



**I  
N  
A  
O  
E**

**Model-Based Design of Digital Hardware Systems  
for Digital Communications**

by

**Tomás Balderas Contreras**  
M.Sc., INAOE

A thesis submitted in partial fulfillment of the  
requirement for the degree of

**DOCTOR OF SCIENCE  
IN COMPUTER SCIENCE**

at

**Instituto Nacional de Astrofísica, Óptica y  
Electrónica**

November 2012

Tonantzintla, Puebla. MEXICO

Supervised by

**Dr. René Armando Cumplido Parra**  
Professor at INAOE

and

**Dr. Gustavo Rodríguez Gómez**  
Professor at INAOE

©INAOE 2012

All rights reserved

The author grants INAOE the right to reproduce and  
distribute full or partial copies of this dissertation





Model-Based Design of Digital Hardware Systems  
for Digital Communications

**By**

Tomás Balderas Contreras

**Supervisors**

Dr. René A. Cumplido Parra and Dr. Gustavo Rodríguez Gómez

Computer Science Department  
Instituto Nacional de Astrofísica, Óptica y Electrónica  
Tonantzintla, Puebla. MEXICO

2012



## **Abstract**

The design complexity of digital hardware systems has grown at the same pace as the scale of integration of integrated circuits, and the need for functionality. This complexity gave rise to different design approaches at different levels of abstraction through time, being the current approach that proposing the description of the functionality of digital hardware systems using languages and tools similar to the ones employed to build software systems. Thus, it is possible to use the latest advances in software engineering to describe the functionality of modern digital circuits. This dissertation describes a design flow consisting of a dialect of UML 2 adapted to describe different algorithms operating on blocks of bits; and a transformation technology that generates structural VHDL code from the high-level models. The technology described in this document exploits the principles of object orientation, model-driven engineering, and transformation from models to text. The primary aim is to alleviate complexity during the design of digital hardware systems implementing demanding operations used by a wide range of computing devices. The generation of descriptions in VHDL from models describing algorithms used by the data processing stages of modern digital communication systems illustrates the usefulness of the proposed technology.

## Resumen

La complejidad en el diseño de los sistemas de hardware digital ha crecido al mismo ritmo que la escala de integración de los circuitos integrados y la demanda de funcionalidad. Esta complejidad motivó el desarrollo de diversos enfoques de diseño a distintos niveles de abstracción, siendo el más reciente el que propone la descripción funcional de los sistemas de hardware digital mediante lenguajes y herramientas similares a las empleadas para desarrollar sistemas de software. Por lo tanto, es posible aplicar los avances más recientes de la ingeniería de software para describir la funcionalidad de los circuitos digitales actuales. La presente disertación describe un flujo de diseño consistente en un dialecto de UML 2 que permite modelar algoritmos que operan con bloques de bits y una herramienta de transformación que genera código estructural en VHDL a partir de dichos modelos de alto nivel. La tecnología expuesta en este documento explota los principios de orientación a objetos, ingeniería dirigida por modelos y transformaciones modelo a texto. El objetivo principal es paliar la complejidad del proceso de diseño de sistemas de hardware digital que implementan operaciones demandantes. La utilidad de la tecnología propuesta se demuestra mediante la generación de descripciones en VHDL a partir de modelos de algoritmos empleados durante las etapas de procesamiento de información en un sistema moderno de comunicación digital.

## **Acknowledgments**

I am indebted with my advisors, Dr. René Cumplido and Dr. Gustavo Rodríguez, for having provided me with guidance, support and encouragement from the beginning of this project up to its culmination.

I am grateful to the reviewers of this dissertation, Dr. Leopoldo Altamirano, Dr. Miguel Arias, Dr. Claudia Feregrino, Dr. María del Pilar Gómez, and Dr. Luis Fernando González, for having read this dissertation, provided valuable feedback, and evaluated every stage of the project.

My recognition extends to every single member of the community working at INAOE, who supported me during my graduate studies; without this assistance, I could not have earned my PhD degree.

My appreciation goes to Marisol Flores, a wonderful person who helped me with the paperwork required by INAOE.

