FEASIBILITY STUDY FOR INTEGRATING A MODEL DRIVEN ENGINEERING APPROACH TO CIRCUIT DESIGN

Nastassia GUMUCHDJIAN

Supervisors:
Prof. Esteban ZIMÁNYI
Prof. Frédéric ROBERT

Mémoire de fin d'études présenté en vue de l'obtention du titre de Master en ingénieur civil en informatique à finalité ingénierie de la décision Année académique 2009–2010
Acknowledgment

First of all, I thank my two thesis supervisors, Prof. Esteban Zimanyi and Prof. Frédéric Robert for supporting me.

I also thank my advisors, Geoffroy Crucifix and Aliénor Richard, whose help, advice and supervision was invaluable.

Special thanks go to Sébastien Leduc, with whom I have had the opportunity to work and who took care of removing most of the technical problems I would have faced. He has given me the possibility to confirm my assumptions and to reflect upon the big picture instead of the details.

I would like to acknowledge the INRIA center with whom I have worked and I would specifically like to thank Alexis Müller who took an entire day to explain Model Transformation and Anne Etien who reacted promptly to any questions I came up with.

I would like to thank both services, BEAMS and CoDE, for their help and for providing me with everything I needed.

I thank both my sisters Tatiana and Larissa, for being such amazing sisters. I also thank my mother, and hope she will someday find the strength to take some rest.

Mustagh, I hope my gratitude reaches out to you. No day goes by where I don’t think of you.

Finally, Gary. You have substantially improved the redaction of this thesis, and, yet, this is not what I am most thankful for. I am a ship lost at sea, and you are the everlasting lighthouse that guides me all the way. Your unconditional love makes me strong.
Résumé

Le présent travail est une étude approfondie de l’applicabilité de l’approche de l’ingénierie dirigée par les modèles (MDE) à la conception de circuits électroniques. Nous commençons par expliquer le contexte dans lequel s’inscrit notre travail en présentant brièvement l’état du marché de l’électronique et les contraintes importantes auxquelles les concepteurs de circuits électroniques sont soumis, notamment le “time to market” et le “right the first time”.

Après avoir présenté les outils et les concepts informatiques nécessaires, nous expliquons brièvement les flots de conception actuels en électronique. Nous commençons par concentrer l’étude autour de Nessie, le simulateur de performance de haut niveau créé par le BEAMS (ULB).

Dans un premier temps, nous avons analysé la possibilité de standardiser ses entrées à l’aide de MARTE, un nouveau standard de modélisation adapté à l’électronique. Nous ne nous sommes pas limités à l’étude théorique du problème et nous avons obtenus des résultats pratiques.

Ensuite, dans un second temps, nous avons comparé sur une étude de cas les résultats obtenus avec ceux de l’équipe DART (INRIA, Lille). Cette démarche nous a permis de soulever certaines questions sur l’intérêt de la modélisation.

Dans un troisième temps, nous proposons une nouvelle méthodologie de conception de haut niveau, qui répond aux contraintes précitées. Nous terminons par une réflexion générale sur l’adoption de l’ingénierie dirigée par les modèles dans le monde de l’électronique.
Contents

1 Introduction 2
  1.1 Background .................................................. 2
    1.1.1 The market ................................................. 2
    1.1.2 Evolution of the costs .................................. 3
    1.1.3 Shortening lifecycle ..................................... 3
    1.1.4 Performance estimation tools .......................... 4
    1.1.5 MDE approach ............................................. 4
  1.2 Scope and problematic ......................................... 4
  1.3 Organization of the thesis ..................................... 5

2 MDE and tools 6
  2.1 Model-driven engineering ................................... 6
    2.1.1 Introduction to models .................................. 6
    2.1.2 Model transformations ................................... 7
  2.2 The MetaObject Facility specification ......................... 8
  2.3 UML .......................................................... 10
    2.3.1 Overview .................................................. 10
    2.3.2 Diagrams .................................................. 12
    2.3.3 UML profile ............................................... 13
    2.3.4 UML views ............................................... 14
  2.4 MARTE ......................................................... 15
  2.5 OCL .......................................................... 16
  2.6 QVT .......................................................... 17
  2.7 Eclipse-based tools .......................................... 18
    2.7.1 The Eclipse platform .................................... 18
    2.7.2 The Eclipse Modeling Framework ....................... 19
    2.7.3 M2M ......................................................... 20
    2.7.4 Papyrus .................................................. 20
## CONTENTS

3 Electronic circuit design  
3.1 Design flow ........................................ 24  
3.2 Nessie .................................................. 26  
   3.2.1 General ........................................... 26  
   3.2.2 XSD ................................................ 27  

4 Case study: RGB downscaler  
4.1 Motivation: downscalers in the real world ............... 29  
4.2 General principle ...................................... 30  
4.3 Instance ................................................. 31  
4.4 INRIA modeling of the downscaler  
   4.4.1 Overview .......................................... 31  
   4.4.2 Application ........................................ 32  
   4.4.3 Architecture ...................................... 34  
   4.4.4 Mapping ............................................ 34  

5 Modeling and transformations applied to Nessie  
5.1 Modeling ................................................... 38  
   5.1.1 Architectural model ................................ 39  
   5.1.2 Functional model ................................... 41  
   5.1.3 Non-functional model .............................. 46  
5.2 Transformations ......................................... 48  
   5.2.1 Mapping, query and helper ........................ 48  
   5.2.2 Example ............................................ 49  
5.3 Graphical user interface ................................ 50  
   5.3.1 The main window .................................... 50  
   5.3.2 Criteria editing .................................... 53  
   5.3.3 The main window .................................... 53  
   5.3.4 Behavior editing .................................... 53  
5.4 Modeling the downscaler  
   5.4.1 The application ..................................... 55  
   5.4.2 The hardware ....................................... 55  
   5.4.3 Discussion on the INRIA's technique .............. 57  

6 Design with performance estimation/multicriteria tools  
6.1 Recapitulation .......................................... 59  
6.2 Towards a new design workflow .......................... 60  
6.3 Generalization of the design approach ................... 62  
   6.3.1 Step by step explanation .......................... 62  
6.4 Example .................................................. 64  
6.5 Integration of multi-criteria tools ...................... 65
## List of Figures

2.1 MOF four layer architecture ........................................ 9
2.2 UML2 organization ..................................................... 11
2.3 An example of UML representation. Inspired from [19] ......... 12
2.4 A stereotyped class. Inspired from [14]. ............................ 13
2.5 UML views ............................................................... 14
2.6 MARTE’s architecture .................................................. 16
2.7 QVT Language Architecture .......................................... 18

3.1 A brief history of abstraction in design. Source: F. Schirrmeister 23
3.2 Typical design flow .................................................... 24
3.3 Iteration process in a design flow .................................. 25
3.4 The description of Nessie’s functioning and its input and output files. Source: Alexis Vanderbiest .............................. 26
3.5 Organization and dependence of XML schemas inside Nessie. Source: Alexis Vanderbiest .............................. 28

4.1 Workflow diagram of the RGB downscaler. ......................... 30
4.2 Downscaler Application : Main application, the application, the downscaler .................................................... 33
4.3 Downscaler Application : Representation of one of the horizontal filter. .................................................... 34
4.4 Downscaler Architecture .............................................. 35
4.5 Downscaler Task Association .......................................... 37

5.1 Representation of the elements that compose composite structure diagrams, and an example of assembled elements .... 40
5.2 Equivalence of the links present in Nessie’s xml file and the composite structure diagram port ............................. 41
5.3 Equivalence of part of the hierarchy present in Nessie’s xml file and the composite structure diagram .......................... 42
LIST OF FIGURES

5.4 Representation of the elements that compose petri nets, and the representation of a valid sequence of elements ......... 43
5.5 Representation of the elements that compose activity diagrams, and the representation of a valid sequence of elements ......... 44
5.6 Equivalence of the action element ........................................ 44
5.7 Equivalence of the transition element ................................. 44
5.8 Equivalence of the begin and final elements ....................... 45
5.9 Equivalence of the Fork and Join elements ......................... 45
5.10 source: Alexis Vanderbiest ............................................. 47
5.11 Modeling the behavior through state graphs ..................... 47
5.12 The overall process ...................................................... 51
5.13 The graphical aid to fill in Nessie’s entries ...................... 52
5.14 Criteria edition window ................................................. 53
5.15 The table edition window .............................................. 54
5.16 The analytical expression edition window ....................... 54
5.17 Downscaler Application : Abstraction 0 , the main application 55
5.18 Downscaler Application : variants ................................... 56
5.19 Downscaler Application : variant with serial RGB filters .... 56
5.20 Downscaler Application : variant with Horizontal and Vertical filters .......................................................... 56
5.21 Hardware: hierarchical modeling. .................................... 57

6.1 Increasing complexity of the input files ......................... 61
6.2 Proposed solution ......................................................... 62
6.3 New efficient design paradigm ....................................... 63
6.4 Step 1 : Creating variants and passing all the models to the GUI .......................................................... 64
6.5 Step 2 : The simulator returns the best variant to the GUI who selects them ................................................. 65
6.6 Step 3 : Only the best variants are refined ...................... 66
6.7 Step 4 : The same process takes place again, at level k+1 ... 67
6.8 An example with the 3 variants of downscalers and three variants of platforms ........................................... 67
6.9 The design paradigm with an MCDA tool ....................... 68
Chapter 1

Introduction

1.1 Background

1.1.1 The market

The electronics market is a key component of today’s economy, with its turnover of 1.3 trillion dollars and 18 million jobs worldwide. While microelectronics has been used in almost every other industry to simplify, automate and improve processes, today, the electronics sector faces complexity challenges of its own.

First of all, research and development costs have dramatically increased following the transition from microelectronics to nanoelectronics, which began in 2003. This transition raises many new challenges, as the existing designs grow ever closer to physical limits in both size and frequency. The second main stream of challenges comes from the evolution of the very nature of the chips we are trying to build. Whereas before we were content with building discrete systems, such as one chip as a microprocessor and another one as a video processor, recent developments have proved that it is possible to efficiently build multiple functionalities inside a single chip. This trend is commonly referred to as ‘more than Moore’.

Finally, the third main flow of problems, common to many other industries, comes from the growing environmental concern. Electronics consume a lot of precious and hard-to-recycle materials, and current production methods yield a huge proportion of waste. In the current psychological climate brought about by global warming, even the most unscrupulous electronics designer has to produce greener circuits.

The combination of those three forces leads to higher costs and shorter life cycles, hence the need for new, hopefully better design flows.
1.1.2 Evolution of the costs

According to a 2008 report presented to the French Assemblée Nationale [25], the development cost of the 90nm technology amounted to 500 million dollars, that of the 65nm to 750 million dollars and that of the 32nm to one billion dollars. Similarly, the evolution of the design costs of integrated circuits, according to the research director of Thales Group via [25], is that of Table 1.1. As can be seen from this data, development costs go up more than 50% with every technological quantum leap.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Number of gates</th>
<th>Cost of circuit development</th>
<th>Cost of the software for development</th>
<th>Hardware costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>130nm</td>
<td>9 millions</td>
<td>9 million €</td>
<td>30%</td>
<td>6.3 million €</td>
</tr>
<tr>
<td>90nm</td>
<td>16 millions</td>
<td>18 million €</td>
<td>50%</td>
<td>9 million €</td>
</tr>
<tr>
<td>65nm</td>
<td>30 millions</td>
<td>46 million €</td>
<td>66%</td>
<td>15.64 million €</td>
</tr>
</tbody>
</table>

Table 1.1: The evolution of the development costs of circuits. Source: Thalès group via [25].

There are several reasons that can explain the figures in table 1.1. The increasing costs of the software are explained by the fact that the power of embedded software doubles every 10 months but productivity of the software that helps in designing the embedded softwares doubles only every 5 years. Therefore, the development teams have to be enlarged to keep up. We will not discuss the other reasons but final costs also increase due to rising cost of production units or the increasing need for cleaner white rooms, amongst other factors.

1.1.3 Shortening lifecycle

The shortening lifecycle of electronic products is not only due to the evolution of technology, but also to other more subjective parameters such as consumer taste. Consumers become attracted not only by performances but also by the design, which changes even faster than technology.

As an example, a cell phone which is to be sold in December would start being developed in April or May and would have to be ready by September. Moreover, the new technologies are more rapidly absorbed by the market: it took almost a decade to saturate the cell phone market, but it only took two years for the MP3 industry.
1.1.4 Performance estimation tools

A wide range of specialists have already tried to address this situation. There are many elements of focus to resolve the lags between product capacity development and production capacity augmentation. In this work, we shall focus on one aspect: the improvement of the design flow, and especially the role of performance simulation tools.

These tools allow for a fast exploration of the design space; such tools are an evolution of both direct measurement and analytical models. Direct measurement is a post-design verification, and as such is not a very flexible way to solve the problem, while analytical models are usually less successful at evaluating subtle design tradeoffs.

Performance estimation tools use, as building blocks, models that are combined to simulate an electronic device.

1.1.5 MDE approach

The model driven engineering approach is based on models. The basic idea is to define a metamodel describing the initial space, another metamodel describing the final space, and a set of transformations between these two metamodels, which would allow for the automatic or semi automatic transformation of models between the two metamodels. A more thorough description of model driven engineering is the subject of Chapter 2.

1.2 Scope and problematic

In the previous section, we have seen that the electronics market is under serious constraints, notably time and costs. These are the market problems. Throughout this thesis we will take a step back.

Our starting point will be the performance simulation tools and we shall focus on Nessie, a high-level performance estimation tool developed at BEAMS. Nessie offers the advantage that it explores the design space in much less time than traditional tools, but it requires a modification of the traditional workflow and a set of input files that are both tedious and error-prone to write.

Given the high potential of the Nessie approach, much of our work has been focused on improving the user experience with Nessie, standardizing its input format and adapting existing graphical tools to produce the input files.

More information about our work on Nessie are presented in Chapter 5, while [28] remains the authoritative reference about Nessie itself.

A second important part of our work has been dedicated to the design of a new workflow, based on the new opportunities opened by tools such as
Nessie. More details about this part of the work can be found in Chapter 6.

As the present work is a feasibility study, we do not provide complete implementations of our proposed solution. We have, however, produced functional proofs of concept.

1.3 Organization of the thesis

Now that we have motivated and briefly sketched our thesis, we shall present the organization of the next chapters.

