParameter) and one unary operation symbol element (./collection/unarySymbol). This unarySymbol element is a string (yeti:unaryOperationSymbolType) restricted to the "log" (logarithm operation) and "-" (opposite operation) values so that any other value will trigger an XML validation error.

3 A binary operation is represented by a root parameter, two sub-parameters (./collection/subParameter1 and ./collection/subParameter2) and one binary operation symbol operation (./collection/binarySymbol).
This binarySymbol element is a string (yeti:binaryOperationSymbolType) restricted to the "+", "-", "*", "/" and "^" (power operation) values so that any other value will trigger an XML validation error.

- ./expression (*yeti:nonEmptyString*) defines a textual expression representing the analytical rule. Although XML schema features *regular expression* verification through the use of facets, it is not of great use in our case since the expression has a recursive structure: verification is thus delayed until run-time.

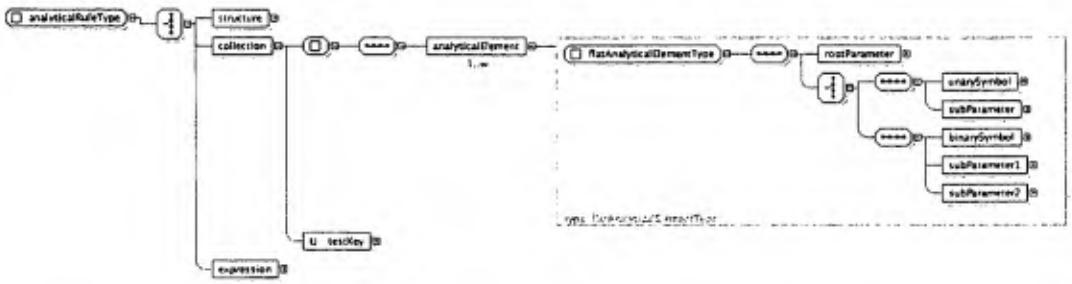

Figure B.6: Schema for the analytical rule type - analyticalRuleType.xsd

tableRuleType.xsd

The schema (Fig.B.7) related to table rules defines a recursive structure able to capture the structure of a multi-dimensional table (with a variable number of dimensions) and its content.

- ./@rootParameter (*yeti:nonEmptyString*) contains the name of the output parameter of the table
- ./tableListEntry (*xs:complexType*) specifies all the information relative to the input parameters of the table and is composed out of a succession of:

1 ./tableInputParameter (*yeti:nonEmptyString*) contains the name of the input parameter

2 ./parameterValue (*xs:float*) defines all the possible input values associated with the corresponding input parameter. The table will only be defined for these values and any attempt to use the table with input values than the predefined ones will result in run-time error messages.

- ./tableStructure (*xs:complexType* named tableElementType) is the root element for the table structure and sets up the recursive structure by defining dimension by dimension the table (see Sec.2.5.1 for more details). It is composed out of the following elements:

  1 ./tableStructure/tableParameterName (*yeti:nonEmptyString*) contains the parameter name of the currently defined table dimension

  2 ./tableStructure/valueList (*xs:complexType*) defines for each value of the considered parameter the substructure of the table. The value is defined by the ./tableStructure/valueList/parameterIndex (*xs:positiveInteger*) element corresponding to the index of the value in the ./tableEntry/parameterValue elements sequence. With each of these values is associated either a ./tableStructure/valueList/tableValue element (*xs:float*) if we reached the last table dimension or a ./tableStructure/valueList/subTable (*yeti:tableElementType*) if we have further table dimensions to define.

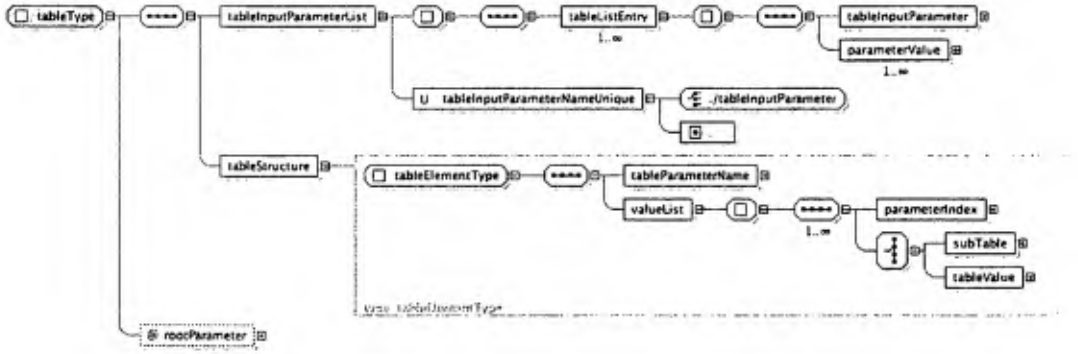

Figure B.7: Schema for the table type - tableType.xsd

valueType.xsd

This schema (Fig.B.8) defines the input values associated with a parameter using either lists, sweeps or single values:

- ./@parameterName (*yeti:nonEmptyString*) contains the name of the parameter for which input values are defined
- ./single (*xs:float*) defines a single float value for the parameter
- ./sweep (*xs:complexType*) complex type contains three different float (*xs:float*) called start (./sweep/start), step (./sweep/step) and stop (./sweep/stop). $YETi^3$ will automatically generate based on these elements a list of values starting from value start till value stop with an incremental step (if the difference between the next to last value and the stop value is smaller than the step, the last value is assigned with the stop value).
- ./list (*xs:complexType*) element contains a list of ./list/item (*xs:float*) elements, each one defining a float value.

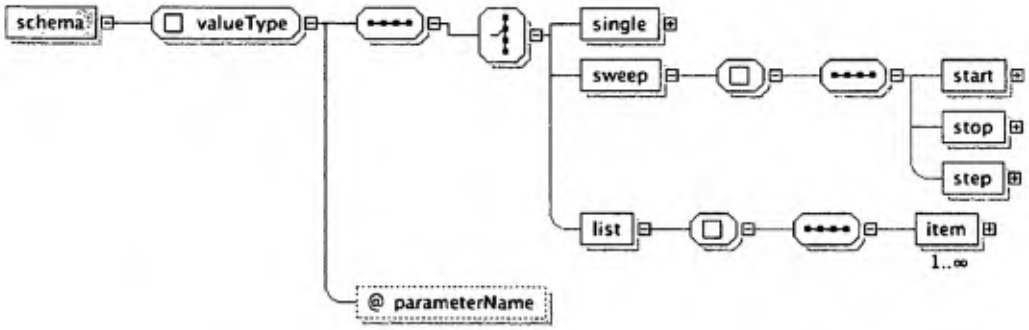

Figure B.8: Schema for the value type - valueType.xsd

constraintsType.xsd

This schema (Fig.B.9) defines the input constraints associated with a parameter. Apart from the ./@parameterName (*yeti:nonEmptyString*) element, the constraints type contains the lower (./lowerBound) and upper (./upperBound) floating values bounds.

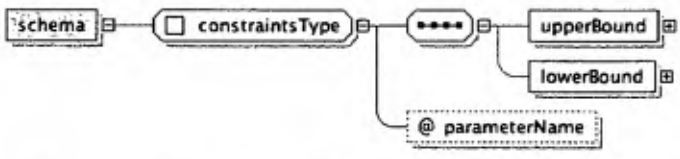

Figure B.9: Schema for the constraints type - constraintsType.xsd

valueInput.xsd

This schema (Fig.B.10) defines one element (./inputSet) containing a list of input values (./inputSet/values defined by yeti:valueType) for the ./inputSet/@behaviourName element.

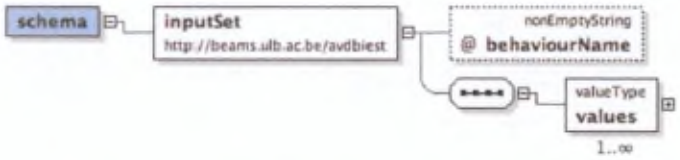

Figure B.10: Schema for the value input - valueInput.xsd

constraintsInput.xsd

This schema (Fig.B.11) defines one element (./inputSet) containing a list of input constraints (./inputSet/constraints defined by yeti:constraintsType) for the ./inputSet/@behaviourName element.