Finally, I would like to thank CONACyT, the Mexican council for science and technology, for financing my graduate studies through the scholarship 41722.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Design complexity . . . . .	2
1.2	Productivity gap . . . . .	4
1.3	Levels of abstraction . . . . .	5
1.4	The proposal and its purpose . . . . .	7
1.5	Contributions . . . . .	8
1.6	Outline . . . . .	9
<b>2</b>	<b>Works related to UML as an ESL language</b>	<b>11</b>
2.1	Review of ESL . . . . .	11
2.2	Languages based on C . . . . .	12
2.3	Functional descriptions using UML . . . . .	15
2.3.1	Modeling systems-on-a-chip using UML . . . . .	15
2.3.2	Mapping from UML to VHDL . . . . .	18
2.4	Discussion . . . . .	20
<b>3</b>	<b>Proposed framework</b>	<b>23</b>
3.1	The paradigm of model-driven engineering . . . . .	23
3.2	Model-driven architecture . . . . .	25
3.3	Description . . . . .	27
3.4	Defining and bounding the application domain . . . . .	29
3.4.1	Digital communication systems . . . . .	29



<b>4</b>	<b>The domain-specific modeling language</b>	<b>32</b>
4.1	Overview of meta-modeling . . . . .	33
4.2	Introduction to the meta-model of UML 2 . . . . .	35
4.2.1	The meta-levels . . . . .	35
4.3	Definition of profiles in UML 2 . . . . .	37
4.4	Definition of the domain-specific modeling language . . . . .	39
4.4.1	Adapting activity diagrams to model flows of bit-blocks . . . . .	40
4.4.2	The organization of the modeling language . . . . .	44
4.4.3	Modules and their interfaces . . . . .	46
4.4.4	Operations and their operands . . . . .	50
4.4.5	Dataflows . . . . .	54
4.4.6	Invoking modules and switching between bit-blocks . . . . .	54
4.5	Application of the profile . . . . .	57
4.5.1	Block ciphering . . . . .	58
4.5.2	Multiplication in finite fields . . . . .	65
<b>5</b>	<b>The code generator</b>	<b>77</b>
5.1	Overview of MOFM2T and Acceleo . . . . .	77
5.2	The simplified grammar of VHDL . . . . .	82
5.3	The transformation to VHDL . . . . .	83
5.3.1	Relevant templates and queries . . . . .	84
5.3.1.1	Literals and signals . . . . .	84
5.3.1.2	Reduction of labels of dataflows . . . . .	85
5.3.1.3	Recursive generation of expressions . . . . .	86
5.3.2	Generating the entity declaration of the design . . . . .	87
5.3.3	Generating the architecture body of the design . . . . .	89
5.4	Results of simulation . . . . .	92
5.5	Discussion . . . . .	97
<b>6</b>	<b>Processes, methods and metrics</b>	<b>100</b>
6.1	It is all about quality . . . . .	100
6.2	Related work . . . . .	102

6.3	Proposal for a software process and metrics . . . . .	103
6.3.1	Principles of the proposal . . . . .	103
6.3.2	Introduction to the Personal Software Process . . . . .	104
6.4	Evaluation of productivity . . . . .	106
6.4.1	Measures of productivity . . . . .	106
6.4.2	Proposed methodology to measure productivity . . . . .	108
6.5	Discussion . . . . .	109
<b>7</b>	<b>Conclusions</b>	<b>111</b>
7.1	Concluding remarks . . . . .	111
7.2	Future work . . . . .	113
7.3	Contributions . . . . .	114
7.4	Publications . . . . .	114
<b>A</b>	<b>Review of the notation of UML 2</b>	<b>123</b>
A.1	Class diagrams . . . . .	123
A.1.1	Classes and relationships . . . . .	124
A.1.2	Derived unions . . . . .	126
A.1.3	Keywords . . . . .	127
A.2	Activity diagrams . . . . .	129
A.2.1	Activities, nodes and edges . . . . .	129
A.2.2	Actions . . . . .	131
A.3	Object diagrams . . . . .	131
<b>B</b>	<b>The profile BitBlockFlow</b>	<b>133</b>
<b>C</b>	<b>Modified version of the grammar of VHDL</b>	<b>148</b>
<b>D</b>	<b>List of acronyms</b>	<b>153</b>

# List of Figures

1.1	Exponential increase in the number of transistors per chip against the slower increase in productivity along time . . . . .	4
2.1	A design framework based on UML to design SoC platforms . . . . .	17
2.2	MODCO's transformation from UML to VHDL . . . . .	20
3.1	A development framework based on MDA: transformation of PIMs into PSMs and code generation . . . . .	25
3.2	Proposed framework based on MDA . . . . .	28
3.3	Block diagram of a typical digital communications system . . . . .	30
4.1	Domain model illustrating the organization of an airline company . . . . .	33
4.2	A model including instances of sub-classes of <b>Employee</b> and their links to an instance of <b>Airplane</b> . . . . .	34
4.3	An example of the notation used in UML 2 to denote that a number of classes are instances of a specific meta-class . . . . .	35
4.4	The meta-levels in the framework that defines UML 2 . . . . .	36
4.5	Definition of a profile that extends a M2 meta-model and its application to extend a M1 model . . . . .	38
4.6	Extracts of the meta-model of UML 2 defining some modeling elements in an activity diagram . . . . .	41
4.7	The packages in the meta-model of UML 2 extended by the profile . . . . .	45
4.8	The internal structure of the profile BitBlockFlow . . . . .	46
4.9	The stereotypes in the package ModuleInterface and the meta-classes extended by them . . . . .	47