- In Chapter 2, we present and describe all the MDE tools used during this thesis;
- Chapter 3, we present the electronics tools and concepts that have been studied;
- In Chapter 4, we start with a case study provided by the INRIA. The way they model the system will also be provided in order to compare with the way it has been done at BEAMS. It will also introduce a general discussion that will take place in Chapter 6;
- Chapter 5 discusses the modeling and the transformations applied to Nessie;
- Chapter 6 finally presents the new design flow model which we believe can solve some of the problems that were presented in the Introduction;
- Finally, in Chapter 7, we present our general conclusions along with some reflections on the current state of model driven engineering in electronics and its future.
Chapter 2

MDE and tools

2.1 Model-driven engineering

2.1.1 Introduction to models

This thesis is based on “model-driven engineering” (MDE). But what is a model? According to [14], a model is “a reduced/abstract representation of some system that highlights the properties of interest from a given viewpoint. The viewpoint defines concern, scope and detail level of the model.”

The more complex the system is, the more modeling can be effective. Good modeling improves understandability of the problem. It also comes with positive side-effects such as easier reuse and sharing. According to [14], useful engineering models should be abstract (emphasizing only the important aspects), understandable, accurate, predictable and inexpensive. We shall come back to this later on, but one way to achieve this set of goals is to use graphical models, though we should keep in mind that best models are those that use the right proportions of graphical and verbal modeling, as they both present different advantages.

In MDE, models are broadly divided between three groups:

- Computation Independent Models (CIM) describe the domain and requirements of the system. They do not have any computation-specific information.

- Platform Independent Models (PIM), do not have any technology-specific implementation information, but can contain abstract, computation-specific information.

- Platform Specific Models (PSM) do contain technology specific implementation information.
2.1.2 Model transformations

As stated earlier, models can be a great tool for communication both the outside world and with oneself, as a way to more clearly represent the problem at hand. In this day and age, however, with information processing technologies everywhere around us, we can expect more from our models than from a mere drawing on a board.

Computer-based model design offers the opportunity to harness the computer’s ability to process information in order to apply transformations on the models.

The mechanism behind model transformation is fairly simple to understand. The goal is to confront the forces of change by separating and relating platform-independent models and platform-specific models using transformation techniques. By keeping these models separate but related, we can work with the model from the perspective that is most conducive to our understanding of the system.

Furthermore, transformations between models can be layered in order to keep each transformation step understandable and manageable.

A mapping is a transformation specification, including rules and other information, for transforming a model to another one, whereas a model transformation is the process itself of converting one model to another. A very common use case of mappings (and, thereby, of transformations) is to transform a PIM to a PSM. If the mapping is sufficiently robust, we can define the mapping once and then work only on the PIM, which is usually more abstract and thus easier to work with.

A commonly encountered illustration of this principle is the use of high-level programming languages, whereas by “high-level” we mean any other language than machine code. Once you trust your C compiler to be robust enough, you can model your program in C and have the transformation done by the compiler. Once you realize that C and machine code are just two different languages to model computations, you have understood the principle of model-driven engineering. There are also examples of higher-level languages for which the compiler generates C code, which is then compiled to machine code.\footnote{Model-driven engineering aims to go a step further in abstraction and work directly with graphical models where it makes sense.}
2.2 The MetaObject Facility specification

The Object Management Group is the group responsible for, amongst other projects, the specifications of the well known Unified Modeling Language\(^2\) and Common Object Request Broker Architecture\(^3\), which are well beyond the scope of this work. They are also responsible for the MetaObject Facility, which will be our focus here.

The MetaObject Facility, or MOF, is the OMG proposition for metamodeling of object-oriented systems.

As an OMG standard, MOF is part of a larger scheme and has to integrate with existing OMG standards. For example, MOF basic types reuse CORBA basic types. Furthermore, the OMG specification has to be language independent and suitable for a distributed environment. It is also an open standard proposed by the OMG with the expectation that it will be implemented by many independent vendors, the explicit goal being to allow interoperation between multiple implementations and multiple tools. As the OMG provides a specification, not an implementation, but expects all players to stick to their rules, the specification has to be fairly stable, with long release cycles.

The OMG objective with MOF is to provide a minimal, general core for Model-Driven Engineering. MOF is not designed to be used directly by humans: it is to be the *lingua franca* of model-driven engineering tools.

There is more to model-driven engineering than purely describing models, and thus there is more to MOF than a static description of object-oriented design: MOF supports model transformations. These transformations are powerful tools; they can be used, as explained in [15], to transform from MOF to EMF and from EMF to MOF. But the original goal of these transformations is the very nature of model-driven engineering: they can transform a Platform-Independent Model into layers of more and more platform-dependent models up to the point where the model is specific enough for generating code.

The Holy Grail of model-driven engineering is a series of such transformations that would permit any modification of the original model to be reflected in the final code without further human intervention.

MOF is located at the very top of the OMG standard that concerns the representation of meta-models and their manipulation and recursively

---

\(^1\)Most notably, this was the case for the first implementations of C++.

\(^2\)A unified language for universal modeling; see [21] for more information.

\(^3\)Best known through its acronym, CORBA is a language-independent specification for distributing object-oriented applications. For more information, see the official website [20].
defines itself. It is at the top of a four layer modeling architecture presented in Figure 2.1.

Figure 2.1: MOF four layer architecture

Let us present a more concrete example to illustrate the point of this four-level separation. To frame the ideas, we can draw a parallel with an object-oriented programming language. Level 0 is the concrete, object level, where real, live objects live. Were we writing and application that used data related to writers, we could have, at this level and at runtime, an object of type `writer` whose `name` attribute would be equal to “Nastassia Gumuchdjian”.

In order to be able to instantiate this live object, we would need a compile-time class describing all `writer` objects in generic terms, for example specifying that writers have a name. That class definition would be an object of the model; it is, for example, a rectangle in a UML class diagram. This is the kind of object that lives at level 1.

The class diagram itself is also a kind of object, though a much more abstract one. If we wanted to formally describe what a class diagram is, we would need a language that would be able to do just that, in the same way that a UML class diagram can describe what the `writer` class is, and in the

---

4Indeed, the MetaObject Facility can be seen as an object-oriented modeling language.
same way that the writer class defines what the runtime objects of type writer are.

The class diagram is an object of level 2 in the MOF hierarchy. As the reader can now infer, it is thus in some way an instance of a more general concept defined in the fourth, final level of the hierarchy, which is the MetaObject Facility itself.

To summarize, MOF can be seen as a “Class” in the Java/C++ meaning of the term, of which UML, QVT or SysML are “instances”. These instances are themselves classes, and in particular the UML class diagram can be seen as the “class” of which the writer class is an instance, and, finally, the “Nastassia Gumuchdjian” object is an instance of the writer class. Since the class hierarchy of a program is often thought of as the model of that program, UML can be called a metamodel and MOF could be said to be a metametamodel. As the process can be infinitely recursive, though, common vocabulary usually stops at metamodel for anything that is more general than a model.

2.3 UML

2.3.1 Overview

The Unified Modeling Language (UML) is a graphical modeling language based on pictograms. It originated in software engineering, more precisely in the object oriented culture, as a tool to simplify the conception of large (object-oriented) software systems. UML is the result of a fusion of several preexisting modeling languages: Booch, OMT,OOSE. UML has now become the standard supported by the OMG. The last revision, dubbed UML2, has been released by the OMG in 2007.

UML2 comprises thirteen types of diagrams, amongst which nine were already present in the UML1 specification. It is important to keep in mind what UML is and is not; UML is not a method, it merely provides a means to model a system. An example is presented in Figure 2.3.

UML is composed of several subsets:

Views They describe the system from a certain point of view, which can be organizational, dynamic, temporal, geographic, logic, etc.

Diagrams They are the graphical elements. They describe the content of the views which are abstract notions. Each diagram can belong to several views.
Figure 2.2: UML2 organization

**Model elements** They are the fundamental bricks of the diagrams, and are used in several diagrams. These are, for instance, the associations, the classes, etc.

UML cannot be summed up and understood in a single paragraph. However, to help with understanding, an example of modeling with the Class diagram is provided in Figure 2.3.

This model contains five elements (Person, Building, House, Apartment building, Apartment). The links between the classes (graphically represented by rectangles) represent the following statements:

- a House is a Building
- an Apartment building a Building
- a Person or multiple Person(s) can have 0 or several Building(s).
- an Apartment building can be composed of one or several Apartment(s).
2.3.2 Diagrams

The thirteen diagrams of the UML specification can be classified in three categories. It is important to understand what each category has the power to represent and then within that category what each diagram represents. This understanding is important to model a system in coherence with the standard’s specifications.

The organization of the thirteen diagrams of the UML2 specification is shown in 2.2. Let us briefly present each of them:

- **Structural or static diagrams**
  - Class diagram: Expresses the static structure of a system in terms of types (called classes in the UML terminology) and relations between them.
  - Object diagram: Shows objects and connections. An object diagram is an instance of a class diagram at a moment in time.
  - Component diagram: Describes the components and their dependencies in the realization environment.
  - Deployment diagram: Shows the physical disposition of the system.
  - Package diagram: Represents the dependencies between the packages that make up a model.
  - Composite structure diagram: Shows the internal structure (including parts and connectors) of a structured classifier or collaboration.

- **Behavior diagrams**
Use-cases: Represents the use-cases, the actors and the relations between the use-cases and the actors.

State machine diagram: Represents finite state automata, from a state and transitions’ point of view.

Activity diagram: It is a variant of the State diagram. In this diagram instead of the states, the focus is set on the activities.

Interaction or dynamic diagrams

Interaction Overview diagram: Shows flow of control of the interactions.

Sequence diagram: Shows the interactions between objects by focusing on the sequence of the interactions from a temporal point of view.

Communication diagram: Presents a set of roles played by objects in a given context, as well as the connections between these objects.

Timing diagram: Interaction diagram where the focus is on timing constraints.

2.3.3 UML profile

In UML, a profile is a way to customize the existing UML models to make them domain specific. The standard semantics can only be refined in an additive manner; this means that the extension mechanisms do not allow a refining that would contradict the standard semantics. Profiles are defined using stereotypes, tag definitions, and constraints that are applied to the specific model elements, such as Classes, Attributes, Operations and Activities. An example of stereotype “hwProcessor” applied to a class is shown in 2.4.

Figure 2.4: A stereotyped class. Inspired from [14].
2.3.4 UML views

One proposed methodology to model a system in UML is to consider the different views that can be superimposed to collaborate.

The use-case views are the description of the model by the protagonists of the system. It corresponds to the needs of every actor and hence addresses the “what” and “who” questions.

The logical view is the definition of the system from the inside. It explains how the needs can be fulfilled and addresses the “how” question.

The implementation view defines the dependencies between the modules.

The process view is the temporal and technical view that expressed the notion of concurrent tasks, stimuli, control, etc.

The deployment view describes the geography and architecture of each element of the system, it addresses the “where” question.

What Figure 2.5 shows is that the description of a system can be made from different perspectives. Now, the UML diagrams cover these different perspectives, and some are located at the intersection between several views. It is important to choose the relevant view to represent and model the part of the system that is needed.
2.4 MARTE

There are two kinds of OMG modeling languages: the general purpose languages (UML2, SysML, ...) and the domain specific languages (UML profile for SOC, MARTE, ...) as shown in 2.4.

We shall focus on MARTE, which is the standard for modeling and analysis of real-time and embedded systems. It is a profile based on the UML standard and an evolution of the profile for Schedulability, Performance and Time (SPT). The motivation behind MARTE was to simplify the design and conception of electronic circuits. Just as UML, a system can be modeled in a very clear, precise and simple way because it is graphical, which appeals to the visual part of our brain, which is generally faster and more efficient than reading plain code.

MARTE’s architecture, represented in Figure 2.6, is composed of three packages\(^5\). The top package is the foundation of MARTE and serves as a base to everything else in MARTE. It consists of six sub-profiles: CoreElements, NFP (Non Functional Properties), Time, GRM (Generic Resource Modeling), GCM (Generic Component Modeling) and Alloc (Allocation modeling). From these foundational concepts, MARTE is split into two categories of extensions: MARTE design model, which allows modeling of the left branch

\(^5\)Actually four, the three presented in Figure 2.6 and an appendix package.
of the “V” development cycle, and MARTE analysis model, which helps in model-based analysis of real-time, embedded systems.

2.5 OCL

Object Constraint Language (OCL) is an OMG language for the formal expression of constraints. A constraint is a relation between elements of the model that defines propositions that should remain true to guarantee the validity of the model.

OCL is usually used in conjunction with UML. The constraints are important additional information that could not be specified by UML elements; OCL allows the expression of several types of constraints, amongst which:

- Invariants within a same class or type: constraints that should always be verified to ensure a correct functioning of the instances of the class or the concerned type.

- Constraints within an operation: constraints that should always be verified to ensure a correct functioning of the concerned operation.
• Guards: constraints on the modification of the state of an object.

• Navigation expressions: constraints that represent the paths within a class structure.

OCL is a typed, declarative language. It allows for the comparison of instances of a same type.

From the Eclipse help page, an example of constraint that would assert that a Book’s reference to its author must not be null would be:

```xml
<constraint
  lang="OCL"
  severity="ERROR"
  mode="Live"
  name="%example2.name"
  id="example2"
  statusCode="102">
  <description>%example2.desc</description>
  <message>%example2.msg</message>
  <target class="Book">
    <event name="Set">
      <feature name="author"/>
    </event>
  </target>
  not author.oclIsUndefined()
</constraint>
```

2.6 QVT

Query/View/Transformation (QVT) is an MDA standard for model transformations defined by the Object Management Group (OMG).

It is the language used for model transformations. In fact, it defines a standard way to transform source models into target models. The models, both source and target, may conform to arbitrary MOF metamodels.

There are many implementations of QVT, amongst which QVT-Operational (Eclipse M2M, Borland Together, SmartQVT), QVT-Core (OpitmalJ), QVT-Relations (Eclipse M2M, ModelMorf, MediniQVT).
2.7 Eclipse-based tools

2.7.1 The Eclipse platform

The Eclipse project started out\(^6\) as a Java Integrated Development Environment, but quickly evolved to become much more. As the project developed, it became apparent that the initial design allowed for a clear separation between a basic platform, which provided the framework required to integrate many different parts, which became to be known as ‘bundles’ or ‘plug-ins’, and the Java IDE itself, which could be considered as a small collection of plug-ins. Eclipse is now an umbrella project that covers a huge number of very different projects, all built on top of the Equinox platform as collections of plug-ins, ranging from IDEs for various languages such as Python [5, 27], PHP [17, 12], C/C++ [26, 7] and, of course, Java [18, 9], to more integrated environments such as the Java EE perspective [8, 22] which integrates both a web server and a web browser directly inside Eclipse for easier development, and up to full-fledged applications which bear little resemblance to anything remotely related to IDEs, such as an e-learning platform for cryptography [3], a bittorrent client [2], a personal information manager [4] or a RSS reader [24], to cite only a few the publicly acknowledged open-source projects.