Figure B.11: Schema for the constraint input - constraintsInput.xsd

yetiScripting.xsd

This schema (Fig.B.12) defines the script structure of all the macro possible operations that can be carried out using $YETi^3$. The schema is composed out of a sequence the following elements:

- ./behaviourBuidling (*complexType* named behaviourBuildingType) builds a behaviour defined in file ./behaviourBuilding/@inputFile
- ./orientationChange (*complexType* named orientationChangeType) changes the orientation (based on input file ./orientationChange/@inputFile) of the previously built behaviour
- ./valueSimulation (*complexType* named valueSimulationType) triggers a value simulation based on the input file ./valueSimualtion/@inputFile and stores the result into the output file ./valueSimulation/@outputFile.
- ./constraintsSimulation (*complexType* named constraintsSimulationType) triggers a constraints simulation based on the input file ./constraintsSimu-

lation/@inputFile and stores the result into the output file ./constraintsSimulation/@outputFile.

- ./orientationSearch (*complexType* named orientationSearchType) launches a search based on the input file ./orientationSearch/@inputFile and stores the result as a series of valid orientationType elements in the output file ./orientationSearch/@outputFile.

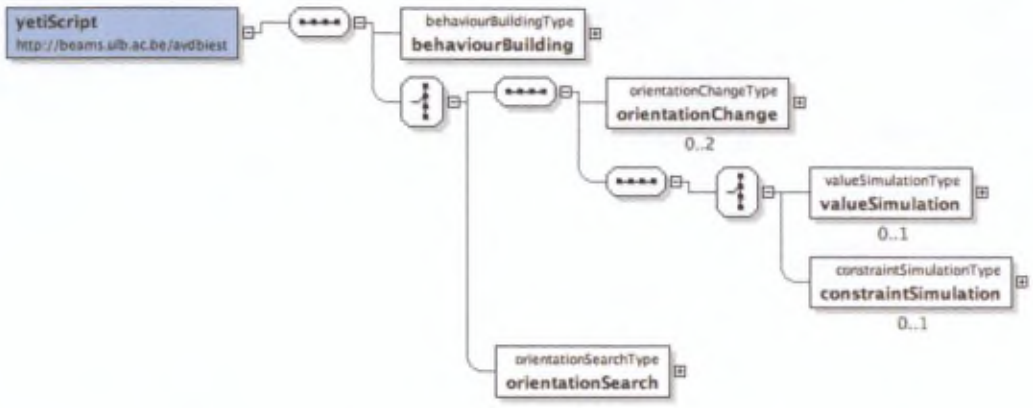

Figure B.12: Schema for the scripting input - yetiScripting.xsd

B.3 NESSIE schema's

B.3.1 Schema's organization

This section is dedicated to the description of the different XML schemas used for the representation of the input/output data inside our framework Nessie. This part of the dissertation should be used as a reference guide for the user wanting to understand the XML schema structure and the meaning of each element composing it: we *really* advise anyone interested in using Nessie to make a careful and thorough reading of this chapter.
The global architecture of Nessie XML schemas is pictured in Fig.B.13. Each schema is represented as a box filled with the corresponding file name while the arrows represent a dependence between two files. The prefix *nessie* defines the namespace "http://beams.ulb.ac.be/avdbiest" used for all the schemas of Nessie and their exported complex types.

B.3.2 Schema's description

In the following sections, we describe one by one of the different schemas, explain their role and detail each element and the meaning of the data that it

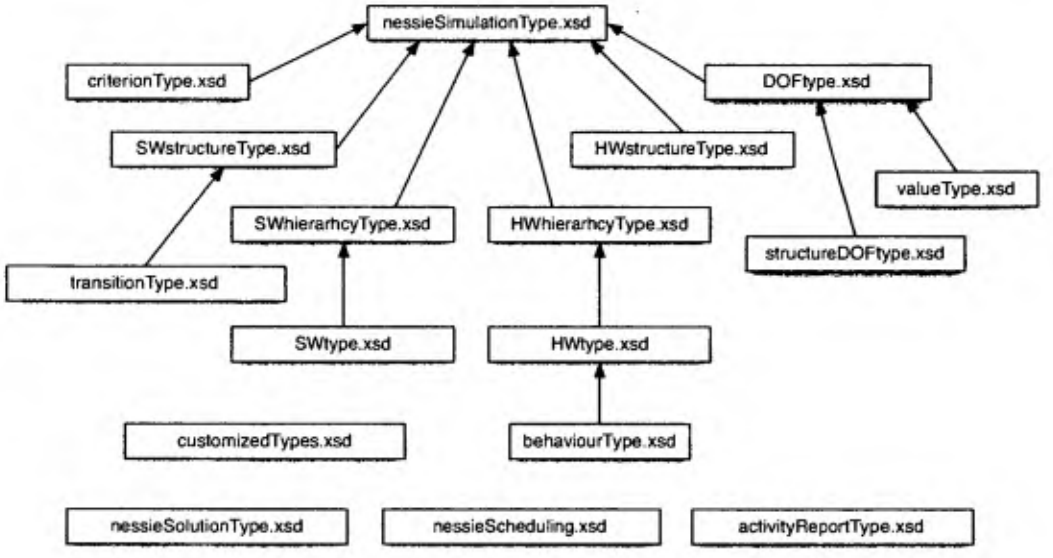

Figure B.13: Organization and dependence of XML schemas inside Nessie

contains.

customizedTypes.xsd

The *customizedTypes* schema (Fig.B.14) gathers different simple types extending predefined XML types. We may mention that this schema is widely used in all the other presented schemas of Fig.B.13 so that we intentionally omitted the arrows going to the different other schemas for the sake of clarity. The different types defined within our present schema are the following:

- nonEmptyString (base type is *xs:string*) restricts the possible values of this type to strings containing at least one element. Although it may seem a very simple and apparently useless restriction, it offers the benefit of generating errors during validation when a string element is left empty in an XML document: this is always a precious verification for distracted users.
- positiveInteger (base type is *xs:integer*) restricts the integer type possible values to positive (including zero) values. This type is mostly used for the identifiers of the different elements used in the other schemas.
- xmlFileType (base type is *xs:string*) limits the basis type possible values to strings with a non-empty prefix followed by the ".xml" extension. The prefix is restricted to the concatenation of literal characters and numbers that may be separated by the "/" symbol to define the file path of the

XML file.

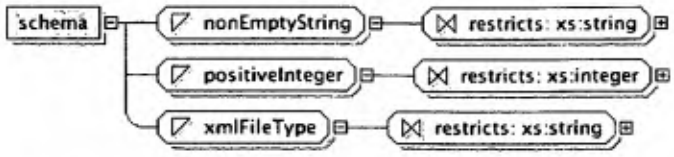

Figure B.14: Schema for the custom types for Nessie - customizedTypes.xsd

nessieSimulationType.xsd

The *nessieSimulationType* schema (Fig.B.15) is the top level XML file used to initialize a performance estimation run in Nessie:

- ./criteriaList (*xs:complexType*): list of all the criteria that need to be estimated for this simulation
- ./SWdescription (*xs:complexType*): description of the functional system part composed out of a *SWhierarchy* element (*nessie:SWhierarchyType*) defining the functional hierarchy and a list of functional structures (*nessie:SWstructureType*) each one associated with a particular functional primitive
- ./HWdescription (*xs:complexType*): description of the platform system part composed out of a *HWhierarchy* element (*nessie:HWhierarchyType*) defining the functional hierarchy and a list of platform structures (*nessie:HWstructureType*) each one associated with a particular platform primitive
- ./DOF (*xs:complexType*): contains all the degrees of freedom of the current simulation run.