4.10	A module, its parameters and some operations . . . . .	47
4.11	The stereotypes in the package Operations and the meta-classes they extend . . . . .	50
4.12	Different kinds of operations and their input and output operands . .	51
4.13	An extended state machine diagram in UML 2 associated to a switch	55
4.14	A module containing invocations to other modules . . . . .	57
4.15	The components of the block cipher KASUMI . . . . .	59
4.16	The modules comprising the model of KASUMI in UML 2 and the profile BitBlockFlow . . . . .	60
4.17	The components of the simplified block cipher KASUMI . . . . .	65
4.18	The modules comprising the model of the simplified KASUMI in UML 2 and the profile BitBlockFlow . . . . .	67
4.19	Modules performing multiplication of integers of different lengths ac- cording to the Karatsuma-Ofman algorithm . . . . .	71
5.1	Syntax diagrams for the declaration of a class in the Java language .	79
5.2	An operation whose operands are the output of an operation and an integer literal. . . . .	85
5.3	Reduction of labels of dataflows. . . . .	86
5.4	Results of the simulation of the description in VHDL implementing KASUMI . . . . .	93
5.5	Results of the simulation of the description in VHDL implementing the simplified KASUMI . . . . .	94
5.6	Results of the simulation of the description in VHDL implementing KOA and LFSR . . . . .	98
6.1	The layers of a software engineering task . . . . .	101
6.2	Process flow for Personal Software Process . . . . .	105
A.1	A class diagram in UML 2 and its components . . . . .	125
A.2	A class diagram illustrating the concept of derived union . . . . .	127
A.3	The definition of an interface in UML 2 requires modifying a class with a keyword . . . . .	128

A.4 An activity diagram in UML 2 and its components . . . . .	129
A.5 An object diagram in UML 2 and its components . . . . .	132

# List of Tables

2.1	Languages for functional description of digital hardware systems . . .	13
2.2	Libraries of hardware components used to build systems . . . . .	13
2.3	Technologies to map functional descriptions to hardware platforms . .	14

# Chapter 1

## Introduction

A computer-based system is a combination of hardware and software that implements a set of algorithms to automate the solution to a number of problems. Computer design technology transforms the designer's ideas and objectives into a number of representations describing software modules and hardware components, which must be tested and manufactured [14]. The design process is not straightforward; the developers always deal with the problem of alleviating the *complexity of their designs* to develop high-quality products within rigid time constraints. This situation arose as a direct consequence of the steady evolution of technology and the constant demand for new functionality.

This document focuses on the challenging process of designing digital hardware systems and proposes a new method to improve productivity during the phase of functional description. These functional descriptions can be tested and implemented in semiconductor platforms like Application-Specific Integrated Circuits (ASIC) or Field Programmable Gate Arrays (FPGA). In this dissertation, a *digital hardware system* is a set of digital circuits that accelerate the execution of algorithms solving problems belonging to an application domain.

## 1.1 Design complexity

Computer-based systems are not becoming easier to design as time passes; on the contrary, the need to meet new usage demands and the advancement of development and manufacturing technologies encourage the development of devices incorporating more and more functionality. This is a list of some key functional aspects that have been addressed by hardware/software engineers during the last years:

**Communication.** A large number of computing devices shall be connected to the Internet nowadays. The options to implement this connection include a broadband wired Ethernet link, a local wireless WiFi link, a global wireless WiMax link, or a link to a cellular telephone network. It is common for a single device to support more than one of the previous standards to increase its flexibility, at the expense of being more challenging to design efficiently.

**Security.** Several computer systems shall implement mechanisms to cipher information, authenticate users, guarantee the integrity and confidentiality of data, and protect against a number of attacks. These systems usually contain hardware accelerators to increase the performance of the encryption algorithms.

**Power management.** Modern computers shall execute operating systems able to switch the operation mode of an idle hardware component to an operation mode that consumes low power, according to the current workload. The hardware components shall implement a set of power states, each corresponding to a specific requirement of power consumption.