\(^6\)More information about the history of the Eclipse project can be found scattered around the Eclipse website [11].
2.7.2 The Eclipse Modeling Framework

Amongst the many projects built on top of Eclipse, one of the major ones is the Eclipse Modeling Framework, or EMF [10]. The goal of this framework is to provide a set of plugins that facilitate model-driven software development.

The most prominent software modeling framework today is, by far, the Unified Modeling Framework, also known as UML. However, UML is often considered too heavy-weight for practical use. Indeed, UML has been designed more as a set of tools to be customized and refined for every application than as a toolbox to be used as-is.

The one part of UML that is really often used in the real world of software development is the class diagram, which statically represents the relationships between the basic components of a data model. The Eclipse Modeling Framework defines a new kind of diagram, called eCore, which is a simplified version of the UML class diagram. Even though the simplification does indeed reduce the modeling power of the diagram, this extra power is actually seldom used in class diagrams, and the eCore model allows for the representation of the expected relationships of a class diagram, such as associations, attributes, compositions, etc.

The Eclipse Modeling Framework can use the eCore model to generate most of the corresponding Java code, leaving relatively little work to be done by hand. By default, the EMF can generate a model, which is the code corresponding to the set of classes represented by the diagram, an ‘edit’ project, which consists of classes and methods used to access the model in a relatively cleaner and more generic way than direct access, in order to limit the problems one could encounter when modifying the model, an editor and a skeleton code for unit tests.

Recent versions of EMF can generate different types of editors, but this is beyond the scope of this work. What these editors have in common is that they allow for the creation and edition of a model defined by the previously defined meta-model. Let us examine the workflow on a very basic example.

The first step in using EMF is to build a meta-model. The basic version of EMF allows us to define the meta-model as a tree, though plug-ins exist to provide a more UML-like interface. Say we define a simple meta-model for books, where a book has one or more authors and an author has one or more books. We create an EClass Book, another EClass Author, and we add a one-to-many reference to Book in Author and vice versa. This is our meta-model.

If we now generate the editor for this meta-model, we can add books and authors to it, and define which author has written which book. Now,
of course, this is a very simple example designed to help the reader in understanding what EMF calls a meta-model and what it calls a model. The EMF meta-model is very intuitively used to model complex data, as our little example illustrates, but it can also be used for more abstract modelisation.

The Eclipse project did not start the Eclipse Modeling Framework from scratch: EMF started out as an implementation of a pre-existing standard called the MetaObject Facility Specification, published by the Object Management Group. While the general principles are common to EMF and MOF, there are a number of differences which mostly come from the different goals and scope of the two projects.

As previously stated, MOF has to be stable and reliable, and is implementation-defined. On the other hand, EMF has been developed almost informally by the open source Eclipse project. The goal of EMF is to bring the benefits of meta-modeling to the Java developer with as little investment as possible on his part. This means that the scope of the project is somewhat more limited, and that EMF is mostly implementation-defined, with the specification lagging behind the implementation.

Despite these differences in scope and goals, there are many similarities between EMG and MOF, as both attempt to define a standard for meta-modeling. They are actually close enough that it is possible to define automated transformations between them [15]. This is somewhat simplified by the fact that they both support a common format for serializing their models and meta-models, namely the OMG’s XML Metadata Interchange [16].

2.7.3 M2M

Model to Model (M2M) is an Eclipse project that provides a framework for model-to-model transformation languages. The spirit behind its architecture is closely related to that of Eclipse: M2M is composed of a core, which is the transformation infrastructure, and transformation engines, which execute the transformations and are plugged into the infrastructure. Three transformation engines have been defined: ATL, Procedural QVT and Declarative QVT.

2.7.4 Papyrus

Papyrus is an open-source tool dedicated to UML2 modeling and integrated in the Eclipse environment. Papyrus respects both the di2 (Diagram Interchange) and UML2 standards and offers an extensible architecture that enables users to add new diagrams, new code generation procedure, etc. It also supports new UML2 profiles development. Among others, it enabled
the development of the MARTE profile, which is the reason why it was used extensively during this thesis.
Chapter 3

Electronic circuit design

As explained in the introduction, the average product lifecycle in electronics is shortening, which puts much pressure on producers of semiconductors [25]. For example, for a new cell phone that has to reach to market in december, semiconductor producers would receive the specifications around april or may and are expected to deliver their products in september, with a huge responsibility in case of any delay as that would likely make the new cell phone miss the Christmas season. The reasons behind this shortening of the expected time to market are well beyond the scope of this thesis; instead, we are focusing our attention on suggesting solutions to reach the intended goal: more efficient electronic devices produced much faster.

As a reaction, the electronics industry has been “disaggregating from a vertically oriented model into a horizontally oriented one” [29]. This means that each company is reducing its activity to only their core competencies. Hence, from IP creation to product specification, every single part of the design flow is done by different actors. Practically what this means is that these competencies are difficult to interconnect, the different actors having some levels of impedance mismatch.

Another phenomenon that appeared in parallel with the growing integration density is a progressive increase in abstraction level, as illustrated by Figure 3.1. These phenomenon are, of course, related: abstraction is increased in order to reduce the apparent complexity of new systems. As the actual complexity of the systems increases, people stack up new levels of abstraction to keep the apparent complexity manageable. Right now, designers work at a system level, assembling IPs to create new systems without ever going into the lower-levels.

We are moving towards design methodologies that favor re-use and right-the-first-time implementations; this trend is summarized in [29] as “Design methodology has become THE focus.” The rise in abstraction helps re-use,
CHAPTER 3. ELECTRONIC CIRCUIT DESIGN

The Next Level of Abstraction in the Architecture Space

Figure 3.1: A brief history of abstraction in design. Source: F. Schirrmeister
understandability and productivity, whereas performance simulators would prevent more errors in the design.

In this chapter, we shall first present the current trends in design methodology for a better understanding of the possible perspectives described in Chapter 6.

We shall then briefly present Nessie, which has been our starting point to apply MDE methodologies.

3.1 Design flow

Figure 3.2: Typical design flow

Figure 3.2 shows the typical System-On-Chip design flow. First, the system-level architects work with application developers to develop the platform and the functional specifications.

Sometimes, the platform and the functional specification are developed independently. This is a simpler way to design the system, but it is only
suitable for systems that are not really critical. Indeed, the resulting system is often far from optimal, as nothing has been done to optimize the software for the hardware and vice-versa.

Let us take a look at the design flow as it stands when hardware and software are designed together. This practically has to be a top-down approach, which makes it harder to estimate final performances. The design flow is as follows. At each level, the system is designed then verified. Either via automatic transformations or through manual labor, the system is then translated to the next, lower level. At that \( n-1 \) level, the system is, again, as thoroughly verified as possible with that level of abstraction, and the process begins anew.

If that was the end of the story, all would be well. However, as the process goes down the levels of abstraction, errors can be discovered in the higher tiers, and we might arrive at a final product that indeed works, in that it does the right thing, but that is way too slow to be of any use. In these cases, the designers have to iterate, going back to a much higher level and remaking, nearly from scratch, all of the translations to go lower in abstraction. This is illustrated on Figure 3.3.

This iterative process is, of course, pretty expensive.
3.2 Nessie

This is where the performance simulation tools come into action. Performance simulation tools are basically tools that, based on numerical or analytical models, can rapidly estimate the performances of a system. Amongst the wide range of estimation tools we put our focus on Nessie.

3.2.1 General

Nessie has been developed by the “digital” team of the BEAMS department, ULB. It is a high level performance estimator that has been designed to be highly flexible on two fundamental aspects: it can simulate systems based on multiple user-defined criteria, and it can work at multiple levels of detail.

The input basically consists of two trees of models, one for the platform models and one for the functional models, a description of the behavior of the hardware elements according to the state the elements are in, and simulation criteria.

The notion of a tree of models warrants more explanation. Each depth level in the tree of models corresponds to a given abstraction level. A node thus represent a variant of the model at the given abstraction level; the children of a node are the different ways to implement the next level of abstraction for this node.

Given these four inputs, at each abstraction layer, Nessie exhaustively tests every possible combination of one platform and one functional model.
For each combination, based on the behavior files that contains the analytical models corresponding to the individual elements, Nessie computes an approximation of the performances. At each level, Nessie chooses the best combination, and for the following levels, only uses the children of the chosen models.

As the abstraction level goes down, the computations that Nessie performs take more time and are more accurate, as Nessie has more and more information about the system. The tree structure of the models allows Nessie to find local optima without having to compute all the possible combinations at the lowest level; the gain comes from the fact that the lower we go in abstraction levels, the more ways there are to implement a system. And, of course, as we want to test all combinations, even a small increase in the number of components can cost a lot of time.

3.2.2 XSD

XML Schema is a language that describes XML schemas; it is most often called XSD, as a reference to the file extension given to XML Schema files, which itself comes from “XML Schema Document”, as in “Microsoft Word Document”. XML Schema is an XML-based language, which means that the XSD syntax is actually XML. Since all input files for Nessie are XML documents, the original author [28] also authored XSD files to describe the format of the input files. The relationships between the different files are represented on Figure 3.5.

The “nessieSimulationType.xsd” is the root file of almost all other types of input files; the “customizedTypes.xsd”, “nessieScheduling.xsd”, “activityReport.xsd” and “nessieSolutionType.xsd” have not been considered at all in this thesis and will not be explained\(^1\).

We focused our efforts on “nessieSimulationType”.

Let us briefly present each of these files:

**criterionType** Represents the criteria according to which the simulation will be run. For instance, if we modeled a system and we wanted to estimate the power consumption and the area of the dye, we would create a list at the beginning of the document with one tag for the power and another for the area.

**SWhierarchyType** Represents the application primitives at different levels of abstraction, in a hierarchical way. The application is described by petri nets in which the elements (SWtype) are places.

---

\(^1\)For further details, please consult Appendix 2 of [28].
CHAPTER 3. ELECTRONIC CIRCUIT DESIGN

Figure 3.5: Organization and dependence of XML schemas inside Nessie. Source: Alexis Vanderbiest

**SWstructureType** Represents the transitions between the different elements (SWtype) that have been described in the SWhierarchyType. A transition can only be defined between elements of the same abstraction level.

**HWhierarchyType** Represents the platform in a way similar to the way SWhierarchyType represents the application. For each abstraction level, this file lists the associated platform primitives (HWtype), each of which has an associated behaviour element that describes the computational model characterizing the performance criteria (behaviourType). The behaviour element is optional; when it is not present, it simply means that the block based on that primitive cannot find itself in that state.

**HWstructureType** Defines a platform structure at a particular abstraction level for a given platform primitive.

**DOFtype** Defines the different degrees of freedom that can be specified for a performance estimation run using Nessie.
Chapter 4

Case study: RGB downscaler

This chapter is an introduction to modeling. It will present a case study provided by the INRIA\(^1\) in Lille with whom we have been collaborating. It will serve as a comparison with the way we shall be modeling our system (cf. Chapter 5).

4.1 Motivation: downscalers in the real world

The resolution of analogical televisions has reached its peak with the DVD format, which provides a $720 \times 576$ pixels resolution. This is largely sufficient for a cathodic television, but not for large plasma screen scoring more than a meter and a half on the diagonal. This is where the need for numerical television screens arise, along with the definition of higher-resolution standards commonly referred to as HDTV, as in High-Definition Television. There are two worldwide norms for HDTV: 720p and 1080i, where the ‘p’ stands for ‘progressive lines’ and the ‘i’ stands for ‘interlaced lines’. For both norms, the indicated number represents the height of the image, and both are to be understood as $16/9$ formats, which means respectively $1280 \times 720$ and $1920 \times 1080$.

The required resolutions are scarcely represented in the whole population of television screens. Actual plasma screens mostly offer smaller resolutions, such as $848 \times 848$, $1024 \times 1024$ or $1366 \times 768$. This is where downscaling comes in: the input stream to the television most often has a higher resolution than the television screen. There is consequently a need for a device that can, for example, take as input a 720p stream and offer as its output, which will be the input of the screen, a corresponding $1024 \times 576$ pixels stream. (considering a $1024 \times 1024$ display).

\(^1\)Institut National de Recherche en Informatique et Automatique.
4.2 General principle

From the field of embedded signal processing, we chose to model a downscaler, which is basically an electronic circuit that reduces the size of images in a stream. At a rather high level, here is how it is used. For each image of the captured video stream, the red, green and blue (henceforth referred to as RGB) components of the image are isolated. On each of these three components, the downscaler filter is applied, then the image is recombined to produce the next image of the output stream. Figure 4.1 shows a graphical representation of this workflow.

Now, of course, the interesting part is the downscaler filter. A first observation is that, through our design, we can separately apply the same filter on each color. This allows us to simplify the filter itself, as it only has to process one grayscale image, and to accelerate the process as we can parallelize the slowest operation. The following discussion is about a single downscaler filter; the reader is invited to keep in mind that there should be three such filters, and that is why we implicitly consider a grayscale image.

There are many ways to reduce the size of an image.\footnote{The simplest approach is to delete one in \( n \) rows. This, of course, is somewhat primitive and has many limitations. Many other techniques have been developed; see for example \cite{6}.} In this case study,
we take a relatively simple approach which consists of successively applying two one-dimensional filters, which will be identical except for the direction and two parameters. The filter will apply the same operation to each row of the image, where a row is either a column or a line, depending on the direction of the filter. On each row, the filter will reduce $n$ pixels to $m$ pixels by means of an averaging operation.

4.3 Instance

In order to work with a more concrete example, let us set the values of the parameters. We want to transform a stream from $352 \times 288$ to $132 \times 128$.

Let us consider that the first filter is the horizontal one, and that the horizontal filter takes 8 pixels and averages them down to 3 pixels. This means that, for each line, we have $352 \div 8 = 44$ averaging operations to compute for a total of $288 \times 44 = 12672$ averages.

The by-now $132 \times 288$ stream then enters the vertical filter. Now, the vertical filter takes 9 pixels at a time and produces 4 pixels as output. This means we have to compute $288 \div 9 = 32$ averages per column, for a total of $132 \times 32 = 4224$ averages. The final output stream has, as intended, $132 \times 128$ pixels.

Now, let us observe two important facts. First, we have chosen to apply the horizontal filter first. What would have happened had we decided otherwise? In our case, we have had $12672 + 4224 = 16896$ averages to compute. Had we applied the filters in the opposite order, we would have had $32 \times 352 = 11264$ vertical averages and then $44 \times 128 = 5632$ horizontal averages, for the same total of $16896$ averages. Now, which is better? Well, it might depend on many factors, such as the relative costs of the two types of averages (vertical and horizontal), or the kind of hardware we have at our disposal.