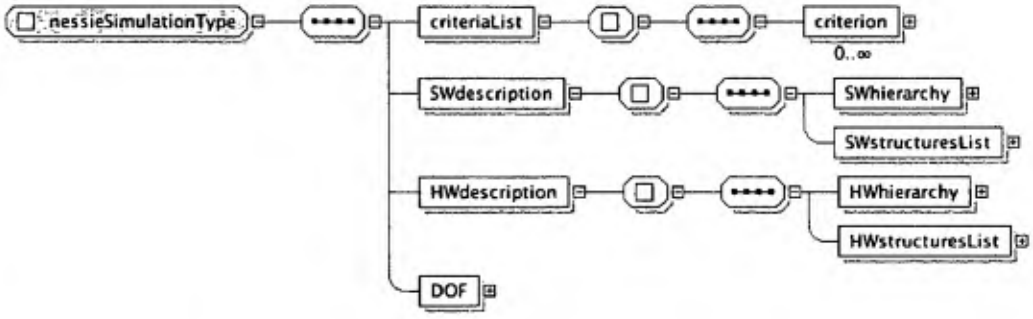

Figure B.15: Schema for the Nessie simulation type - nessieSimulationType.xsd

criterionType.xsd

The *criterionType* schema (Fig.B.16) defines a criterion by its name, its dependence over time and over the platform:

- ./@name (*nessie:nonEmptyString*): name of the criteria restricted to a non-empty string value
- ./@timeDependent (*nessie:timeDepenceType* simple string-based type with value restriction): defines the time dependence of this criterion by a string whose value is restricted to *none*, *maximum*, *integrate* and *additive*. Any other attribute value will trigger a validation error.
- ./@combinationRule (*nessie:timeDepenceType* simple string-based type with value restriction): defines how the values of this criterion are combined over different platform blocks by a string whose value is restricted to *additiveRule* and *maxRule*. Any other attribute value will trigger a validation error.

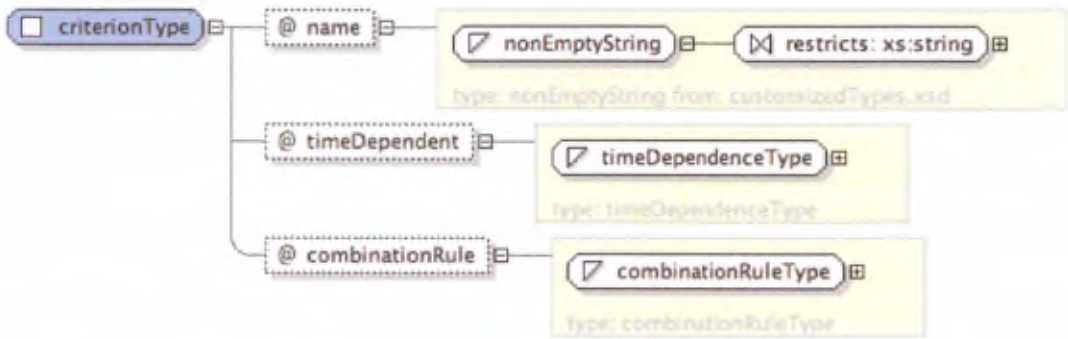

Figure B.16: Schema for the criterion type - criterionType.xsd

SWhierarcahyType.xsd

The *SWhierarchyType* schema (Fig.B.17 defines the hierarchy from the functional point of view by listing for each abstraction level the different associated functional primitives. The different elements are the following:

- ./abstractionLevel (*xs:complexType*): defines an abstraction level by an identifier (*@abstractionLevel*) of simple type *xs:integer* and by its different functional primitives (./abstractionLevel/SWsubTypes). Since each identifier must be unique and have an incremental value starting from 0, the framework performs a run-time verification and triggers an error if this condition is violated.

- ./abstractionLevel/SWsubTypes (*xs:complexType* named *SWtype*): collection of all the functional primitives defined within the current abstraction level.

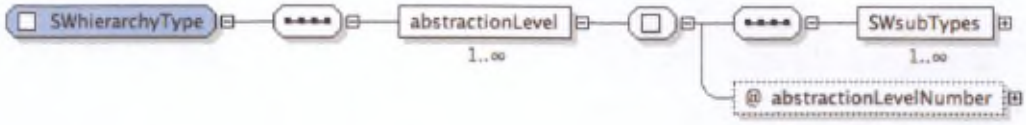

Figure B.17: Schema for the functional hierarchy type - SWhierarchyType.xsd

SWtype.xsd

The *SWtype* schema (Fig.B.18) defines a functional primitive and is composed out of the following elements:

- ./@ID (*nessie:positiveInteger*): identifier of the functional primitive unique among each functional abstraction level
- ./SWparameterList (*xs:complexType*): list of all the functional parameters used in the Yeti models for performance criteria.
- ./SWparameterList/SWparameter (*xs:complexType*): functional parameters are defined by a *nonEmptyString* simple type representing the name of the parameter that is extended by a boolean value attribute (*@localParameter*) defining if this parameter value will only yield for this particular functional primitive or will be used for all functional parameters with the same name whatever the primitive and abstraction level.
- ./dataOut (*nessie:positiveInteger*): size of the data resulting from the operation represented by this functional primitive

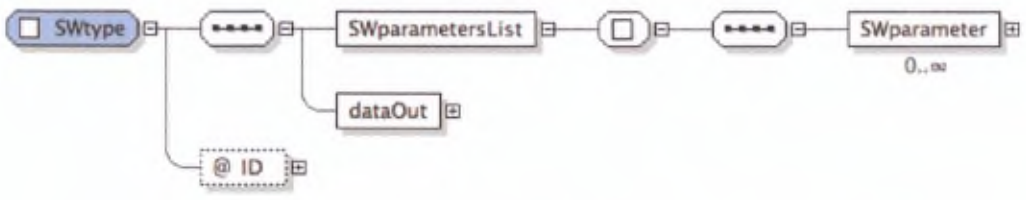

Figure B.18: Schema for the functional primitive type - SWtype.xsd

SWstructureType.xsd

The *SWstructureType* schema (Fig.B.19) defines a functional structure at a particular abstraction level for a given functional primitive. The complex type

SWstructureType begins with a *xs:choice* element to allow the selection the model of computation that will be used for this functional structure (at the moment, only petri networks are available). The different elements composing the schema are the following:

- ./@SWtypeID (*nessie:positiveInteger*): the functional primitive identifier for which the functional structure is defined
- ./@abstractionLevel (*nessie:positiveInteger*): the abstraction level of the functional primitive ./@SWtypeID for which the functional structure is defined
- ./petriNetwork (*xs:complexType*): element describing a complete petri network based on a *netList* and *transitionsList* element.
- ./petriNetwork/netList (*xs:complexType*): contains all the places of the petri network that can either be a normal *petriNet* or a *dummyPetriNet* type.
- ./petriNetwork/netList/petriNet (*xs:complexType*): a place of the petri network
- ./petriNetwork/netList/dummyPetriNet (*xs:complexType*): a dummy node that simply forwards the token from its input to its output
- .//@ID (*nessie:positiveInteger*): defines the identifier of a place and guarantees its uniqueness thanks to a key relative to it. Additionally Nessie will check during object building if all identifiers have succeeding values starting from 0: if not fulfilled, this condition will trigger a run-time error.
- ./petriNetwork/netList/petriNet/@type (*nessie:positiveInteger*): defines the functional primitive identifier which the place refers to at the *immediately lower* abstraction level
- ./petriNetwork/transitionList (*xs:complexType*): defines all the transitions used in the petri network and linking the places. We may also mention that there is a unique element (*transitionUnique*) guaranteeing the uniqueness of the identifiers of the different transitions and that Nessie checks the identifier value succession at run-time as for the places. Additionally the key defined on the places (*petriNetsIDkey*) is used as a reference for a keyref element (*ouputPetriNetIDkeyref*) to guarantee that each place identifier used in the transitions actually refers to an existing place identifier. This mechanism maintains consistency between the place and transition definition and prevents the using from making references to undefined places in any transition.
- ./petriNetwork/transitionList/stratingTransition (*nessie:transitionType*): defines the starting transition for where output places will be triggered once the petri network is initialized. Using this particular transition, we define the initial token marking: this element is thus required in the schema whatever the number of transitions

- ./petriNetwork/transitionList/transition (*nessie:transitionType*): defines all the transitions except the starting transition.