**Multimedia processing.** A wide range of computing devices shall execute software to visualize video streams and files, produce high definition sound, process 2D images, and render 3D images. These devices include different kinds of mobile devices; every video game console; and some desktop computers, workstations, and servers. In many cases, there exist hardware accelerators that increase the performance of the most demanding algorithms.

**Fault tolerance.** Contemporary high performance servers and supercomputers used in mission critical applications shall implement algorithms to detect errors dur-



ing their operation. If the errors cannot be corrected, then the hardware and software must prevent them from spreading and compromising the whole system. These systems also implement algorithms to provide information redundancy and protect sensitive information as much as possible.

When designing the digital hardware of a computer-based system, the developers face the challenge of implementing some of the previous functional requirements while meeting a number of design constraints. The following is a list of the most common restrictions in hardware engineering:

**Higher performance.** Quite often it is not enough to solve a problem; its solution shall be fast. The functionality of a system must be implemented using algorithms with a high degree of performance. Performance is measured in different ways depending on the nature of the problem.

**Power consumption efficiency.** Portable devices must meet their operational requirements while providing long battery life. In this case, the systems must be designed with the goal of consuming power as efficiently as possible.

**Low area.** When there are not enough hardware resources available, the developer must conceive small designs that use iteratively a hardware component until the main operation is complete.

It is not possible to optimize all of the previous parameters at the same time because some of them contradict to each other; therefore, the designer must make trade-offs. To illustrate this situation consider two implementations of a block cipher algorithm for 3G cellular networks proposed by Balderas [8]. The first is an area-efficient implementation encrypting information at a rate of 317.8 Mbps (mega-bits per second); the second is a high-performance implementation requiring 6.3 times more hardware resources with a performance of 5.32 Gbps (giga-bits per second).

It is not possible to stop the evolution of technology over time or prevent computer-based systems from implementing more and more functionality and becoming more complex. Hardware and software engineers will always face the challenge of designing products that implement lots of functionality and meet contradicting constraints in shorter periods of time.

## 1.2 Productivity gap

Current design complexity imposes serious limits to the ability of designers to develop high quality products that fully meet their requirements in a short time, that is, to their *productivity*. This situation occurs in spite of having millions of transistors available for designing digital hardware systems<sup>1</sup>. Figure 1.1 shows the exponential increase in the number of transistors per silicon chip along the last 30 years (continuous line) and the increase in design productivity along the same period (dashed lines). The considerable separation between the lines is called the *productivity gap*, which is defined as the challenge that arises when the number of available transistors grows faster than the ability to design meaningfully with them [20, 14].

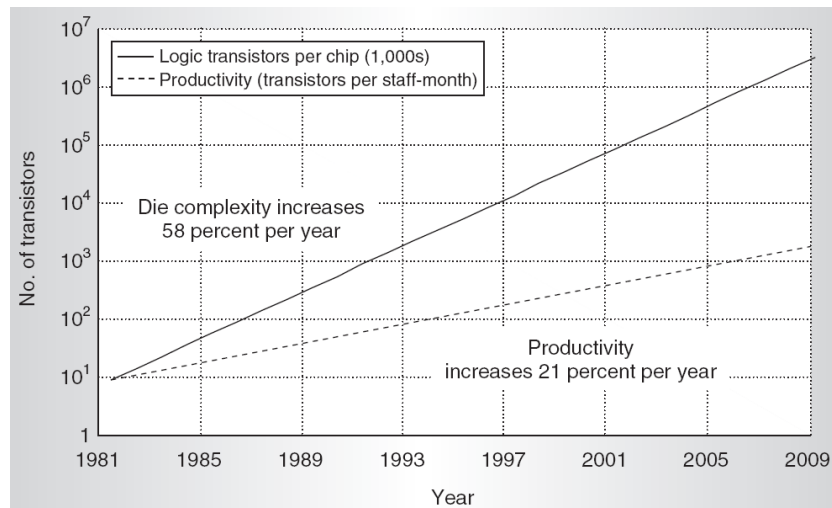


Figure 1.1: Exponential increase in the number of transistors per chip against the slower increase in productivity along time (from [20]).

Developing mechanisms to reduce the productivity gap decreases the chance of introducing errors in the development process of a system. Marketing computer products that were not correctly designed and extensively validated causes economical losses for the manufacturer, which are induced by recalls of buggy products, correc-

<sup>1</sup>The prediction commonly known as “Moore’s Law”, from 1965, states that the number of transistors on a silicon chip doubles every two years, which gives an indication of the current degree of miniaturization and scale integration in a single semiconductor die.

tion of errors in the design, and production and deployment of healthy units. This occurred with the Intel Pentium processor, whose floating-point unit's FDIV instruction produced unexpected results under certain conditions [53]. Another consequence is the total loss of critical artifacts that rely on the functioning of a faulty computer-based system. The self-destruction of the European Space Agency's Ariane 5 rocket, flight 501, in 1996 is an example of this event [34].