In order to keep the discussion at a very concrete level, we shall assume that the downscaler must run on a quadri-core general-purpose processor with shared memory.

4.4 INRIA modeling of the downscaler

4.4.1 Overview

At the INRIA, MDE is used to generate code from a MARTE model. Their work is mainly focused on intensive signal processing modules such as video streams processing, of which the downscaler is a simplified example. They
use composite diagrams to model both the software and the hardware in the following way:

- The application TE represents the building blocks of the application;
- The application itself is represented on Figures 4.2 and 4.3;
- The application deployment;
- The architecture represents the hardware architecture;
- The task association models the association of the software with the hardware;
- The memory mapping models the mapping between the application and the memory.

We shall present the application, architecture and task association models in details.

4.4.2 Application

Let us first present the application. We have split its description between Figures 4.2 and 4.3, where the reader can see that the application is hierarchically described.

Another important fact about these figures is the use of composite structure diagrams, which are considered as static diagrams in UML and are thus intuitively not well suited to the representation of dynamic software. However, as previously stated, they focus mainly on stream processing, which is a fairly linear operation and can be basically described as an elaborate filter, which means there are no major state changes in the program. In a more general case, however, this would be impractical.

On these example diagrams, we can also note the presence of ports. In the original UML diagram, these ports are used to represent communication between services, which was not appropriate in this case. They have therefore enriched the original ports with several stereotypes, such as the flowPort and shaped types that we can see on Figure 4.3.

Finally, a last observation concerns the way they model loops in the program. Since the composite diagram by itself is not designed to incorporate loops, they had to find another way. One thing the composite diagrams do provide us with is the instanciation mechanism, i.e. the ability to define “classes” and then use “instances” of those classes, just as in most object-oriented languages. Their approach has thus been to create a class with a
Figure 4.2: Downscaler Application: Main application, the application, the downscaler
loop attribute, and set the value of this loop attribute to the number of loops to execute in the instance.

To clarify, if the horizontal downscaler had to run three hundred times, they would define a `Downscaler` class with a `looping` attribute, and in the instance they would set this attribute to 300.

### 4.4.3 Architecture

The description of the architecture is represented on Figure 4.4. We can observe that it, too, is represented by a composite structure diagram, which in this case makes more general sense. The actual architecture used in this case study is pretty generic: we have two processors, two cache memories, a RAM memory, a bus, a sensor and an actuator. So as we can see the model is fairly simple. According to the INRIA team members we have met on several occasions, they apply several successive transformations, and, during those transformations, many details are added.

### 4.4.4 Mapping

The mapping, or association, as the INRIA calls it, is the result of deciding which hardware piece executes which software component. For the case study, it is represented in Figure 4.5, once again with a composite structure diagram, which in this case makes a lot of sense given that both other components
Figure 4.4: Downscaler Architecture
use these same diagrams. On the diagram, we see for instance that the HDFrameGenerator would run on the sensor, whereas the filters would run on either one of the virtual processors.

In the next chapter we shall first see how we decided to model the inputs and why we did it that way. We shall then briefly show how we model the downscaler.
Figure 4.5: Downscaler Task Association
Chapter 5

Modeling and transformations applied to Nessie

In this chapter, we discuss the modeling and the transformation that enables the standardization of Nessie’s input as a MARTE diagrams.

First, we shall discuss how we have chosen to model electronic systems. For that, we created a segmentation of Nessie’s inputs which have been presented in Chapter 3, and modeled them differently according to their specificity. At the same time, we shall also present the model transformations that allow us to automatically generate most of the Nessie input files.

Then, we shall present the intermediate GUI that assists designers in completing Nessie models.

Finally, we shall conclude by presenting the overall process.

5.1 Modeling

We have segmented the models used by Nessie in three different categories:\footnote{A fourth one, which we did not focus on, should be added. It would serve the representation of the Platform-Software compatibility.}

Architectural model Represents the hardware architecture.

Functional model Represents the functional specifications of the system, meaning the application.

Non-functional model This is the analytical modeling of the components. It is the part of the model that serves simulation purposes exclusively.
In chapter 3, we have discussed the xml schemas that were provided by Alexis Vanderbiest. We have created an .ecore model to write the transformations. The .ecore metamodel is very similar to the xml schemas presented in Chapter 3, so we shall simply list the parameters or components that have been transformed with qvto. We shall also list the ones that have not been transformed with qvto and justify that choice.

We shall discuss each category by explaining how we modeled it (meaning that we have found the more appropriate UML representation for each of the models) and how we transform the model. A visual example will be provided in each section for clarity.

5.1.1 Architectural model

The hardware (or platform) is a hierarchical model where an abstraction level is based on several lower level abstraction primitives. As can be seen in the XML schemas discussed extensively in Appendix 2 of [28], the platform model in itself does not contain any information on the type of hardware material that the component represents. At each abstraction level, a HWSubType (a component) is described by unique identifiers and links between other components of the same abstraction level. The key concepts that should be easily represented are:

- The hierarchy that represents the different abstraction levels.
- A component, with its identifier.
- A link between components of the same abstraction level.
- I/O ports.

Composite structure diagram

Initially, this diagram was meant to display the inner structure of a class as well as the collaboration between classes. It is thus composed of the following elements, represented in 5.1:

Parts Represent a set of instances which are owned by a containing instance. For instance, if we had a class “car”, then “wheels” or “brake” could be parts of the instance.

Ports Represent the externally visible parts of a containing class instance. They define the interactions between a classifier and its environment.
Interfaces Represent the interface that the class implements, and the interfaces with which it can interact.

Connections Represent the links between the parts inside the class or the links between classes through the ports.

In Marte, the diagram has been enriched and to correspond to the needs of real-time and embedded systems\(^2\); for example ports can communicate not through services but by streams of data.

![Composite Structure Diagram Example](image)

Figure 5.1: Representation of the elements that compose composite structure diagrams, and an example of assembled elements

**Comparison between the XML model and composite structure diagrams**

If we want to model the architecture through a composite structure diagram, we shall have to translate that diagram to an XML file, as that is the input format that Nessie expects.

As the correspondence between the XML structure and the composite diagram may not be all that apparent, let us show through two simple examples how we can relate these two metamodels.

\(^2\)For more information see [1]
Please be aware that for the sake of clarity, we do not show complete, valid XML files, but only the most relevant parts.

In the simplest cases, the equivalence is pretty straightforward, as in Figure 5.2. The XML link is represented by the connection in the diagram, and the direction is represented by the directions of the ports. Notice how it is quite easy to imagine user interface improvements on the graphical representation, such as the use of an arrow or a different graphical representation for outgoing and incoming ports.

Of course, the XML in this example is pretty straightforward itself, and the gain might not seem so important. However, imagine that we have ten links between five components; the graphical representation will surely be much clearer, especially if we implement some extra improvements as discussed in the preceding paragraph.

The gain can be much clearer, however, when it comes to represent the hierarchy of abstractions. As can be seen on Figure 5.3, it is quite easy to represent an embedded component graphically, while the XML representation already begins to show some limitations.

Again, it is quite easy to imagine a user interface that could hide unnecessary levels of complexity by only displaying up to a given abstraction level; the designer could then click on a component and ask to zoom further down in the abstraction hierarchy.

Commercial UML modeling tools already have that feature, though free tools like Papyrus do not have it yet.

### 5.1.2 Functional model

**Petri nets**

The Nessie input format for the functional model is an XML representation of Petri nets. As for the architectural model, we shall have to explain how
our new proposed format can be related to the existing Petri net format. Towards that end, let us first describe Petri nets themselves.

A Petri net is a graphical representation of a workflow. Its basic elements are places, transitions and directed arcs; their graphical representations are drawn on Figure 5.4.

A Petri net is a bipartite, alternated graph, which means that each arc has to go from a place to a transition or vice-versa, but we cannot have an arc from a place to a place or from a transition to a transition.

Place from which an arc originates are called input places, and places to which an arc arrives are called output places.

The objective of a Petri net is to represent all the states in which the system can be as places, and all the possible transitions between these states.

In order to represent complex systems, Petri nets can be enriched with three more types of elements: forks, joins and tokens. Forks and joins allow the Petri net to model branches, such as if statements in a programming language. With places, transitions, arcs, forks and joins, we can model all the possible states of a complex system.

However, we are often also interested in knowing in which state the system currently is. That is the purpose of tokens: a token indicates that the system currently is in the state corresponding to the place on which the token is, or that the system is experiencing a transition if the token is on a transition. A Petri net on which tokens a represented is called a decorated Petri net, while...
the distribution of the tokens themselves is called a marking.

Tokens also allows for the representation of concurrency, simply by drawing more than one token on the decorated Petri net.

Activity diagrams

Like Petri nets, activity diagrams are used to represent workflows. They model activities and actions, and support the representation of choices, iterations and concurrency. The basic elements are activities, decisions and transitions, which are represented on Figure 5.5.

Activity diagrams can be recursively nested: an arbitrarily complex activity diagram can be encapsulated into an activity.

Comparison between activity diagrams and Petri nets

Again, as our objective is to enable designers to use activity diagrams instead of Petri nets to design the software part of their system, we have to show that it is indeed possible to establish a bijection between these two representations, or at least a surjection such that any activity diagram can be translated to a Petri net.

To that end, we only have to show the correspondence of each basic element, which is done in Figures 5.6, 5.7, 5.8 and 5.9.

There are two main differences. The first one is that activity diagrams allow a hierarchical representation, which is closer to the Nessie metamodel. The second one is that, in activity diagrams, we cannot find any element
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

Figure 5.5: Representation of the elements that compose activity diagrams, and the representation of a valid sequence of elements

Figure 5.6: Equivalence of the action element

Figure 5.7: Equivalence of the transition element
Figure 5.8: Equivalence of the begin and final elements

Figure 5.9: Equivalence of the Fork and Join elements
that would correspond to tokens. This is a serious limitation to represent massively parallel systems, such as the downscaler where we have loops that have to iterate around 300,000 times. Another limitation of the tool that we have been using, Papyrus, is that it does not allow us to represent activity diagrams.

5.1.3 Non-functional model

The non-functional model was more difficult to apprehend than the two previous models. From the start we wanted to think the overall process in order to respect two features:

**Simplicity** The success of a software or an idea is usually highly correlated with its simplicity.

**Independence** For the sake of portability and re-use, we prevented ourselves from mixing the non-functional part of the model with the platform or the functional parts of the model.

A great addition to this list would be productivity. While it is not of direct importance in the overall design of a workflow, as we are doing here, if this workflow was ever to be actually implemented through software, productivity would have to be amongst the top priorities. No detail should be spared: there should be no unnecessary click, no unintuitive behavior that can bite the designer in the back, etc.

With these two aspects constantly in mind, and after testing several possibilities, the solution we came up with was to create a graphical user interface to assist the designer in decorating the platform and functional models before sending them to Nessie. The description of this GUI is the subject of Section 5.3.

**Behaviors**

The main part of the non-functional model are the behavior files of each component. As we have presented in Chapter 3, each component has several behavior files that describe its behavior in different situations such as computing, idle, memorizing and so forth. Creating and filling all these files can quickly become a tedious, error-prone job. For the non functional properties this has been our focus. The mechanism we came up with is to model the dependance graph of a behavior graphically as shown in Figure 5.10.

For the modeling we chose state diagrams which can serve to represent those graphs. As a proof of concept, we have worked on the “Coderescu” example explained in [28], page 69.
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

Figure 5.10: source: Alexis Vanderbiest

Figure 5.11: Modeling the behavior through state graphs
Figure 5.11 is the state-diagram representation of Figure 5.10; they are almost identical. The various relations are correctly generated with the corresponding parameters. This creates the behavior files backbone and fills up the file partially. As a last step, the GUI asks the designer to fill in the analytical expressions of the behaviors to complete the files for Nessie consumption.

5.2 Transformations

In the previous section, we have presented new metamodels for describing the Nessie input format. As previously stated, for this new approach to be of any use, there has to be some way to transform from our proposed metamodel to the actual metamodel of Nessie input. In the previous section, we have also shown that these transformations were, in principle, possible.

We did not stop at a theoretical possibility, though. These transformations are not only possible in principle, they are possible in practice and can even be automatized, as we indeed have done. This section presents some of these automated transformations in details.

First, we shall present the difference between a mapping and a transformation by explaining the concepts of mapping, query and helper. Then we shall briefly present the behavior transformations because they are fairly short and simple. Readers interested in the full set of transformations we have developed can find them in appendix A.

5.2.1 Mapping, query and helper

All mappings are transformations but not all transformations are mappings. Sometimes when the transformation is not straightforward there is a need for intermediate transformations and this is done with query and helper.

Certain elements of the destination .ecore contain lists of elements. For instance: “RelationListType” contains a field “relation” which is a list of “RelationType”. A mapping always has to have an outgoing element that is present in the destination .ecore. Hence it is impossible to have a mapping that would look like this:

```plaintext
mapping input::function() : list(something)
```

This is the reason why we use a query. Basically, we decompose it in two steps:

```plaintext
mapping input::function() : list(something)
```
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

mapping input::relations() : RelationListType{
    relation := self.realComputingOfTheRelations();
}

query input::realComputingOfTheRelations() : Set(RelationType){
    //The list is computed...
    return theList;
}

A helper is just like a query, except that it can also accept “inout” parameters, which can be modified inside the helper.

5.2.2 Example

As an example, let us present the transformation of a state diagram into a Nessie behavior, as was presented in the previous section. Each state of the diagram is a parameter of the behavior (see Figures 5.11 and 5.10). For each state, the arrows represent the functional dependencies between the parameters. The entering parameters are the entry point of the graph (node without a predecessor), the outgoing parameters are the exit points of the graph (without successors).

mapping Package::behaviour() : Nessie::BehaviourType{
    self.allOwnedElements()[StateMachine]->forOne(machine){
        name := machine.name;
        relationList := machine.map relations();
        orientationList := machine.map orientation();
    };
}

This is the part of the code where the behavior is generated. Most of the backbone is generated.

mapping StateMachine::orientations() : Nessie::OrientationType{
    var inputs : Sequence(String);
    var outputs : Sequence(String);
    self.allOwnedElements()[State]->forEach(state){
        if state.isinitial() then {
            inputs += state.name;
        }else{
            if (state.isfinal()) then{
                outputs += state.name;
            }
        }endif;
    }
There, the input and the output of the behavior are extracted. An input (for instance) verifies the condition that it has no predecessor. That would be implemented as such:

```java
query State::isinitial() : Boolean{
    return (self.incoming->size() = 0);
}
```

Verifies if a state is final. This is an example of use of query.