transitionType.xsd

The *transitionType* (see Fig.B.20) defines a transition linking places in a petri network and is composed out of the following attributes and elements:

- ./@transitionID (*nessie:positiveInteger*): transition identifier with a unique value within the petri network (uniqueness checked at the petri newtork level see B.3.2).
- ./inputPlaceList (*xs:complexType*): list of all the input places
- ./outputPlaceList (*xs:complexType*): list of all the output places
- ./inputPlace (*xs:complexType*): defines one input place linked to the transition
- ./outputPlace (*xs:complexType*): defines one output place linked to the transition
- .//@placeID (*nessie:positiveInteger*): identifier of the input/output place used in the transition. The identifier value is checked by the XML parser to verify if it corresponds to an existing place defined in the network.
- .//@numberOfTokens (*nessie:positiveInteger*): number of tokens required in an input place to fulfill the transition condition or number of tokens generated to an output place after transition firing

HWhierarcahyType.xsd

The *HWhierarchyType* schema (Fig.B.21 defines the hierarchy from the platform point of view by listing for each abstraction level the different associated platform primitives. The different elements are the following:

- ./abstractionLevel (*xs:complexType*): defines an abstraction level by an identifier (@abstractionLevel) of simple type *xs:integer* and by its different platform primitives (./abstractionLevel/SWsubTypes). Since each identifier must be unique and have an incremental value starting from 0, Nessie performs a run-time verification to trigger an error if this condition is violated.
- ./abstractionLevel/HWsubTypes (*xs:complexType* named *HWtype*): collection of all the platform primitives defined within the current abstraction level.

HWtype.xsd

The *HWtype* schema (Fig.B.23) defines a functional primitive and is composed out of the following elements:

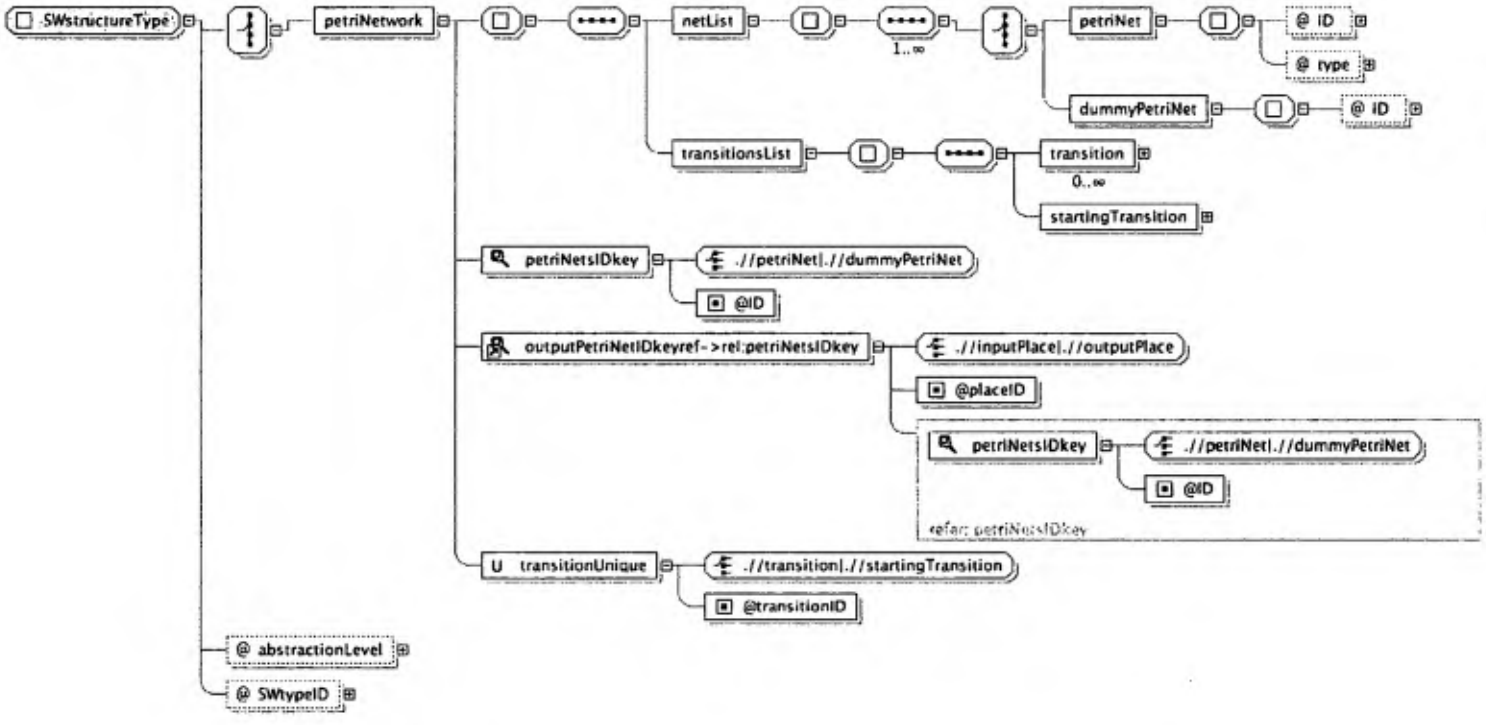

Figure B.19: Schema for the functional structure type - SWstructure.xsd

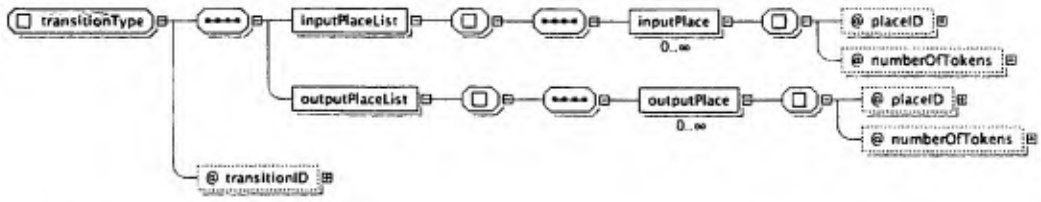

Figure B.20: Schema for the functional primitive type - SWtype.xsd

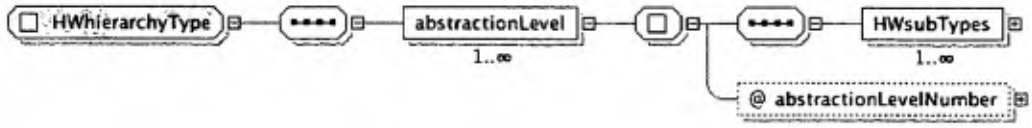

Figure B.21: Schema for the platform hierarchy type - HWhierarchyType.xsd

- ./@ID (*nessie:positiveInteger*): identifier of the platform primitive unique among each platform abstraction level
- ./HWparameterList (*xs:complexType*): list of all the platform parameters used in the Yeti models for performance criteria.
- ./HWparameterList/HWparameter (*xs:complexType*): a platform parameter is defined by a *nonEmptyString* simple type representing the name of the parameter and extended by a boolean value attribute (*@localParameter*) defining if this parameter value will only yield for this particular platform primitive or will be used for all platform parameters with the same name whatever the primitive and abstraction level.
- ./models (*xs:complexType*): contains all the models used for performance criteria estimation of the core states, the port states and for the different computation states associated to each compatible functional primitive.
- ./models/coreStateList (*xs:complexType*): list of all the core state models associated to the platform primitive
- ./models/coreStateList/coreStateModel (*xs:complexType*): this element defines the performance model for one particular core state and is composed out of two different elements:
  - A *behaviour* element (*nessie:xmlFileType*) defining the XML file name that describes the Yeti model characterizing the performance criteria. This element is optional since some states could remain undefined (for instance memorizing or transmitting states) meaning that a platform block based on this primitive could not enter this state and therefore be unable to provide the associated service. Nessie will verify during

the object building phase if the mandatory states are defined and trigger a run-time error otherwise.

- The *coreState* attribute defines the name of the state which the model is associated with. The type of this attribute is *nessie:coreStateType* restricting the string simple type to the "idle", "sleeping", "memorizing" and "transmitting" values.