### 1.3 Levels of abstraction

Booch et al. stated that “abstraction is one of the fundamental ways that we as humans cope with complexity” [12]. An effective way to alleviate design complexity and reduce the productivity gap is to raise the *level of abstraction* at which developers carry out their activities. The goal is to design correct systems fast by making easy for designers to check for, identify, and fix errors. The raise in the level of abstraction has been performed many times in the past for both software and hardware development.

The following is a brief description of the different levels of abstractions that have been conceived to aid in the design of digital hardware systems throughout the last decades:

**Transistor-level design.** The first solid-state computers were built using discrete transistors and other electronic components. These machines were relatively complex systems with little memory that consumed several kilowatts of power. Their hardware became more complex as new architectural techniques to increase performance were conceived, which made the design with discrete components impractical.

**Schematic design.** When Medium-Scale Integration (MSI) and Large-Scale Integration (LSI) integrated circuits became ubiquitous, the discrete components that made up a whole computer module were gathered together and encapsulated into a single silicon die. This allowed a high degree of miniaturization and the description of hardware components as a set of schematics specifying the interconnection of a number of integrated circuits.

**Register-Transfer Level (RTL) design.** The behavior of a digital circuit is defined in terms of a flow of signals (data transference) between hardware registers and the logical operations performed on those signals. This level of abstraction employs hardware description languages (HDLs), like VHDL and Verilog, to create a more manageable description of a system. This representation can be simulated, validated, and transformed into a description of the electronic components that make up the system and the interconnections between them (net-list), which can be implemented in a Very Large Scale Integration (VLSI) silicon platform.

**Electronic System Level (ESL) design.** The functionality of a digital hardware system is described by means of high level languages built from existing programming languages (like C and Java) and/or graphical tools. The main goals are achieving a high degree of comprehension and reutilization of the functional descriptions, and automating the implementation process [35].

In spite of their advantages to describe the functionality of digital hardware systems at higher levels of abstraction, some ESL technologies have serious drawbacks that prevent them from being used to design some systems, like low-power embedded hardware, efficiently. There is a strong need for high-level design languages and tools customized for different application domains that help alleviate design complexity. ESL is a recent research trend that has been neither fully explored nor fully standardized, and there is still room for significant innovations [17].

At the ESL level, there are lots of similarities between the process of *functional description* of digital hardware systems and the process of software development<sup>2</sup>. Thus, we can think of taking advantage of the recent advances in software engineering to raise the level of abstraction even further, alleviate design complexity, increase reuse of existing designs, and automate the production of representations at lower levels of abstraction.

---

<sup>2</sup>The divergence point is at the moment of implementing the functional description. For software, the description is synthesized as machine code that is executed by a microprocessor; for hardware, the description is synthesized as a set of interconnected logic elements that are placed and routed in a silicon die.

## 1.4 The proposal and its purpose

Model-Driven Engineering (MDE) is a recent paradigm intended to raise the level of abstraction in the design of software systems even further [31]. The first goal of this approach is to conceive solutions (models) in terms of concepts in the problem domain, those that the designers and/or customers know well, instead of concepts in the solution domain, those related to specific hardware/software technologies. A second goal is to alleviate the complexity of current hardware and software platforms by automatically transforming the models into an appropriate implementation using such platforms' technologies. The ultimate goal is to improve both short-term productivity (increased functionality) and long-term productivity (greater longevity) during the development process [5].

This research shows that it is possible to apply the principles of MDE to the design of digital hardware systems. The questions to answer by this project are:

1. What modeling languages better describe the functionality of digital hardware systems using a model-based paradigm?
2. What are the algorithms needed to transform the functional descriptions expressed as high-level models to representations at a level of abstraction closer to the implementation platform?
3. What is the best way to implement the transformation algorithms designed when answering the previous research question?
4. How does the proposed design flow alleviate design complexity?

The general objective of the research documented in this dissertation is to take advantage of the virtues of the MDE paradigm to conceive a *model-driven design flow* (framework) that allows digital hardware designers to perform their duties in a more productive manner. The term design flow refers to an explicit combination of Electronic Design Automation (EDA) tools to accomplish the design of an integrated circuit [33]. The main principles of the proposed framework are: the functional description of systems at a higher level of abstraction, the reuse of the designs, and

the automation of the implementation process. These systems intend to accelerate algorithms used by digital communication devices.