### 5.3 Graphical user interface

The transformations of the preceding section lead to intermediate xml files. As shown on Figure 5.12, the graphical user interface is a bridge between the result of the transformations and the final simulation file.

The GUI is still in development and is thus not complete. However, the relation between the hardware and the behaviors have already been implemented. For each behavior, we know:

- In how many relations the behavior has to be decomposed.
- Which are the dependencies between the different parameters.

The only thing that is left is to specify these dependencies via analytical relations or association tables.

The GUI also allows to determine some other characteristics such as the timeDependent field.

The GUI has been written in Python.

#### 5.3.1 The main window

In Figure 5.13, we can see on the left side of the window the part of the file that follows the exact same tree structure as the XML file that has been generated (and that passes Nessie’s parser). We can edit the criteria and edit and link the behaviors to the states of the hardware structure.
Figure 5.12: The overall process
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

Figure 5.13: The graphical aid to fill in Nessie’s entries
5.3.2 Criteria editing

When the criteria element is selected on the left side, an interface such as the one shown in Figure 5.14 appears, where we can edit the “timeDependent” and “combination rule” criteria.

5.3.3 The main window

![Criteria edition window](image)

Figure 5.14: Criteria edition window

5.3.4 Behavior editing

If one of the states of the hardware structure (Idle, Sleeping Memorizing, Transmitting, Inactive, Receiving, Sending) is selected on the left side the window presented in Figure 5.13 appears.

On the right side, we can see the elements that have been generated by the transformation, after charging the behavior generated by the qvto transformation.

The “edit the relation” button allows the editing of the relation. The relation can be represented by a table, as shown in Figure 5.15 or by an analytical expression, as show in Figure 5.16. Both can be edited.

After clicking on the button, a new pop up window appears. There, the analytical expression can easily be edited. The GUI checks the syntax and provides autocompletion.
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

Figure 5.15: The table edition window

Figure 5.16: The analytical expression edition window
It is only the beginning of the GUI, but we will keep it as simple as what we have done so far.

For more specific information, the reader should refer to Sebastien Leduc’s documentation: “Design de plateforme électronique via UML”.

5.4 Modeling the downscaler

In Chapter 4, we have started to describe the downscaler application that is the main example the INRIA is working on. In order to allow the reader to compare our approach with theirs, we shall now show how the same downscaler would be modeled in our framework.

With what we have studied above, we shall now present the modeling of the downscaler. The reader has to note, however, that since Nessie can map different variants to estimate their performances we have had to create variants as well.

5.4.1 The application

Let us start with the application. As explained above, we represent it through an activity diagram. In Figure 5.17 we represent the main, which apart from the fact that the diagrams chosen are not same, resembles the main represented by the INRIA. It is defined with more details and two variants in Figures 5.18, 5.19 and 5.20.

![Figure 5.17: Downscaler Application : Abstraction 0 , the main application](image)

5.4.2 The hardware

Figure 5.21 shows the modeling of the hardware. For the moment, since Papyrus does not allow an easy visualization of the different levels of details as other tools would, we have represented the different levels by arrows. The arrow points to its higher level of abstraction.
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

Figure 5.18: Downscaler Application : variants

Figure 5.19: Downscaler Application : variant with serial RGB filters

Figure 5.20: Downscaler Application : variant with Horizontal and Vertical filters
CHAPTER 5. MODELING AND TRANSFORMATIONS APPLIED TO NESSIE

5.4.3 Discussion on the INRIA’s technique

INRIA’s technique makes sense for the type of applications they are working on: signal intensive, massively parallel and sequential. Software is described as an assembly of several software building blocks represented with a static composite diagram for mainly three reasons:

- First, they do not need the behavior diagrams offered by UML 2.0 (see Chapter 2), and it is technically much easier to make the association (mapping) between a hardware class and a software block if both are represented by the same diagram. The transformations that follow are naturally less complex, since the focus is only on composite diagrams.

- Second, the composite diagram offers the notion of flow-ports, which is used extensively in the applications that are treated. These ports represent the data streams that have to be treated by the application.

- Third, they kept the hardware really simple, which is a good thing. The drawback here is that they do not allow any modification and, as a user, we have to know how the tool (GASPARD) works internally in order to know what will happen to the platform model.
Overall, this makes their approach quite effective in their selected field of application, but it does not seem to be an adequate model for a more general use.
Chapter 6

Design with performance estimation/multicriteria tools

As we announced in the introduction, we shall now present the bigger picture and explain where the present work fits in the grand scheme of all things electronics.

In this chapter, we shall first quickly review our development so far, underlining the most relevant elements to the proposed thesis. We shall then show how these elements can lead to a new approach to electronic circuit design, and then proceed to demonstrate the new design on a small example. We conclude this chapter with a few open ideas that would be worthy further exploration.

6.1 Recapitulation

In the preceding chapters, we have presented and analyzed the models with which Nessie can work. We proposed to separate Nessie models into three main components, namely the architectural or hardware model, the functional or software model, and the non-functional model.

We have then compared each of these three components to UML in search for a minimal subset of the UML diagrams that would correctly and completely represent our model. We reached the following conclusion. The architectural model is best represented by the composite structure diagram, the functional model should be represented by either an activity or a state diagram, and we have found no direct match between the non functional model and the UML diagrams, though we did find some partial coverage through the state diagram. See Section 5 for more on why this coverage is only partial.
Beyond the simple representation of the three parts of our model, we need some way to express the possible combinations between all the possibilities for each part.

In order for these UML diagrams to be of any use, they have to be translated in a format that Nessie can understand. Since UML diagrams are models, we can define (and have defined!) transformations from these three models to the Nessie input model. Unfortunately, due to some conceptual mismatch between UML and the Nessie input format, and due to some limitations of currently available UML tools, some information cannot be put in the original UML models. To account for these, a GUI is being developed to fine-tune the output of the transformation and add the missing data.

Writing XML files directly is tedious and error-prone. Allowing for a way to generate these XML files through graphical authoring of UML diagrams is a huge productivity booster, both in terms of designing the models, as the visual aspect helps the designer keep the bigger picture in mind, and in terms of debugging, as parsing errors are going to be much rarer with generated text. Furthermore, the GUI fine-tuner helps completing and verifying the model.

6.2 Towards a new design workflow

Now that we know what has been done, let us insert these results in a design flow. The problem stated in the introduction is that for a system to be efficient, time needs to be spent on the modeling and design. So what many have been studying for years now is how to make the design flow more efficient, meaning that it leads to optimized systems (in regards of one or several designer defined criterion) in lesser time.

We created our solution bearing in mind the two factors that can make design, and in fact any process, more efficient:

1. Keep it simple!
2. Verify the work as much as possible.

Coding the entire model is not visual, nor easily understandable. Therefore, in the real world, models are seldom reused. However, it is very flexible and accurate. Ruling over a graphical dictatorship is also a trap in which it is best not to fall. Although graphical models certainly possess advantages over text solutions, there is always a point were more accuracy cannot be obtained without dramatically complexifying the model. The solution we came up with requests only straightforward graphical modeling. Afterwards,
the graphical model can be enriched with more textual modeling through the assistance of a simple GUI. Our solution respects rule number 1, because it combines the simplicity of graphical modeling with the help of a GUI.

Intuitively, when creating anything, we first think of it globally. We then refine the model, step by step. At each step, variants can be thought of. Variants add possibilities, refinement adds detail to the possibilities. Hence, it becomes important that at each step of the refinement process, variants are pruned to keep only the better ones. What happened with Nessie is shown in Figure 6.1. The combinatorial explosion of the number of variants at the lowest levels prevents the designer from exploring all of them.

Our solution, presented in Figure 6.2, is to run the simulation at each step and refine only the best variants to control the number of models to create and to explore.

It is both more comfortable for the designer, which has a more limited number of variants to think of and to explicitly model, and for the computer, which has less variants to simulate. That way, we can gain a lot of time on both preparing the input files for Nessie and running Nessie simulations. This is in addition to the productivity boost coming from the usage of graphical models and GUI presented in Chapter 5.
6.3 Generalization of the design approach

These two concepts lead us to the generalization of the design process shown in Figure 6.3.

6.3.1 Step by step explanation

The methodology that we propose is the following:

Let us start with Figure 6.4. It represents a general iteration in our proposed methodology; say we begin at abstraction level \( k \). At that level, we have already designed both the functional and architectural models, each with a few variants, in a graphical modeling language such as the one presented in Chapter 5. We also already have the non-functional model. Notice, however, that we have not yet defined anything at abstraction level \( k + 1 \). This is the most important aspect of this methodology.

In the first step of this iteration, the functional and architectural models, along with all their variants, and the partial non-functional model are transformed and sent to the GUI. The GUI is used to complete the models with the last details, and all those information are combined into input files for the simulator, which could be Nessie. The most important characteristic of the simulator, for this design flow to work, is that it has to be capable of working at multiple abstraction levels, just like Nessie.
Figure 6.3: New efficient design paradigm
As shown in Figure 6.5, the simulator then runs all the simulations and selects the best combination of variants of the functional and architectural models.

The designer can now refine these variants and start anew at abstraction level $k + 1$, as illustrated by Figures 6.6 and 6.7.

This means that the simulator would be used as a fast validator throughout the design process. The point is that if architectural and functional models are separated from the non-functional models needed for the simulation, then they can be used and detailed step by step, instead of having to rewrite the entire model at each abstraction level. Since the simulator is fast in exploring the design space and that part of the models it uses can be reused, the process can lead only to more efficient systems, since at every stage a confirmation is given by the simulator, and it can also be faster as graphical editing is faster and less error-prone.

Figure 6.4: Step 1 : Creating variants and passing all the models to the GUI

6.4 Example

To clarify, let us show an iteration step on a concrete example, such as the downscaler discussed in Chapter 4.

As a reminder, the functional model and the architectural model have to be simple, reusable, and refinable so they are mostly modeled in UML Marte. The transformations generate XML files which are sent to the simulator.
Similarly, the simulator returns an XML file. The rest has not been coded yet. An example is provided in Figure 6.8; we show the 3 variants of downscalers and three variants of platforms. On this example it is the horizontal-vertical version of the downscaler that is selected with the 2-processor version of the platform. Should the designer wish to keep on refining the model, she can focus on these two variants.

6.5 Integration of multi-criteria tools

At the time of writing, the design process described in the preceding sections is only a proposition: it has not been fully implemented yet. Most notably, even though all the building blocks exist as independent proofs of concept, we would want a single software that seamlessly integrates all of these steps.

That software would have to integrate the graphical editing of MARTE models, the last few steps that our GUI currently provides, the running of the simulation and the presentation of the results to the designer. For that last step, the software should provide some multi-criteria decision aid to guide the designer.

This multi-criteria decision aid (MCDA) could also help Nessie in his computations. Indeed, Nessie currently has to exhaustively try all possible combinations, which can take quite a lot of time when there are many com-
Figure 6.6: Step 3: Only the best variants are refined.
**CHAPTER 6. DESIGN WITH PERFORMANCE ESTIMATION/MULTICRITERIA TOOLS**

Figure 6.7: Step 4: The same process takes place again, at level k+1

Figure 6.8: An example with the 3 variants of downscalers and three variants of platforms
plex variants, as the number of combinations grows exponentially with the number of variants. The MCDA could thus guide Nessie in order to reduce the number of combinations to try out.

In [13] the departments of BEAMS and CODE-SMG are working on ways to perform multi-objective optimization and MCDA. By focusing on the increasingly complex designs of 3D-SICs\(^1\), they seek new solutions to improve and simplify the design. Multi-objective optimization would yield the automatic generation of optimized variants of platforms for a given application.

In Figure 6.9 we show a way to integrate this research field within our design flow. The parts where these techniques would apply are represented in blue. MCDA 1 represents the automatic generation of optimized variants, whereas MCDA 2 represents the multi-criteria decision aid tool.

![Figure 6.9: The design paradigm with an MCDA tool](image)

6.6 Using libraries of components and architecture

The good news is that the proposed paradigm is logical and flexible which allows for even more improvement. For example, it is easy to imagine that,

\(^1\)3D stacked integrated circuits. The principle is to “stack” traditional integrated circuits and to interconnect them which is considerably more complex than tradition circuit design as we have to consider a third dimension.
over time, we could build libraries that would contain elements of the func-
tional, architectural or non functional model so that modeling can keep on
getting faster and on allowing designers to work on ever higher levels of
abstraction.
Chapter 7

Reflexions on the role of MDE

The original purpose of the present work was to analyze the model-driven engineering methodology and the possibilities that its application to hardware design could open. In the course of this work, we have uncovered some obstacles to MDE adoption in electronic circuits design; this chapter proposes some reflexions on these obstacles.

While there certainly are some purely technical challenges that the MDE evangelists will have to tackle in order for MDE to gain more grounds in electronics design, we believe that some of the non-technical issues are at least as important, and we would like to present some of them.

In the next sections of this chapter, we shall present what we perceive as the most important influences on the future of MDE in electronics, and we shall propose some fields that should be explored in order to further the global understanding of the context, constraints and needs specific to electronics design, and how MDE can be adapted to them.

7.1 Roots in the software world

Model-driven engineering originated in the software world, and the idea was mainly popularized with the advent of UML. However, even though the idea is relatively well known, the actual practice of MDE does not seem to be so commonplace. Indeed, one has the impression that UML diagrams are seldom drawn in advance, and even more rarely used to generate code. Usually, the best you can expect is that they are generated from the actual code, while the most common scenario seems to be that they are there only for documentation purposes and are not really kept up-to-date.

This scepticism in the software world is strong enough that researchers decided to try to assess the actual use and efficacy of MDE in practice [30].
CHAPTER 7. REFLEXIONS ON THE ROLE OF MDE

This is, of course, somewhat glum for the perspectives of MDE in electronics circuit. After all, why try to emulate something that does not work in its original context?

We strongly believe that MDE can greatly improve the productivity of electronic circuits designers. The perceived advantages of MDE include the following:

- Increased productivity
- Eased maintenance and portage of software to different hardwares
- Better communication of ideas through the more abstract representation
- Better tools to reflect on problems on a more abstract level
- Development and use of domain specific languages
- Automatic transformations from models to models

However, while we believe that these improvements are actually achievable, we also have to express some concerns about the way MDE is currently implemented.

7.2 Usage in the real world

As we have seen through our work with the INRIA center in Lille, France, usage of MDE in the real world generates some unforeseen problems. For example, their exclusive usage of composite diagrams seems to have led them through some troubles.

Their somewhat narrow field of applications also allowed them to make some simplifying assumptions that cannot be made in a more general case.