To guarantee that each core state will be defined once and only once, we have set an occurrence constraint equals to 4 on the *coreStateModel* element combined to a unique element constraint on *@coreState* to avoid the definition of core states with similar names. The combination of those two constraints together with the definition of the restricted *coreStateType* involves that exactly four different core states with their exact names will be required for the document to be valid.

- ./models/compatibleSWlist (*xs:complexType*): list of the computing states associated with all the functional primitives compatible with the current platform primitive
- ./models/compatibleSWlist/computingModel (*xs:complexType*): performance model defined by the *behaviour* element (*nessie:xmlFileType*) for the associated functional primitive with identifier *@ID* defined at the same abstraction level as the current platform primitive
- ./models/IOstateModelsList (*xs:complexType*): list of the port state models associated with the platform primitive
- ./models/IOstateModelsList/IOstateModel (*xs:complexType*): this element defines the performance model for one particular port state and is composed out of a *behaviour* element (*nessie:xmlFileType*) and the attribute *@IOstateName* with a type (*nessie/IOStateType*) restricting the string type value to "inactive", "sending" and "receiving". Again the unique element constraint on *@IOstateName* and the occurrence constraint on *IOstateModel* set to three guarantees that all the three port states will be defined once and only once.
- ./models/transitionalTimeTable (*xs:complexType* named transitionalTimeTableType): a double entry table to specify the time delay required to jump from one core state to another.

The *nessie:transitionalTimeTableType* is defined inside the *nessie:HWtype* XML file and not in a separate file like other types usually are. The elements of this type are presented in Fig.B.22:

- ./startState (*xs:complexType*): contains the state from where the transition starts. The ./startState/@coreState attribute is a *nessie:coreGeneralStateType* adding "computing" to the list of acceptable strings defined by *nessie:coreStateType*
- ./startState/endState (*xs:complexType*): defines the state to which the transition ends and is composd out of a float value *xs:float* defining the

transitional time extended by a *@coreState* attribute defining the name of the ending state.

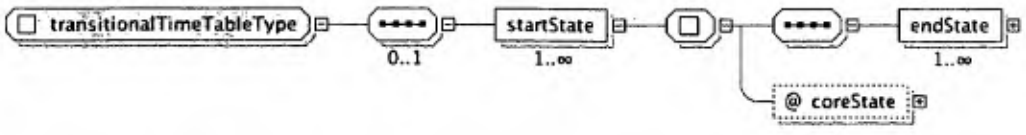

Figure B.22: Schema for the platform hierarchy type - HWhierarchyType.xsd

HWstructureType.xsd

The *HWstructureType* schema (Fig.B.24) defines a platform structure at a particular abstraction level for a given platform primitive. The different elements composing it are the following:

- ./@HWtypeID (*nessie:positiveInteger*): the platform primitive identifier for which the platform structure is defined
- ./@abstractionLevel (*nessie:positiveInteger*): the abstraction level of the platform primitive ./@HWtypeID for which the platform structure is defined
- ./localizedElements/HWstructure (*xs:complexType*): contains a description of the platform structure based on a collection of platform blocks connected by links.
- ./localizedElements/HWstructure/HWblockList (*xs:complexType*): contains the platform blocks (*HWblock* element) of the platform structure.
- ./localizedElements/HWstructure/HWblockList/HWblock (*xs:complexType*): platform block defined by its two attributes
  - *@ID* (*nessie:positiveInteger*): identifier of the block inside the platform structure
  - *@type* (*nessie:positiveInteger*): represents the platform primitive identifier at the immediately lower abstraction level.

  We also set a key element to guarantee the uniqueness of the identifier for each platform block.
- ./localizedElementsHWstructure/linkList (*xs:complexType* named linkType): list of all the links connecting the platform blocks together
- ./localizedElementsHWstructure/linkList/link (*xs:complexType*): element defining a link between two platform blocks and composed out of the following elements:
  - ./localizedElementsHWstructure/linkList/link/source (*nessie:positiveInteger*): source block identifier

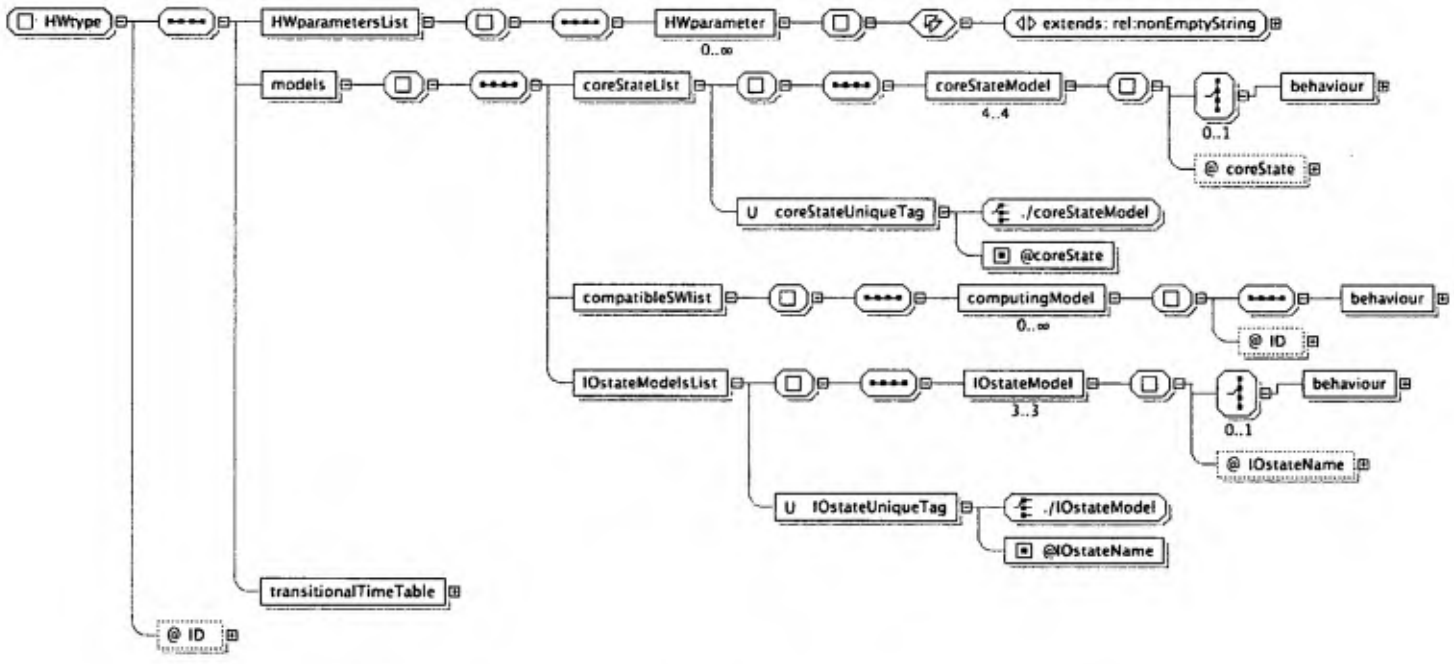

Figure B.23: Schema for the platform primitive type - HWtype.xsd

- ./localizedElementsHWstructure/linkList/link/sink (*nessie:positiveInteger*): sink block identifier
- ./localizedElementsHWstructure/linkList/link/@bidirectional (*xs:boolean*): boolean defining if the link is bidirectional or not. If the link is not bidirectional, data will only be able to flow from the source to the sink block.

We have defined a reference to the key *HWblockIDkey* to make sure that both *source* and *sink* elements relate to existing identifiers: if this constraint is not fulfilled the validating parser will trigger an error.