The specific objectives of this project provide more detail on the steps followed to accomplish the previous general objective, and specify the components of the framework. These objectives are as follows:

1. The definition of custom modeling languages that provide the user with primitive constructs representing concepts and abstractions from the domain of digital communications. These Domain-Specific Modeling Languages (DSMLs) are constructed by extending an existing modeling language with new modeling elements representing abstractions in the application domain. The intention of these DSMLs is to allow the designer to conceive solutions to a number of problems within the problem domain by using such domain's jargon. The base modeling language of choice for this project is the Unified Modeling Language version 2 (UML 2) due to its extensibility and widespread use [47, 48].
2. The design of algorithms that synthesize an RTL description of a digital hardware system from the higher level models expressed in DSML. The resulting code in VHDL can be simulated, tested, and implemented in a FPGA platform by using the commercial design flows available in the EDA market.
3. The proposal of a methodology that evaluates the impact of the design flow on the productivity of the designers, accompanied by a suggestion of what existing software development process is the most appropriate to be extended to use the design flow. The execution of this methodology to collect actual results is beyond the scope of this dissertation, mainly because it requires a large team of designers and the application of various tests throughout long periods of time to obtain accurate conclusions.

## 1.5 Contributions

Many worldwide operations currently rely on the correct functioning of computer-based systems. It is extremely important to conceive technologies that allow devel-

opers to design efficient products in a timely manner and minimize the number of errors in such devices. The project documented in this dissertation will contribute to advance the state of the art of technology and improve people's quality of life. The expected contributions to the body of knowledge of EDA are the following:

1. A model-driven design flow to develop high-level functional descriptions of a digital hardware system by means of DSMLs and the automatic transformation of these descriptions to a lower level representation.
2. One modeling language, in the form of a domain-specific dialect of UML 2, to describe the algorithms that perform encryption or compression operations on bit-blocks.
3. A tool that implements the algorithms that transform domain-specific models into an implementation in VHDL. The description of the transformation algorithms is part of the design flow's infrastructure and one of the main deliverables of this project, along with the specification of the modeling language.

## 1.6 Outline

The following chapters of the dissertation describe the main components of the design flow and the key principles that ruled its conception. The remainder of the document is organized as follows:

- Chapter 2 provides further details on ESL and some technologies classified at this level of abstraction. It also reviews the existing projects that use UML to describe the functionality of digital hardware systems. Finally, it analyzes the strengths and weaknesses of each proposal.
- Chapter 3 provides an overview of the proposed design flow and the domain to which it is applied (digital communications). It describes the basic principles behind the design flow and its components: the modeling language used to build the high-level functional descriptions and the algorithms that generate VHDL code.

- Chapter 4 provides a thorough description of the construction of the DSML used to build high-level models. It also documents the features of the language and its application. This chapter introduces the reader to the concept of meta-modeling and the procedure used to derive custom dialects from UML 2 (profiling).
- Chapter 5 provides a description of the algorithms that transform high-level models into VHDL code. This chapter describes the specification that rules the process of transforming models into text and the technologies used to implement such specification.
- Chapter 6 discusses an appropriate method to evaluate the impact of the design flow on productivity. This chapter takes into account the body of knowledge on software engineering processes and metrics.
- Chapter 7 summarizes the results and contributions of this project and describes future work.

The following appendixes provide additional and complementary information to achieve a better comprehension of the work reported in this thesis:

- Appendix A summarizes the notation of UML 2. The reader is encouraged to examine the information in this appendix to become familiar with the language used throughout this dissertation.
- Appendix B provides a thorough documentation of the domain-specific modeling language proposed in this dissertation.
- Appendix C describes the grammar of the language generated by the transformation algorithms, which is a sub-set of VHDL.



# Chapter 2

## Works related to UML as an ESL language

The design flow proposed in this dissertation is an ESL technology providing a level of abstraction that allows the developer concentrate on the features of the algorithms to design, instead of the implementation platform. Thus, the proposed framework allows a better comprehension of the solution under design, which increases the designer's chances of achieving a correct product. This chapter describes the characteristics of ESL, as well as the technologies encompassed by this level related to the proposed framework.

### 2.1 Review of ESL

Bailey et al. define ESL as: “the utilization of appropriate abstractions in order to increase comprehension about a system and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints” [35]. The primary concern is to leverage the use of functional descriptions at higher levels of abstraction to mask not only the implementation details, but also the programming mechanisms. The second concern is reuse, which refers to both architectures and design environments [59].