For instance, they represent software with composite diagrams, because that allows them to use the Class/Instance mechanism, which among other things enables them to roll out loops. Arguably, in the general case, composite diagrams are not suited to representing software, as they are static diagrams while software is generally perceived as dynamic, in their particular application it is a valid approach, as they consider software as immutable building blocks.

Consequently, we believe it would be quite difficult to try and integrate Nessie, which has a broader scope, with the work currently being done at the INRIA.
More generally, this shows at least two limitations with the way MDE is currently done in the real world out there, namely that there is currently no standard way to describe a system — and it is quite easy to see that most of the promised benefits of MDE depend on a common language —, and that the modeling paradigm we presented in Section 2.3.4 might not be universally respected.

7.3 UML and MARTE

Just like UML, MARTE is a toolbox rather than a methodology, which means that there is no more a standard way to use MARTE than a standard way to use the roman alphabet — every language has its own words, but also its own grammar.

This ‘language impedance’ is a major obstacle, as it prevents both communication and the development of automated translators, just as it does for human languages.

7.4 Current tools

Another huge obstacle for MDE adoption today is the quality and completeness of available tools. People who are not already MDE fanatics are not going to pay the enormous license fees of the commercial products, and the main free tool, namely Papyrus, is far from being ready for prime time.

The first, most obvious limitation of Papyrus is its incompleteness: it only implements four out of the thirteen UML diagrams. This is by itself a show-stopper for many use cases.

The other problem with Papyrus is common to most free open-source softwares, namely user-friendliness. This can be seen as a subjective matter, and it is indeed quite hard to properly assess, so let us give two example circumstances in which Papyrus behaves in an inconsistent or unexpected way that can alter productivity.

The first example is that there is no easy way to define a hierarchy of states in a state diagram, by which we mean that the user is not allowed to define the level of abstraction he wants to see: all the details have to be displayed at all times.

The second example is much less obvious and potentially much more perturbative. When the user deletes an element of a diagram he is editing, the element is only removed from the graphical representation of the model. This means that the element is no longer displayed, but is still very much
CHAPTER 7. REFLEXIONS ON THE ROLE OF MDE

present in the model files. This might not be too much of a problem if
the model is only used for its own visualization, but as soon as you start
using transformations on it, this is an invitation to really nasty, hard-to-find
bugs.
Chapter 8

Conclusion

8.1 Contributions

The two biggest contributions of the present work are the following.

The first one resulted from the study we conducted in order to standardize the inputs of Nessie. We have formalized the way to think about the models in terms of a functional model, an architectural model, and a non functional model. We then proceeded to study the adequacy of UML diagrams with these three models, and we found out which diagrams best represent each of the three types of models.

On a more practical level, we also wrote the transformations which stand both as a proof of concept and as a basis for creating a toolbox that can be used in the entire standardization process. We have been able to transform most of the functional and architectural models, and about half of the non functional one. Furthermore, we have designed a GUI to allow the designer to add the missing details manually, in a much more convenient and less error-prone way than directly editing the transformed model files. Overall, the whole experience of using Nessie has been made much more pleasant and productive.

The second contribution is the new design methodology. By taking a step back, we figured out a way to adapt the design flow to the constraints that have been presented in the introduction. In this workflow, we have step by step performance simulation, variant generation and multi-criteria decision aid. It is a highly efficient concept, that is also flexible so it is prone to many improvements.
8.2 Perspectives

We have been at the very beginning of the study and brought out many interesting perspectives in several fields of study that would be worth analyzing. We will cite only a few for the short-term and long-term ones, however it should give ideas to those who would like to take this study to the next level.

- **Short-term perspectives:**
  
  - Market study: We are at the beginning of a standard. It has been well thought out and it could really dramatically improve the quality and the productivity. In order to be sure to really solve a need and a problem and to be sure of the way it should be solved, it is essential to find out how things are actually done on the field. Hence, we would suggest to study the industrial electronic design in order to have a better understanding.
  
  - Programming: Enable the user of the GUI presented in Chapter 5 to transform the UML models from the GUI and to launch Nessie from the GUI as well.
  
  - Study: Complete this study, which, so far, has only focused on the main parts of the problem. Also do the reverse transformation to enable the multi-criteria decision aid to operate. This would be the second step to integrate Nessie with the design flow proposed in Chapter 6.

- **Long-term perspectives:**

  - Study: Adapt the study to the market study conducted in the short-term perspective.
  
  - Programming: Contribute to the Papyrus project. As we have seen in Chapter 7, tools can greatly influence the success of the technology.
Bibliography


BIBLIOGRAPHY


Appendix A

QVT Transforms

1 import library Strings;
2 modeltype UML uses 'http://www.eclipse.org/uml2/2.1.0/UML';
3
4 modeltype nemo uses Nemo('http://beams.ulb.ac.be/nemo');
5 modeltype marte uses 'http://org.omg.marte';
6 modeltype MM uses MARTE('http://org.omg.marte');
7 modeltype Nessie uses Nemo::nessieSimulationType('http://beams.ulb.ac.be/nemo');
8 modeltype.ecore uses "http://www.eclipse.org/emf/2002/Ecore";
9 transformation Downscaler(in software : UML,in hardware : UML,out target : Nessie);
10 intermediate property UML::dof : Set(Nessie::StructureDOFtype);
11 intermediate property UML::crit : Sequence(String);
12
13 main() {
14 xmap createsimu();
15 }
16 }
17
18 mapping createsimu() : Nessie::nessieSimulation{
19 software.rootObjects()[Package]->forEach(pack){
20 //Transformation de l'architecture software
21 SWdescription := pack.map statemachine();
22 //Calcul des différents degrés de liberté pour le software
23 //ce sont les différents variants de chaque niveau d'abstraction
24 DOF := pack.map doftype();
25 }
26 hardware.rootObjects()[Package]->forEach(pack){
27 //Extraction des critères
28 criteriaList := pack.map criterias();
29 //Transformation de l'architecture hardware
30 HWdescription := pack.map hardware();
31 }
32 }
33 */
34 /* Ajout des DOF au modèle
35 Ces-ci sont calculés dans la variable globale 'source.dof' au court de la
36 ...transformation
37 */
38 mapping Package::doftype() : Nessie::DOFtype{
39 structureDOF := software.dof;
40 xmlns := "xmlns"; //facilite le post-traitement
41 }
/** A partir d'une statemachine, on calcule 
- swhierarchy : une liste de tous les niveaux d'abstraction software 
- SWstructuresList : Pour chaque "Statemachine", les differentes organisation 
  ...internes 
(reseau de petri) */

mapping Package::statemachine() : Nessie::SWdescriptionType{
  SWhierarchy := self.map swhierarchy() ;
  SWstructuresList := self.map swstructureslist();
  xmlns := "xmlns"; //facilite le post-traitement ;
}

/* On part du diagramme complet, et on calcule swhierarchy. */

mapping Package::swhierarchy() : Nessie::SWhierarchyType{
  PHASE 1 : On utilise un dictionnaire, representant une table d'association
  niveau d'abstraction => liste des structures logicielles qui
  ...sont a ce niveau.
  On parcourt la SubMachine "Main", et on descend recursivement dans
  ...les SubMachines qui la composent, en mettant a jour le dictionnaire.

  var dico : Dict(Integer,OrderedSet(Nessie::SWtype));
  //On extrait la statemachine de nom "Main"
  self.allOwnedElements()[StateMachine]->forOne(machine | machine.name = "Main"){
    machine.abstractions(0,dico);
  };

  PHASE 2 : Le dictionnaire etant maintenant rempli, on cree les objets "
  ...AbstractLevelType" a partir de 'dico'.
  Les differents "AbstractLevelType" sont stockes dans la variable
  ...res, qui est ensuite
  passe au champs 'abstractionLevel' du resultat.

  var res : OrderedSet(Nessie::AbstractLevelType1);
  dico->keys()->forEach(cle){
    res += xmap createabstract(cle,dico->get(cle));
  };
  abstractionLevel := res;
}

/* Fonction permettant de generer un AbstractLevelType1 partir de la liste des 
...subtypes et 
  du niveau d'abstraction. */

mapping createabstract(lvl : Integer, lst : OrderedSet(Nessie::SWtype)) : Nessie::
  ...AbstractLevelType1{
SWsubTypes := lst;
abstractionLevelNumber := lvl;
}
/**
Cette fonction parcourt toutes les submachines d'une machine courante, et s'
appelle récursivement.
A chaque appel, on cree l'abstraction du niveau courant (phase1), puis on l'
appelle récursivement
sur toutes les submachines qui la composent.
*/
helper StateMachine::abstractions(lvl : Integer,inout dico : Dict(Integer,OrderedSet(
...Nessie::SWtype))) {
  // phase 1 : On cree l'abstraction du niveau courant
  self.createabstraction(lvl,dico);
  // phase 2 : On parcourt la liste des submachines qui compose la Statemachine de
  // ... depart, et
  // on appelle récursivement la fonction.
  self.allOwnedElements()[State]->forEach(elt | elt.isSubmachineState)
  {
    elt.submachine.abstractions(lvl+1,dico);
  }
}
/**
Cette fonction cree un niveau d'abstraction, dans le sens ou elle remplit le
... dictionnaire
avec tous les etats qui la composent
*/
helper StateMachine::createabstraction(lvl : Integer,inout dico : Dict(Integer,
...OrderedSet(Nessie::SWtype))){
  // On recupere le dernier id qui etait present a ce niveau d'abstraction dans le
  // ... dictionnaire,
  // pour pouvoir ajouter les nouveaux elements du meme niveau a la suite.
  var id : Integer := getdicoid(lvl,dico);
  // On parcourt les etats qui ne sont pas des submachines (pour lesquels il n'y a
  // ... pas de recursion a faire)
  self.all OwnedElements()[State]->forEach(elt | (not(elt.isSubmachineState) and not
  // ...(elt.getLabel() = null)){
    if(elt.resolveone(Nessie::SWtype) = null) then{
      addtodico(lvl,elt.map swsubtypes(id),dico);
      id := id +1;
    }endif;
  }
  // On parcourt les etats qui sont des submachines, et on appelle la fonction
  // ... addtodico sur la submachine en question
  self.allOwnedElements()[State]->forEach(elt | elt.isSubmachineState){
    if(elt.submachine.resolveone(Nessie::SWtype) = null) then{
      addtodico(lvl,elt.submachine.map machinesubtypes(id),dico);
      id := id +1;
    }endif;
  }
}
/**
Comme explique dans la fonction precedente, cette fonction calcule le
... plus grand id present dans le dictionnaire a un niveau d'abstraction donne.
S'il n'y a aucun element au niveau d'abstraction donne, on renvoie 0.
Sinon on renvoie le nombre d'elements dans la liste associee au niveau donne.
*/
query getdicoid(lvl : Integer,dico : Dict(Integer,OrderedSet(Nessie::SWtype))) : ...
   Integer{
APPENDIX A. QVT TRANSFORMS

var ret : Integer := 0;
if(lvlexistindico(lvl,dico)) then{
    ret := dico->get(lvl)->size();
}endif;
return ret;

/**
En cree un SWtype a partir de son id.
*/
mapping State::swsubtypes(id : Integer) : Nessie::SWtype{
    ID := id;
dataIn := 1;
dataOut := 1;
SWparametersList := self.map swparamlist();
name := self.name;
}

mapping State::swparamlist() : Nessie::SWparametersListType{
    SWparameter := Set{};
    // ////////////// TODO
    ... : A remplir !!
}

/**
Comme la fonction precedente, mais s'applique aux statemachines
*/
mapping StateMachine::machinesubtypes(id : Integer) : Nessie::SWtype{
    ID := id;
dataIn := 1;
dataOut := 1;
SWparametersList := self.map paramlst();
name := self.name;
}

mapping StateMachine::paramlst() : Nessie::SWparametersListType{
    SWparameter := Set{};
    /// TODO A REMPLIR
}

/**
On ajoute au dictionnaire un SWtype, avec son niveau.
Pour cela on extrait la liste de SWtype associee au niveau en question, et on y ...
ajoute l'element SWtype.
*/

def addtodico(lvl : Integer,type : Nessie::SWtype,inout dico : Dict(Integer,
    ...OrderedSet(Nessie::SWtype))){
    //La nouvelle liste a laquelle on va rajouter l'element
    var newlst : OrderedSet(Nessie::SWtype);
    //Si le niveau existe
    if( lvlexistindico(lvl,dico)) then{
        //On extrait la liste, et on y ajoute l'element
        newlst := dico->get(lvl);
        newlst += type;
    }else{
        //Sinon on ajoute directement l'element a newlst qui etait vide
        newlst := type;
    }endif;
    //On place au niveau 'lvl' la liste mise a jour
    dico->put(lvl,newlst);
}
APPENDIX A. QVT TRANSFORMS

202 /**
203 * On verifie si un niveau d'abstraction donne existe dans le dictionnaire ou pas
204 * (ie si on a deja ajoute des elements a ce niveau d'abstraction)
205 */
206 helper lvlexistindico(lvl : Integer, dico : Dict(Integer, OrderedSet(Nessie::SWtype))):
207 ...: Boolean {
208     dico->keys()->forEach(cle) {
209         if (cle = lvl ) then {
210             return true;
211         } endif;
212     } endif;
213     return false;
214 }
215 } end(code)
216 /**<
217 ..........................................................CALCUL de sustructureslist
218 ..........................................................
219 */
220 /**<
221 * On extrait la machine de nom "Main" pour commencer la transformation.
222 * S'il n'existe pas de machine avec ce nom, on renvoie une erreur.
223 */
224 mapping Package::sustructureslist() : Nessie::SWstructuresListType {
225     var res : Set(Nessie::SWstructureType);
226     var ismain := false;
227     self.allOwnedElements()[StateMachine]->forOne(machine | machine.name = "Main") {
228         ismain := true;
229         SWstructure := variantecalc(machine, -1, 0);
230     });
231     if (not ismain) then {
232         log("ERREUR/uni2423:/uni2423Aucun/uni2423bloc/uni2423'Main'/uni2423n'a/uni2423ete/uni2423trouve/uni2423dans/uni2423l'architecture/uni2423logicielle
233             ...
234             ...
235             ...");
236     } endif;
237 }
238 /**<
239 * Pour une StateMachine donnee, on parcourt toutes les "regions" qui la composent,
240 * et qui representent une variante de l'architecture en question.
241 */
242 query variantecalc (machine : StateMachine, lvl : Integer, swtype : Integer) :
243     ... OrderedSet(Nessie::SWstructureType) {
244     var tmp : OrderedSet(Nessie::SWstructureType) := OrderedSet();
245     var index : Integer := 0;
246     // Pour chaque region contenue dans la StateMachine
247     machine.allOwnedElements()[Region]->forEach(region) {
248         // On calcule la variante correspondante
249         tmp += onevariante(region, lvl, index, swtype);
250         index := index +1;
251     };
252     return tmp;
253 } end(query)
254 /**<
255 * Pour une region donnee, on calcule la SWstructureType correspondante,
256 * et on fait les appels recursifs pour calculer les variantes de toutes les
257 * StateMachine contenues dans la region.
258 */
259 query onevariante(region : Region, lvl : Integer, index : Integer, swtype : Integer) :
260     ... OrderedSet(Nessie::SWstructureType) {
83