DOFtype.xsd

The *DOFtype* element (Fig.B.25) defines the different degrees of freedom that can be specified for a performance estimation run using Nessie. It supports both the determination of the different possible values for a Yeti model parameter and the definition of the different functional and platform possible structures: the combination of all these degrees of freedom defines the design space that can be explored. The different elements present in this schema are the following:

- ./valueDOF (*xs:complexType* named valueType): defines, as a degree of freedom, all the possible values that the parameter named ./valueDOF/@parameterName (*nessie:nonEmtptyString*) can take.
- ./structureDOF (*xs:comlexType* named structureDOFtype) determines as a degree of freedom which functional/platform structure will be used to explore a particular primitive. It is composed out of different elements:
  - ./structureDOF/@typeOfStructure (*nessie:HW_SWstringType*): defines the nature (functional or platform) of the structure and the related primitive. This simple type used restricts the string to the "HW" and "SW" values.
  - ./structureDOF/@abstractionLevel (*nessie:positiveInteger*): abstraction level of the primitive for which the structure is defined
  - ./structureDOF/@ID (*nessie:positiveInteger*): identifier of the primitive within the corresponding astraction level for which the structure is defined
  - ./structureDOF/structureChoice (*nessie:positiveInteger*): list of all the different possible structures identifiers defining the current primitive

valueType.xsd

The *yeti:valueType* element has been initially defined inside Yeti (see Sec.B.2.2) and is reused inside Nessie to define the values of the degrees of freedom. As

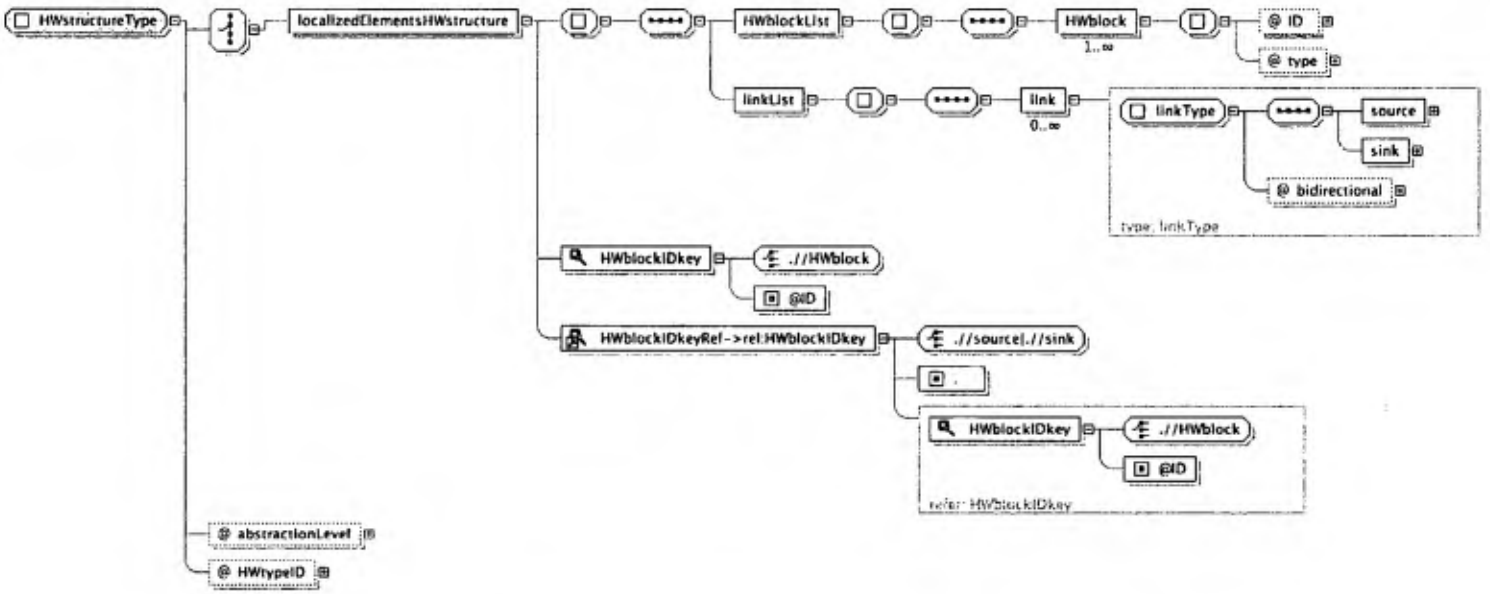

Figure B.24: Schema for the platform structure - HWstructureType.xsd

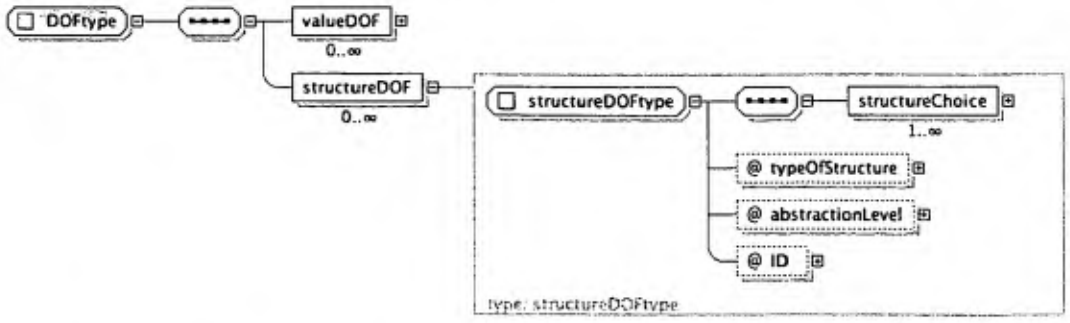

Figure B.25: Schema for the degree of freedom type - DOFtype.xsd

a remainder, the values can be either defined as a single value, picked among a list or generated based on a sweep process.

nessieSchedulingType.xsd

The *nessieSchedulingType* element (Fig.B.26) contains all the information relative to the event creation/triggering involved in the mapping of a functional structure on a platform structure. This file can be used for scheduling and timing analysis but also to understand how Nessie proceeds to the different allocation, scheduling and routing steps. The files generated are generally very large even for simple simulation due to the large event related activity of the mapping core: if only performance criteria results are desired, we advise the user to disable the generation of these scheduling files to reduce Nessie computation time. A *nessieSchedulingType* element is composed out of several occurrences of the ./timeStep complex type element containing:

- ./timeStep/@timeStamp (*nessie:timeType*): defines the time stamp associated with the addition or triggering of events. This type is a restriction of the *xs:float* simple type to positive numbers (including zero) to prohibit negative times.
- ./timeStep/addEvent (*xs:complexType* named addEventType): mentions, at the current time stamp, the creation of one event (selected among the four possible types of events) scheduled to be triggered at a time defined by ./timeStep/addEvent/@timeStamp (*nessie:timeType*)
- ./timeStep/triggerEvent (*xs:complexType* named triggerEventType): mentions the triggering of one event (selected among the four possible types of events) at the current time stamp

The four different possible events are the following[3]:

[3]To avoid long notations, the path describing the different attributes of the event type elements are considered as relative to an element instantiating the given event type

- The *SWdataTokenReceptionEventType* is an event related to the reception of a token at a given port or in the memory of a platform block requiring this token for execution. It is characterized by the following attributes:
  - ./@HWblockID (*xs:nonNegativeInteger*): identifier of the platform block receiving the token
  - ./@HWtype (*xs:nonNegativeInteger*): platform primitive from which the ./@HWblockID platform block derives
  - ./@SWblockID (*xs:nonNegativeInteger*): identifier of the functional block associated with the ./@HWblockID platform block and waiting for the token
  - ./@SWtype (*xs:nonNegativeInteger*): functional primitive from which the ./@SWblockID functional block derives
  - ./@dataTokenID (*xs:nonNegativeInteger*): identifier of the data token transmitted to the platform block
  - ./@memorized (*xs:boolean*): boolean indicating if the token is memorized or transmitted to a port of the platform block
  - ./@HWblockLinkedToPort (*xs:nonNegativeInteger*): reference to the platform block connected at the other end of the link that has sent the token. This information is useful to identify a port by their link which is more explicit than the identifier port in a token transmission context
- The *HWreleaseEventType* is an event related to the release attempt of a platform block: if all the required conditions are gathered, the block will eventually be released. It is characterized by the following attributes:
  - ./@HWblockID (*xs:nonNegativeInteger*): identifier of the platform block being potentially released
  - ./@HWtype (*xs:nonNegativeInteger*): platform primitive from which the ./@HWblockID platform block derives
- The *HWstateChangeEventType* is an event related to the state change of a platform block core or port and is characterized by the following attributes:
  - ./@HWblockID (*xs:nonNegativeInteger*): identifier of the platform block whose state will change
  - ./@HWtype (*xs:nonNegativeInteger*): platform primitive from which the ./@HWblockID platform block derives
  - ./@newCoreStateType (*nessie:coreStateNameType*: new core state value of the platform block. This type restricts the string simple type to "idle", "sleeping", "memorizing", "transmitting" and "computing" values
  - ./@HWblockLinkedToPort (*xs:nonNegativeInteger*): reference to the platform block connected at the other end of the link related to the current port

  - ./portID (*xs:nonNegativeInteger*): identifier of the port whose state is switched
  - ./newIOstateType (*nessie:IOstateNameType*): new port state value of the platform block. This type restricts the string simple type to "inactive", "receiving" and "sending".