Even though ESL is a recent approach, there already exist several technologies,

both commercial and academic, at this level of abstraction. According to Densmore et al., these technologies and tools can be classified into three different categories. The first category, referred to as Bin F, includes languages that allow designers to express what the system should do. The second category, referred to as Bin P, includes libraries of modules and hardware components that can be used to implement the functionality. The third category, referred to as Bin M, includes software tools to map the functionality described using the languages to platforms built from the elements in the libraries [17]. Tables 2.1, 2.2 and 2.3 illustrate some of these academic and industrial technologies.

In this dissertation, we are only interested in evaluating the virtues and weaknesses of the technologies employed to describe the functionality of digital hardware systems, those classified as Bin F. The proposed synthesis technology is not directly comparable to the existing technologies in Bin M, listed in Table 2.3, because the source representations are different. Finally, since we are not proposing a new hardware platform, we do not discuss the existing component libraries listed in Table 2.2. The next sections describe some technologies for functional description at higher levels of abstractions; they provide a discussion of the strengths and disadvantages of each technology.

## 2.2 Languages based on C

Edwards [19] describes the two main reasons that motivate the development of variants of the C language to describe the functionality of digital hardware systems:

**Familiarity.** Were it possible to synthesize hardware from C code, there would be no need to invest time and money in learning a new language.

**Hardware/Software co-design.** Sometimes, it is convenient to partition the implementation of a solution to a problem into a hardware unit that accelerates demanding operations and software that implements the rest of the functionality. Using a single language to describe both hardware and software would simplify the designer's work.

Table 2.1: Languages for functional description of digital hardware systems (from [17]).

Bin F	
Provider	Description
MathWorks	High-level technical computing language and interactive environment for algorithm development, data visualization, analysis, and numeric computation
Maplesoft	Mathematical problem development and solving
Wolfram Research	Graphical mathematical development and problem solving with support for Java, C, and .NET
National Instruments	Test, measurement, and control application development
Celoxica	Compiling programs into hardware images of FPGAs or ASICs
University of California, Irvine	ANSI-C with explicit support for behavioral and structural hierarchy, concurrency, state transitions, timing, and exception handling

Table 2.2: Libraries of hardware components used to build systems (from [17]).

Bin P	
Provider	Description
Prosilog	Standard-based IP libraries and support tools (SystemC)
Altera	FPGAs, CPLDs, and structured ASICs
Xilinx	FPGAs, CPLDs, and structured ASICs
Stretch	Compile a subset of C into hardware for instruction extensions
Sonics	On-chip interconnection infrastructure

Table 2.3: Technologies to map functional descriptions to hardware platforms (from [17]).

Bin M		
Provider	Tool	Description
Y Explorations	eXCite	Take virtually unrestricted ISO or ANSI-C with channel I/O behavior and generate Verilog or VHDL RTL output logic synthesis
Forte Design Systems	Cynthesizer	Behavioral synthesis
Future Design Automation	System Center co-development Suite	ASCI-C to RTL synthesis toolset
Catalytic	DeltaFX, RMS	Synthesis of DSP algorithms on processor or ASICs
ACE	CoSy	Automatic generation of compilers for DSPs

There are several dialects of C that contain constructs and primitives that represent digital hardware structures and components. For instance, the Handel-C language [25] provides language constructs to describe parallel blocks, memory entities, signals, and variables with arbitrary bit lengths.

Since digital hardware systems are inherently parallel, synchronous electronic circuits, a hardware synthesizer from C should figure out how to turn a conceptually sequential representation into an efficient parallel description. The synthesizer should also determine, by itself, the time it would take for every operation to execute, and synchronize it with the proper clock signal. In addition, the synthesizer would need to determine the best bit-length for the data elements involved to produce an area-efficient design. Since performing these three tasks automatically is a challenging task, current tools leave the parallelization of the application to the designer.

## 2.3 Functional descriptions using UML

The use of notations originally intended to model software systems as languages to describe the functionality of digital hardware systems is an incipient area of research. This research includes the development of software tools to implement the descriptions in a hardware device. The following sub-sections describe the existing works that apply an object-oriented modeling language (UML) to the design of digital hardware systems.

### 2.3.1 Modeling systems-on-a-chip using UML

A system-on-a-chip (SoC) is a computer-based system that integrates processing elements, communication links, memory hierarchies and other electronic components, either digital or analog, within a single silicon die. The complexity of these devices has made the designers turn to UML to ease their design tasks.