### APPENDIX A. QVT TRANSFORMS

```plaintext
var tmp : OrderedSet(Nessie::SWstructureType) := OrderedSet();
// On calcule la SWstructure de la region courante
if (lvl != -1) then{
    tmp += region.map swstructure(lvl,index,swtype);
}endif;

// Pour chaque StateMachine contenue dans la region
region.allSubobjects()[State]->forEach(obj | obj.isSubmachineState){
    // On appelle recursivement le calcul des variantes de cette submachine a un
    // niveau d'abstraction superieur
    tmp += varianteCalc(obj.submachine,lvl+1,obj.submachine.resolveone(SWtype).ID ...);
};
return tmp;
```

```plaintext
/**
 * Generation d'un objet SWstructureType avec comme entrees une region , le niveau d'
 * abstraction courant , le numero d'index de la structure et l'id du type du bloc considere
 */

mapping Region::swstructure(lvl : Integer,structindex : Integer,swtype : Integer) :
    ...Nessie::SWstructureType{
    // Generation du reseau de petri correspondant a l'architecture contenue dans la
    ...region d'appel
    petriNetwork := self.map petrinetwork(lvl);
    SWtypeID := swtype;
    abstractionLevel := lvl;
    structureIndex := structindex;
    name := self.name;
    add_dof(HWSWstringType::SW,lvl,swtype,structindex);
}

/**
 * Calcul du reseau de petri equivalent au contenu d'une region
 */

mapping Region::petrinetwork(lvl : Integer) : Nessie::PetriNetworkType{
    // Calcul de la liste des noeuds constituant la variante en entree
    netList := self.map netlist();
    // Calcul de la liste des transitions de la variante
    transitionsList := self.map transitionlist();
}

/**
 * Calcul de la liste des noeuds de la variante
 */

mapping Region::netlist() : Nessie::NetListType{
    var tmp : OrderedSet(Nessie::PetriNetType) := OrderedSet();
    var id : Integer := 0;
    var swtypeid : Integer := 0;
    log(self.name);
    self->forEach(elt){
        log(elt.repr());
    };
    // Pour chaque State qui n'est pas un PseudoState (ni initial, ni final, ni
    ...transition)
    self.allOwnedElements()[State]->forEach(state | (not(state.getLabel() = null))){
        // On genere un noeud a ajouter dans la NetList
        if(state.isSubmachineState) then{
            swtypeid := state.submachine.resolveone(SWtype).ID;
        }else{
            swtypeid := state.resolveone(SWtype).ID;
        }endif;
```
### APPENDIX A. QVT TRANSFORMS

```java
310       tmp += state.map petrinets(id,swtypeid);
311       id := id+1;
312   }
313   petriNet := tmp;
314 }
315 /**<
316 * Generation d'un Noeud de la NetList a partir de l'id et de l'id du type du noeud
317 */
318 mapping State::petrinets(id : Integer,newtype : Integer) : Nessie::PetriNetType{
319     ID := id;
320     type := newtype;
321 }
322 /**<
323 * Fonction auxiliaire qui verifie si un noeud est initial ou pas.
324 */
325 query Vertex::isinitial() : Boolean{
326     var res : Integer := self.repr().indexOf("kind: ");
327     var ret : Boolean;
328     if(res = -1) then {
329         ret := false;
330     }else{
331         if (self.repr().substringAfter("kind:").substringBefore(" ") = "initial")
332             then {
333                 ret := true;
334             }endif;
335     }
336     return ret;
337 }
338 /**<
339 * Fonction de debug pour afficher un boolean en console
340 */
341 query Boolean::toString() : String{
342     var res : String := "false";
343     if(self) then{
344         res := "true";
345     }endif;
346     return res;
347 }
348 /**<
349 * Calcul de la liste des transitions contenus dans une region
350 */
351 mapping Region::transitionlist() : Nessie::TransitionsListType{
352     // On recupere le noeud initial pour creer la transition de depart ( ...startingtransition)
353     self.allOwnedElements()[Pseudostate]->forEach(state | state.hasinitial()){ 
354         startingTransition := state.map createtransition(0);
355     }
356     //On calcule les autres transitions
357     transition := self.othertransitions();
358 }
359 /**<
360 * On cherche si un Pseudostate (qui represente une transition) a en entree un etat initial.
361 */
362 query Pseudostate::hasinitial() : Boolean{
363     var res : Boolean := false;
364     self.incoming->forEach(income){
365         //Si la source de la transition est un etat initial, on renvoie true
366         if (income.source.isinitial()) then{
367             res := true;
368         }endif;
369 ```
/** Calcul des transitions qui ne sont pas initiales dans une région (icm une variante ...)
 */
query Region::othertransitions() : OrderedSet(Nessie::TransitionType){
  var tmp : OrderedSet(Nessie::TransitionType) := OrderedSet();
  var id : Integer := 1;
  /* L'état initial et les transitions sont vues par QVTO comme des 'Pseudostate'
    Ici on veut parcourir les transitions qui ne sont pas initiales, ie qui ne
    ...partent pas de l'état initial
    - isinitial permet d'exclure l'état initial
    - hasinitial permet d'exclure les transitions qui partent de l'état initial
   */
  self.allOwnedElements()[Pseudostate]->forEach(state | not(state.hasinitial() or
    ...state.isinitial()) ){
    tmp += state.map createtransition(id);
    id := id + 1;
  };
  return tmp;
}

/** A partir d'un Pseudostate (qui représente les transitions dans le diagramme), on
 calcule le TransitionType correspondant, qui contient la liste des états en
 entrée et en sortie de la transition.
 */
mapping Pseudostate::createtransition(id : Integer) : Nessie::TransitionType{
  /* Si on n'est pas dans l'état de départ (celui d'id 0), on cherche les entrées
  de la transition*/
  inputPlaceList := self.map inputplace(id);
  transitionID := id;
  // On calcule la liste des états en sortie de la transition.
  outputPlaceList := self.map outputplace();
}

// Un mapping intermédiaire pour calculer la liste des entrées d'une transition
mapping Pseudostate::inputplace(id : Integer) : Nessie::InputPlaceListType{
  inputPlace := self.inputplacelist(id);
}

// On calcule effectivement la liste des entrées de la transition
query Pseudostate::inputplacelist(id : Integer) : OrderedSet(Nessie::InputPlaceType){
  var tmpin : OrderedSet(Nessie::InputPlaceType) := OrderedSet();
  /* Self. incoming représente sur le graphe tous les états qui entrent sur la
   ...transition courante.
   Pour chacun d'entre eux, on calcule le InputPlaceType correspondant */
  self.incoming->forEach(transin){
    tmpin += transin.map oneinput();
  };
  return tmpin;
APPENDIX A. QVT TRANSFORMS

428 }
429 // On calcule le 'InputPlaceType' qui correspond pour Nessie a un etat en entree d'une 
... transition
430 mapping Transition::oneinput() : Nessie::InputPlaceType{
431 var tmpid := 0;
432 // On recupere dans la liste des noeuds du reseau de petri l'id de la source de la 
... transition
433 self.source.resolveone(PetriNetType)->forOne(resolv){
434 tmpid := resolv.ID;
435 }
436 // Valeur arbitraire, on ne represente pas cette information dans le diagramme
437 placeID := tmpid;
438 numberOfTokens := 1;
439 }
440
441 // Le calcul des etats de sortie est similaire a celui des etats d'entree...
442 mapping Pseudostate::outputplace() : Nessie::OutputPlaceListType{
443 outputPlace := self.outputplacelist();
444 }
445
446 query Pseudostate::outputplacelist() : OrderedSet(Nessie::OutputPlaceType){
447 var tmpout : OrderedSet(Nessie::OutputPlaceType) := OrderedSet{};
448 var isfinal := true;
449 self.outgoing->forEach(transout){
450 if(transout.target.metaClassName() != "FinalState") then{
451 isfinal := false;
452 }endif;
453 }
454 if(not isfinal) then{
455 self.outgoing->forEach(transout | transout.target.metaClassName() != " 
... FinalState"){
456 tmpout += transout.map oneoutput();
457 }
458 }endif;
459 return tmpout;
460 }
461
462 mapping Transition::oneoutput() : Nessie::OutputPlaceType{
463 var tmpid := 0;
464 log(self.target.metaClassName());
465 self.target.resolveone(PetriNetType)->forOne(resolv){
466 tmpid := resolv.ID;
467 }
468 placeID := tmpid;
469 numberOfTokens := 1;
470 }
471
472 /*
...*******************************************************************************/
473 /* GESTION DES DOF
... */
474 /*
...*******************************************************************************/
475 /*
...*******************************************************************************/
476 // On utilise la propriete intermediaire source.dof
477 Lorsqu'on ajoute une variante d'une structure, on verifie si la structure parente 
... est deja 
478 presente auquel cas on ajoute simplement la variante.
479 sinon on cree une nouvelle StructureDOF
APPENDIX A. QVT TRANSFORMS

481 */
482 query add_dof(type : HWSWStringType,abstractlvl : Integer,typeid : Integer,
483 structindex : Integer){
484 var modif : Boolean := false;
485 software.dof->forEach(elt){
486 // Si on a deja ajoute une variante de la meme structure
487 if(elt.abstractionLevel = abstractlvl and elt.ID = typeid and elt.
488 ...typeOfStructure = type ) then{
489 elt.structureChoice += structindex;
490 modif := true;
491 }endif;
492 };
493 // Si on n'avait pas encore de variante pour cette structure
494 if(not modif) then{
495 // On en cree une nouvelle
496 software.dof += object StructureDOFtype{
497 structureChoice := Sequence{structindex};
498 abstractionLevel := abstractlvl;
499 typeofStructure := type;
500 ID := typeid;
501 } endif;
502 }
503 }
504 */
505 /* GESTION DE LA LISTE DES CRITERES */
506 */
507 /***************************************************************************/
508 /* Extration de la liste des criteres a partir du bloc de depart. */
509 /***************************************************************************/
510 query Class::criterialist() : Set(Nessie::CriterionType){
511 var criteres : Set(Nessie::CriterionType);
512 self.allOwnedElements()[Constraint]->forEach(elt){
513 //On extrait la liste des criteres a partir du bloc "comment" qui
514 a pour nom "criteria" dans le bloc "Main"
515 Cette liste est une chaine de caracteres utilisant comme separateur ";"
516 **/
517 if (elt.getLabel() = "criteria") then{
518 var strcriteres := elt.specification.stringValue();
519 strcriteres.split(";")->forEach(critere){
520 criteres += object Nessie::CriterionType{
521 name := criteres;
522 //POST-TRAITEMENT NEEDED !!!!
523 timeDependent := Nessie::TimeDependenceType::none;
combinationRule := Nessie::CombinationRuleType::
    ...noneRule;
}

hardware.crit += critere;
}
}

return criteres;
}

/*/ 
/* Creation d'un element CriterionType a partir du nom du criterre. 
/* Les autres parametres du CriterionType seront fixes en post-traitement 
/* dans l'interface graphique. */

mapping Class::onecriterion(critname : String) : Nessie::CriterionType{
    name := critname;
    timeDependent := null;
    combinationRule := null;
}

/*/ 
/** 
/* GESTION DU HARDWARE 
/* */

(mapping Package::software() : HWdescriptionType{
    HWhierarchy := self.map hwhierarchy();
    HWstructuresList := self.map hwstructures();
}

xmlns := "xmlns";  // facilite le post-traitement
)

(/*/ 
/*
/** 
/* Calcul de la Hierarchy. 
/* Un part du bloc "Main" et on calcule recursivement le contenu des blocs 
/* du diagramme. 
/* Comme pour la partie software, on utilise un dictionnaire 
/* niveau d'abstraction => HWtype 
/* qu'on met a jour, et qu'on parcouru a la fin pour creer les differents 
/* AbstractionLevelType */

(mapping Package::hwhierarchy() : HWhierarchyType{
    var dico : Dict(Integer,OrderedSet(Nessie::HWtype));
    self.allOwnedElements()[Class]->forOne(classe | classe.name = "Main"){ 
        abstractionLevel := classe.abstractions(0,dico);
    };
    var res : OrderedSet(Nessie::AbstractionLevelType);
    dico->keys()->forEach(cle){
        res += xmap createabstract(cle,dico->get(cle));
    };
    abstractionLevel := res;
}

(/*/ 
/*
/** 
/* Creation d'un objet "AbstractionLevelType" representant un niveau d'abstraction, 
/* ...a partir du niveau (lvl) et de la liste des structures de ce niveau (lst) */
mapping createabstract(lvl : Integer,lst : OrderedSet(Nessie::HWtype)):
  ...AbstractionLevelType{
    HWsubTypes := lst;
    abstractionLevelNumber := lvl;
  }

Fonction recursive calculant le contenu des différent niveaux d'abstraction du modèle.
On utilise la variable 'dico' qui est mise à jour au cours des appels récursifs.

helper Class::abstractions(lvl : Integer,inout dico : Dict(Integer,OrderedSet(Nessie::HWtype))){
  var src : Class;
  var tar : Class;
  // Une "Generalization" est "une flèche" reliant un niveau d'abstraction au suivant
  self.allOwnedElements()[Generalization]->forEach(elt){
    // On prend la source de "la flèche"
    elt.source[Class]->forOne(gensrc){
      src := gensrc;
      // On prend la destination de "la flèche"
      elt.target[Class]->forOne(gentar){
        tar := gentar;
        // On appelle récursivement la fonction sur le niveau suivant
        if(src.name = self.name) then{
          tar.abstractions(lvl+1,dico);
        }endif;
    }endfor;
  }endfor;
  // On crée l'abstraction de niveau courant
  self.createabstraction(lvl,dico);
  // Pour chaque sous-block contenu dans le bloc courant, on calcule son HWtype
  self.allOwnedElements()[Class]->forEach(classe){
    classe.abstractions(lvl,dico);
  }endfor;
}

helper Class::createabstraction(lvl : Integer,inout dico : Dict(Integer,OrderedSet(Nessie::HWtype))){
  var id : Integer := getdicoid(lvl,dico);
  self.allOwnedElements()[Class]->forEach(elt){
    if(elt.resolveone(Nessie::HWtype) = null) then{
      addtodico(lvl,elt.map hwsotypes(lvl,id),dico);
    }endi;
  }endi;
}

query getdicoid(lvl : Integer,dico : Dict(Integer,OrderedSet(Nessie::HWtype))) : ...
...Integer{
APPENDIX A. QVT TRANSFORMS

646 var ret : Integer := 0;
647 if(lvlexistindico(lvl,dico)) then{
648 ret := dico->get(lvl)->size();
649 }endif;
650 return ret;
651 }
652
653 helper addtodico(lvl : Integer,type : Nessie::HWtype,inout dico : Dict(Integer,
...OrderedSet(Nessie::HWtype))){
654 // La nouvelle liste a laquelle on va rajouter l'element
655 var newlst : OrderedSet(Nessie::HWtype);
656 //Si le niveau existe
657 if( lvlexistindico(lvl,dico)) then{
658 //On extrait la liste, et on y ajoute l'element
659 newlst := dico->get(lvl);
660 newlst += type;
661 }endif;
662 //On place au niveau 'lvl' la liste mise a jour
663 dico->put(lvl,newlst);
664 }
665 }
666
667 /**
668 On verifie si un niveau d'abstraction donne existe dans le dictionnaire ou pas
669 (ie si on a deja ajoute des elements a ce niveau d'abstraction)
670 */
671 helper lvlexistindico(lvl : Integer,dico : Dict(Integer,OrderedSet(Nessie::HWtype)))
...: Boolean{
672 dico->keys()->forEach(cle){
673 if(cle = lvl ) then {
674 return true;
675 }endif;
676 }
677 return false;
678 }
679 }
680
681 /**
682 Creation d'un HWtype a partir du niveau d'abstraction et de l'id
683 */
684 mapping Class::hwsubtypes(lvl : Integer,id : Integer) : Nessie::HWtype{
685 HWparametersList := self.map hwparamlist();
686 ID := id;
687 transitionalTimeTable := self.map transitionaltimetable();
688 name := self.name;
689 models := self.map hwmodels(lvl,id);
690 }
691
692 /**
693 Ces elements sont traits en post-traitement (GUI)
694 */
695 mapping Class::hwparamlist() : Nessie::HWparametersListType{
696 HWparameter := OrderedSet();
697 }
698
699 /**
700 Ces elements sont traits en post-traitement (GUI)
701 */
702 mapping Class::transitionaltimetable() : Nessie::TransitionalTimeTableType{
703 startState := OrderedSet();
704 }
705
706 /**
707 Creation des differentes "models" (behaivour) d'une structure hardware
708 */
709 mapping Class::hwmodels(lvl : Integer,hwtypeid : Integer) : Nessie::ModelsType{
710 }
APPENDIX A. QVT TRANSFORMS