  It is important to mention that both core and port will not necessarily switch at the same time: the different attributes are thus optional so that only the required attributes could appear depending on the part of the platform which has switched state.

- The *dataTokenMemorizationEvent* is an event relative to the reception of a data token in a platform block that is not associated with a functional block. This means that the platform will thus only be used to store the data token and deliver it afterwards to another block.
  - ./@HWblockID (*xs:nonNegativeInteger*): identifier of the platform block receiving the data token to memorize
  - ./@HWtype (*xs:nonNegativeInteger*): platform primitive from which the ./@HWblockID platform block derives
  - ./@dataTokenID (*xs:nonNegativeInteger*): identifier of the data token transmitted to the platform block

activityReportType.xsd

The *activityReportType* element (see Fig.B.27) represents the results of the timing analysis of the different blocks platform blocks based on the scheduling that has been performed. For each port and the core of each platform block, the absolute and relative time (expressed as a percentage of the a percentage of the total execution time) spent in each state is computed and reported in this XML file. This information is very useful to estimate the impact of the degrees of freedom value changes on the resulting scheduling. The different elements composing the *activityReportType* are the following:

- ./@duration (*xs:nonNegativeFloat*): the period of simulation time separating the beginning and the end of the scheduling or in other words the execution time resulting from the mapping of a given functional structure on a platform structure
- ./HWblockActivity (*xs:complexType*): describes the activity for the platform block defined by the identifier ./HWblockActivity/@ID (*xs:nonNegativeInteger*). The activity is split into two different parts: the core and the port related activity.
- ./HWblockActivity/coreActivity (*xs:complexType*): gathers all the activity measures for the different core states

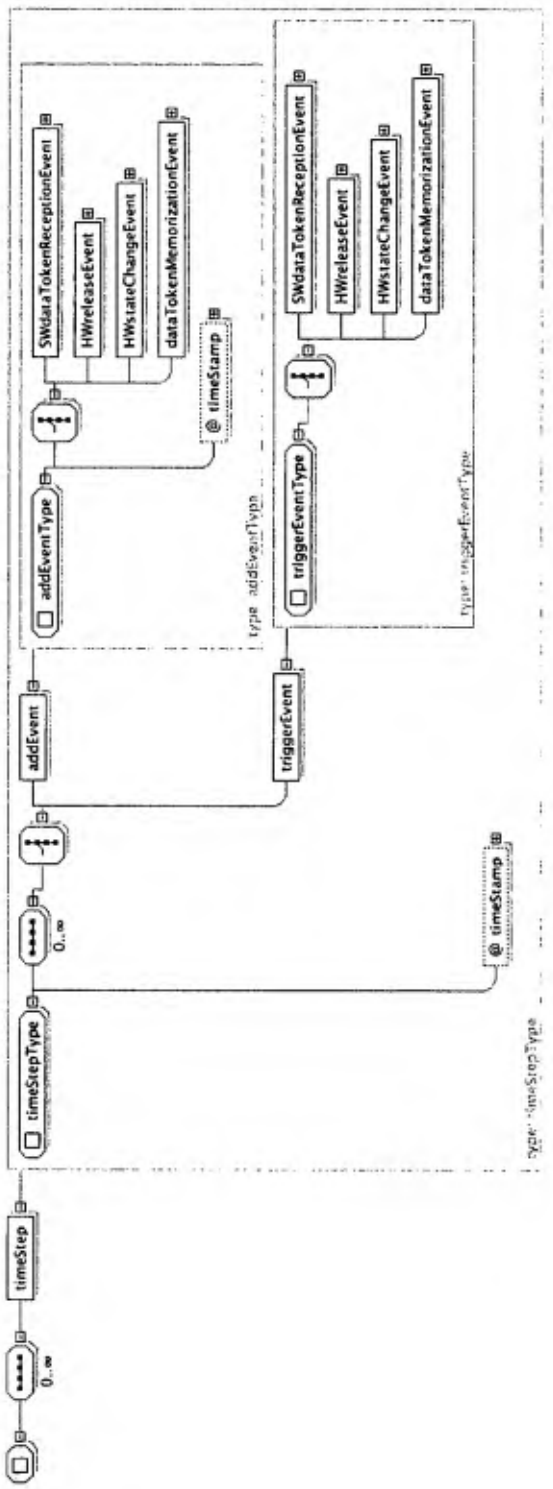

Figure B.26: Schema for the nessie event scheduling - nessieSchedulingType.xsd

- ./HWblockActivity/coreActivity/coreState (*xs:complexType*): defines the activity measures for one particular core. The different elements composing it are the following:
  - ./@name (*nessie/coreStateName*): defines the name of the core state (idle, sleeping, transmitting, memorizing, computing)
  - ./@relativeTimeOccupation (*xs:nonNegativeFloat*): total time during which the platform block has remained in the ./@name state expressed as a percentage of the execution time
  - ./@absoluteTimeOccupation (*xs:nonNegativeFloat*): total time during which the platform block has remained in the ./@name state
- ./HWblockActivity/portActivity (*xs:complexType*): gathers all the activity measures for the different port states
- ./HWblockActivity/portActivity/port (*xs:complexType*): port with identifier ./HWblockActivity/portActivity/port/@ID (*xs:nonNegativeInteger*) for which the activity measures are calculated. There is no limitation in the number of instantiated ports in a block so that the XML schema doesn't put any occurrence constraint on the number of those elements.
- ./HWblockActivity/portActivity/port/portState (*xs:complexType*): defines the activity measures for one particular port. The different elements composing it are the following:
  - ./@name (*nessie/portStateName*): defines the name of the port state (inactive, receiving, sending)
  - ./@relativeTimeOccupation (*xs:nonNegativeFloat*): total time during which the platform block has remained in the ./@name state expressed as a percentage of the execution time
  - ./@relativeTimeOccupation (*xs:nonNegativeFloat*): total time during which the platform block has remained in the ./@name state

nessieSolutionType.xsd

The *nessieSolutionType* element (see Fig.B.28) describes all the different solutions explored during a performance estimation run: each solution is composed out of a list of the estimated performance criteria and a list of the corresponding degrees of freedom values. The schema is composed out of the following elements for the criteria related part:

- ./criteria (*xs:complexType*): list of all the performance criteria values enumerated for the current solution
- ./criteria/time (*xs:nonNegativeFloat*): value of the execution time for the current solution. Since time is a cornerstone value for the mapping process, this attribute is mandatory and will always be available whatever the different defined criteria.

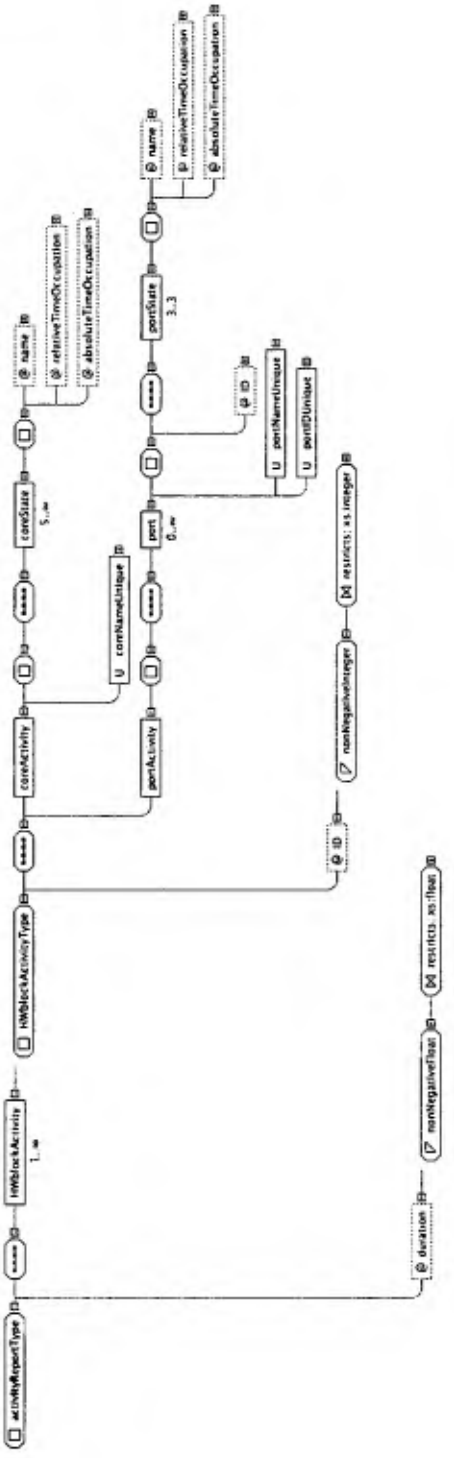

Figure B.27: Schema for the activity report - activityReportType.xsd

- ./criteria/timeDependentCriteria (*xs:complexType*): list of all the time dependent criteria values for the current solution
- ./criteria/timeIndependentCriteria (*xs:complexType*): list of all the time independent criteria values for the current solution
- .//criterion (*xs:complexType*): float element defining the value of the criterion named .//name (*xs:string*)
- ./DoFs (*xs:complexType*): list of all the degrees of freedom corresponding to the current solution. This list is composed out of three types of degrees of freedom respectively associated to the value of one parameter, the functional structure choice or the platform structure choice.
- ./DoFs/valueDOF (*xs:complexType*): degree of freedom defining the value of a Yeti parameter model. It is composed out of the following elements:
  - ./@parameterName (*xs:string*): name of parameter whose value is set by the current degree of freedom
  - ./@value (*xs:float*): parameter value chosen for this degree of freedom
  - ./numberOfPossibleDOFs (*xs:nonNegativeInteger*): number of possible parameter values for the current degree of freedom
  - ./@DOFindex (*xs:nonNegativeInteger*): index of the chosen degree of freedom within the vector of all possible degrees of freedom values
- ./DoFS/HWstructureDOF (*xs:complexType*): degree of freedom defining the structure chosen for a given platform primitive:
  - ./@abstractionLevel (*xs:nonNegativeInteger*): abstraction level of the platform primitive for which we choose a structure
  - ./@HWtypeID (*xs:nonNegativeInteger*): identifier of the chosen platform structure
  - ./numberOfPossibleDOFs (*xs:nonNegativeInteger*): number of possible structures associated with the platform primitive for the current degree of freedom
  - ./@DOFindex (*xs:nonNegativeInteger*): index of the chosen degree of freedom within the vector of all possible degrees of freedom values
- ./DoFS/SWstructureDOF (*xs:complexType*): degree of freedom defining the structure chosen for a given functional primitive:
  - ./@abstractionLevel (*xs:nonNegativeInteger*): abstraction level of the functional primitive for which we choose a structure
  - ./@SWtypeID (*xs:nonNegativeInteger*): identifier of the chosen functional structure
  - ./numberOfPossibleDOFs (*xs:nonNegativeInteger*): number of possible structures associated with the functional primitive for the current degree of freedom
  - ./@DOFindex (*xs:nonNegativeInteger*): index of the chosen degree of freedom within the vector of all possible degrees of freedom values

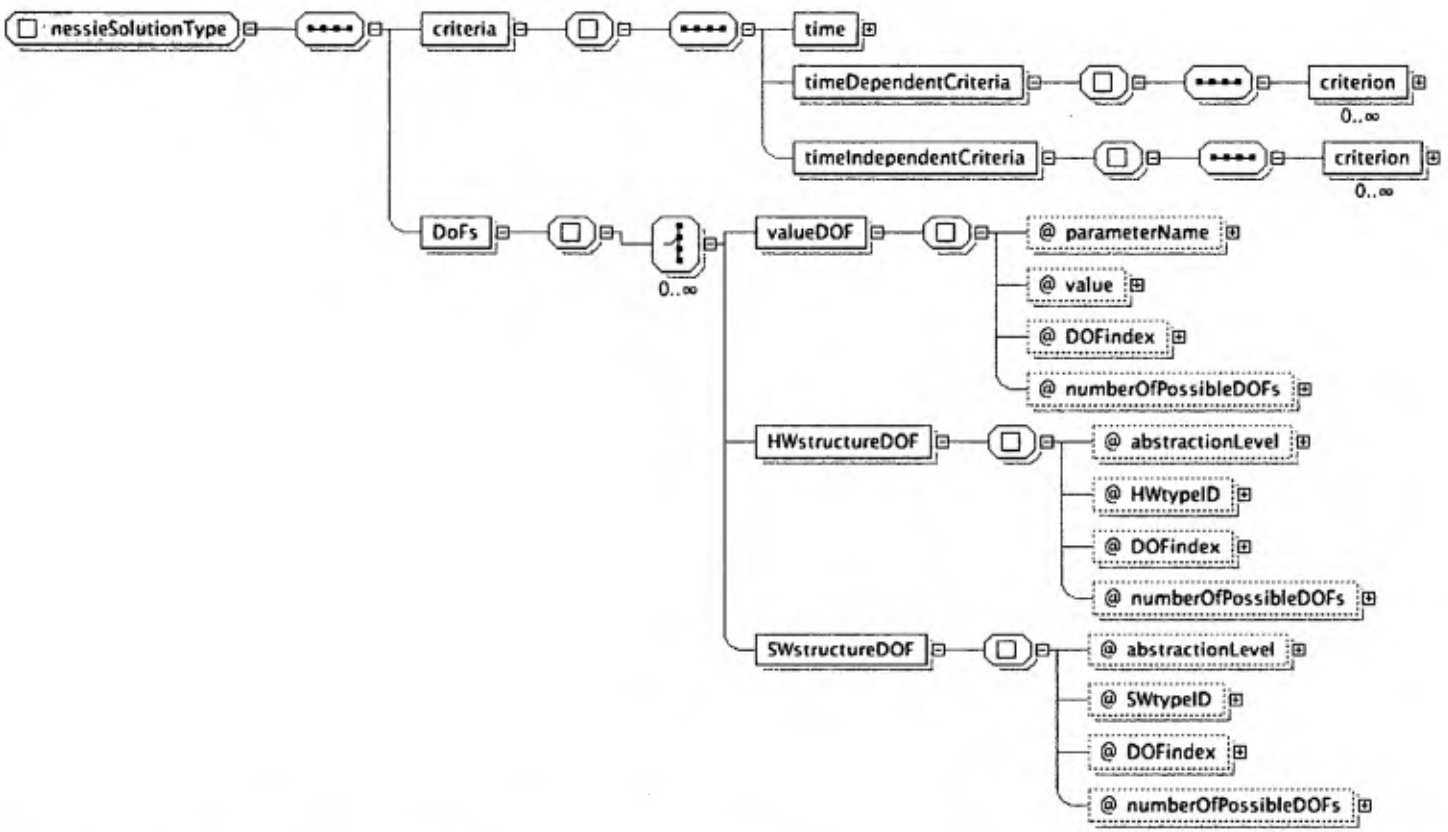

Figure B.28: Schema for the different solutions with the performance criteria and their corresponding degrees of freedom - nessieSolutionType.xsd

Bibliography

[1] "Xercesc 2.7 parser documentation," 2007. [Online]. Available: http://xml.apache.org/xerces-c/