```plaintext
coreStateList := self.map corestatelist(lvl,hwtypeid);
compatibleSWlist := self.map compatiblelist(lvl,hwtypeid);
IOstateModelsList := self.map iostatelist(lvl,hwtypeid);
}
/**
Creation de la liste des CoreState
*/
mapping Class::corestatelist(lvl : Integer,hwtypeid : Integer) : Nessie::...CoreStateListType{
coreStateModel := corestatemodellst(self,lvl,hwtypeid);
}

/**
Un CoreState peut contenir les behaviour "idle","sleeping","memorizing","...transmitting"
Le nom du fichier XML est genere automatiquement par genfilename.
*/
query corestatemodellst(classe : Class,lvl : Integer,hwtypeid : Integer) : OrderedSet...(Nessie::CoreStateModelType1){
var tmp : OrderedSet(Nessie::CoreStateModelType1);
tmp += classe.map corestatesmodel(CoreStateType::idle,genfilename(lvl,hwtypeid,"...idle"));
tmp += classe.map corestatesmodel(CoreStateType::sleeping,genfilename(lvl,...hwtypeid,"sleeping"));
tmp += classe.map corestatesmodel(CoreStateType::memorizing,genfilename(lvl,...hwtypeid,"memorizing"));
tmp += classe.map corestatesmodel(CoreStateType::transmitting,genfilename(lvl,...hwtypeid,"transmitting"));
return tmp;
}

/**
On genere le champ CoreStateModel.
Ici on place une reference vers xmlfile.
Celui-ci n’est pas genere par cette transformation mais le sera en post-traitement (gui)
*/
mapping Class::corestatesmodel(corestate : CoreStateType,xmlfile : String) : Nessie::...CoreStateModelType1{
behaviour := xmlfile;
coreState := corestate;
SubFileBehaviour := null;
}

/**
Les trois fonctions suivantes fonctionnent comme les trois precedentes
*/
mapping Class::iostatelist(lvl : Integer,hwtypeid : Integer) : Nessie::...IOstateModelsListType{
IOstateModel := iostatemodellist(self,lvl,hwtypeid);
}
query iostatemodellist(classe : Class,lvl : Integer,hwtypeid : Integer) : OrderedSet...(Nessie::IOstateModelType){
var tmp : OrderedSet(Nessie::IOstateModelType);
tmp += classe.map iostatesmodel(IOStateType::inactive,genfilename(lvl,hwtypeid,"...inactive"));
tmp += classe.map iostatesmodel(IOStateType::receiving,genfilename(lvl,hwtypeid,"...receiving"));
tmp += classe.map iostatesmodel(IOStateType::sending,genfilename(lvl,hwtypeid,"...sending"));
return tmp;
}
```

92
mapping Class::iostatesmodel(iostate : IOStateType,xmlfile : String) : Nessie::...
          IOstateModelType{
          behaviour := xmlfile;
          IOstateName := iostate;
          SubFileBehaviour := null;
        }

mapping Class::compatiblelist(lvl : Integer,hwtypeid : Integer) : Nessie::...
          CompatibleSWlistType{
        computingModel := self.composableswlist(lvl,hwtypeid);
      }

query Class::compatibleswlist(lvl : Integer,hwtypeid : Integer) : OrderedSet(Nessie::...
          ComputingModelType2){
      var tmp : OrderedSet(Nessie::ComputingModelType2);
      return tmp;
    }

/**
   * Generation des noms des sous-fichiers, de la forme
   * HW_i_j_type.xml avec
   *  - i : le niveau d'abstraction
   *  - j : l'ID du block hardware
   *  - type : le type du behaviour (idle, sleeping, memorizing, transmitting, ...
inactive, receiving, sending,)
   */

query genfilename(abstractionlvl : Integer,hwsubtypeid : Integer,type : String) :
          String{
      var res : String;
      if(hwsubtypeid != -1) then{
        res := "HW_"+abstractionlvl.toString() + "_" + hwsubtypeid.toString() +"_"+
          type".xml";
      }else{
        res := "HW_"+abstractionlvl.toString() + "_"+type".xml";
      }endif;
      return res;
    }

/**
** Calcul de la structure du block, avec ses différents liens.
On extrait le block "Main" et on effectue les calculs a partir de lui.
**
mapping Package::hwstructures() : Nessie::HWstructuresListType{
      var ismain := false;
      var tmp : Dict(ConnectorEnd,Port);
      tmp := self.dicoPorts();
      self.allOwnedElements()[Class]->forOne(classe | classe.name = "Main"){
        ismain := true;
        HWstructure := classe.hwstructure(0,0,0,tmp,true);
      };
      if (not ismain) then{
        log("ERREUR/uni2423:/uni2423Aucun/uni2423bloc/uni2423'Main'/uni2423n'a/uni2423ete/uni2423trouve/uni2423dans/uni2423l'architecture/uni2423materielle
          ...d'entree");
      }endif;
    }

/**
** Fonction auxiliaire initialisant un dictionnaire (ie une table d'association).
Pour chaque Port, on enregistre a quel "ConnectorEnd" il est associe.
**
Un "Connector" est le lien entre deux blocs, un "Port" est ce qui est place sur le bloc, et auquel est connecte le "Connector".

Le "ConnectorEnd" est le "bout" du "Connector" qui est place sur le bloc.

```
package::dicoPorts() : Dict(ConnectorEnd,Port){
  var tmp : Dict(ConnectorEnd,Port);
  self.allOwnedElements()[Port]->forEach(port){
    port._end[ConnectorEnd]->forEach(ending){
      tmp->put(ending,port);
    }
  }
  return tmp;
}
```

```
class::hwstructure(lvl : Integer,hwid : Integer,index :Integer,porttoclass : ...
  ...Dict(ConnectorEnd,Port),addblock : Boolean) : OrderedSet(Nessie::...
  ...HWstructureType){
  var tmp : OrderedSet(Nessie::HWstructureType);
  var indexrec : Integer := 0;
  var hwtype : Integer := -1;
  // Pour chaque classe constituant le block courant, on recupere l'id de la classe ...
  // (calcule au cours de la generation de la hwhierarchy).
  // On calcule alors l'hwstructure de ce block
  self.allOwnedElements()[Class]->forEach(subclasse){
    hwtype := subclasse.resolveone(HWtype).ID;
    tmp += subclasse.hwstructure(lvl,hwtype,index,porttoclass,false);
  }
  // Generation effective de la structurelist (cas terminal de cette fonction ...
  // recursive)
  if(addblock) then{
    tmp += self.map onehwstructure(lvl,hwid,index,porttoclass);
  }endif;
  var src : Class;
  var tar : Class;
  // Pour chaque element de niveau d'abstraction superieur, ie chaque variante ...
  // specifiant le comportement de ce block
  self.allOwnedElements()[Generalization]->forEach(elt){
    elt.source[Class]->forOne(gensrc){
      src := gensrc;
    }
    elt.target[Class]->forOne(gentar){
      tar := gentar;
    }
    // On calcule l'hwstructure
    if(src.name = self.name) then{
      tmp += tar.hwstructure(lvl+1,src.resolveone(HWtype).ID,indexrec,
        ...porttoclass,true);
    }
    indexrec := indexrec + 1;
  }
  return tmp;
}
```

```
// Cas terminal de la fonction precedente.
```
Generer l'objet HWstructureType a partir des elements le caracterisant (niveau d'
...abstraction, hwtype, index).

/*
mapping Class::onehwstructure(lvl : Integer,hwtype : Integer,index : Integer,
...porttoclass : Dict(ConnectorEnd,Port)) : Nessie::HWstructureType{
abstractionLevel := lvl;
HWtypeID := hwtype;
structureIndex := index;
localizedElementsHWstructure := object Nessie::LocalizedElementsHWstructureType{
HWblockList := self.map hwblocklist();
linkList := self.map hlinklist(porttoclass);
};
/**
Calcul de la liste des sous-blocks d'un block
*/
mapping Class::hwblocklist() : Nessie::HWblockListType{
HWblock := self.calcblocks();
}
/**
Calcul effectif de la liste des sous-blocks
*/
query Class::calcblocks() : OrderedSet(HWblockType){
var tmp : OrderedSet(HWblockType); 
var id : Integer := 0;
var type : Integer;
// Pour chaque sous-block, on recupere son type et on genere le block avec '...
oneblock'
self.allOwnedElements()[Class]->forEach(block){
type := block.resolveone(HWtype).ID;
tmp += block.map oneblock(id,type);
id := id + 1;
};
return tmp;
/**
Generation d'un bloc
*/
mapping Class::oneblock(id : Integer,hwtype : Integer) : HWblockType{
ID := id;
type := hwtype;
}
/**
Calcul de la liste des liens dans un block
*/
mapping Class::hlinklist(porttoclass : Dict(ConnectorEnd,Port)) : Nessie::
...LinkListType{
link := self.alllinks(porttoclass);
}
/**
Calcul effectif de la liste des liens
*/
query Class::alllinks(porttoclass : Dict(ConnectorEnd,Port)) : OrderedSet(Nessie::
...LinkType){
var tmp : OrderedSet(Nessie::LinkType);
// pour chaque lien, on appelle 'createlink' pour creer le lien
self.allOwnedElements()[Connector]->forEach(elt){
tmp += elt.map createlink(porttoclass);
};
return tmp;
On part d'un lien (Connector) entre deux blocs.
On genère un LinkType correspondant, constitué des éléments :
- source
- destination
- bidirectionnel

On prend chaque bout du connecteur, et on cherche à quoi il est relié.

Un block peut être in, out, inout
Suivant les différentes combinaisons sur les 2 ports reliés, on calcule la ...
... source, la destination et si le lien est bidirectionnel.
Tableau récapitulatif :

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>in</th>
<th>out</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>cas1=&quot;ERREUR&quot;</td>
<td>cas2=&quot;source:p2,dest=p1&quot;</td>
<td>cas3=&quot;source:p2,dest=p1&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>cas4=&quot;cas2 inverse&quot;</td>
<td>cas5=&quot;ERREUR&quot;</td>
<td>cas6=&quot;cas3 inverse=&quot;source:p1,dest=p2&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inout</td>
<td>cas7=&quot;cas3 inverse&quot;</td>
<td>cas8=&quot;source:p2,dest=p1&quot;</td>
<td>cas9=&quot;source:p2,dest=p1&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... les deux sens sont équivalents, seul cas bidirectionnel

*/

var cas1 : Boolean := (dir1.equalsIgnoreCase("in") and dir2.equalsIgnoreCase("in" ...));
var cas2 : Boolean := (dir1.equalsIgnoreCase("in") and dir2.equalsIgnoreCase("out" ...));
var cas3 : Boolean := (dir1.equalsIgnoreCase("in") and dir2.equalsIgnoreCase("out inout"));
var cas5 : Boolean := (dir1.equalsIgnoreCase("out") and dir2.equalsIgnoreCase("...
...out"));
var cas6 : Boolean := (dir1.equalsIgnoreCase("out") and dir2.equalsIgnoreCase("...
inout"));
var cas7 : Boolean := (dir1.equalsIgnoreCase("inout") and dir2.equalsIgnoreCase("...
in"));
var cas8 : Boolean := (dir1.equalsIgnoreCase("inout") and dir2.equalsIgnoreCase("...
out"));
var cas9 : Boolean := (dir1.equalsIgnoreCase("inout") and dir2.equalsIgnoreCase("...
inout"));

if(cas2 or cas3 or cas8) then{
  idsource := id2;
  idsink := id1;
} else {
  if(cas4 or cas6 or cas7 or cas9) then {
    idsource := id1;
    idsink := id2;
  } else {
    if(cas1 or cas5) then {
      log("L'architecture materielle est invalide dans les liens");
    } endif
  }
} endif

bidirectional := cas9;
source := idsource;
sink := idsink;