The Object Management Group (OMG) is a consortium of software vendors focused on driving the standardization of technologies based on object-orientation like Common Object Request Broker Architecture (CORBA) and UML. The consortium also maintains the specification of a dialect of UML (profile) that adds constructs

representing the fundamental components of every SoC (modules and channels) to UML, describes transfers of information between modules, and includes support for both synchronous and asynchronous semantics [45]. This profile extends the modeling elements in UML, adds constraints to them, and introduces specialized diagrams to describe the structure of a SoC in a hierarchical manner. The profile's specification does not impose an implementation mechanism to derive a lower level representation and gives the designer freedom to implement the models.

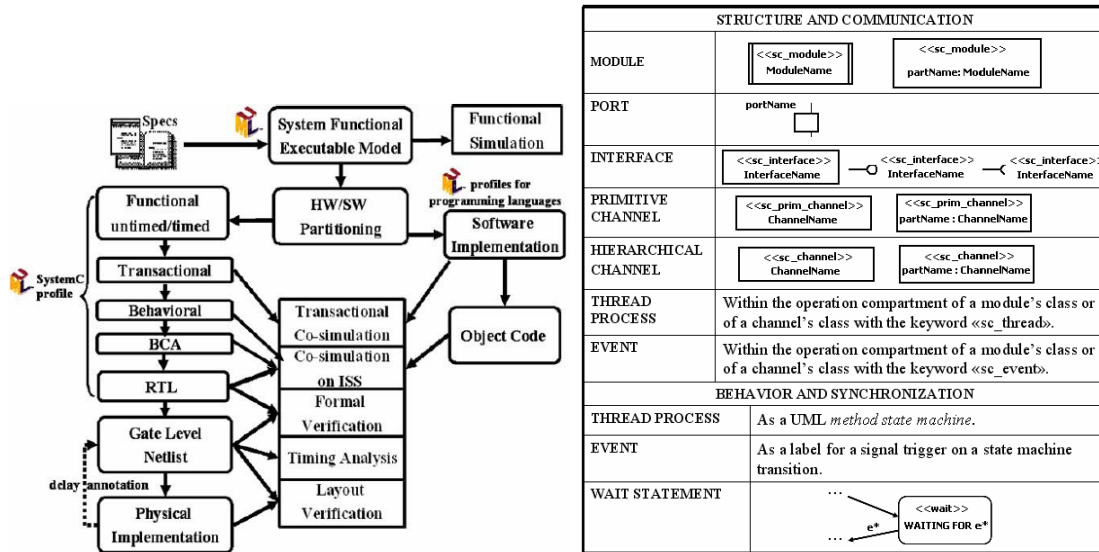
Benkermi et al. [9] describe a general UML model of a SoC platform that includes reconfigurable hardware components. The model has three layers that describe every aspect of the system: the hardware architecture of the SoC; the so-called middleware made up of the operating system, the communication protocols, and the tasks to be executed by either the microprocessor or the specialized hardware; and the high-level applications that are to be mapped to the tasks in the middleware. The purpose of this project is to support the exploration of the design space for reconfigurable SoCs and enable validation of different design choices. There is no mention of any synthesis process going from the models to an implementation in hardware or software.

Mueller et al. [40] and Riccobene et al. [57] describe a profile that adds constructs to UML to express the structural and behavioral features of a design in SystemC through diagrams. The authors also propose a design flow to model both the hardware platform of a SoC and the embedded software by means of UML, see Figure 2.1(a). According to the authors, UML improves the development process in three ways:

1. UML can be adapted to describe the overall functionality of every SoC.
2. The profile for SystemC describes a digital hardware system at a level of abstraction that is higher than that of the RTL representation of the same digital hardware system.
3. The software branch in the general flow may use UML profiles tailored for programming languages like C/C++ and Java.

The authors claim that modeling at the level of abstraction of UML has the following advantages over writing SystemC code: visualization, design reuse, integration, documentation, analysis of the model, and automatic generation of SystemC code.

The authors’ profile provides the user with the ability to “visualize” SystemC and design by building diagrams instead of writing code. Figure 2.1(b) illustrates the most significant modeling elements in the profile. Some of these elements represent the structural aspects of SystemC and the rest model the behavioral aspects of the language.



(a) Design flow to develop hardware and software (b) The components of the profile for SystemC

Figure 2.1: A design framework based on UML to design SoC platforms (from [57]).

Mellor et al. [37] describe a more advanced infrastructure to model SoCs using UML. The authors identified the following problems that occur during the process of designing a SoC:

**Partitioning.** Hardware and software engineers gather together at the beginning of the project to specify the requirements of the product. They need prototypes of the system’s hardware and software as soon as possible to determine whether both teams have the same understanding of the requirements.

**Interfacing.** The only link between separate teams with different skills working in parallel is the specification of the interface between hardware and software,

usually written in natural language. A common failure occurs when the teams neglect to document changes to the interface in the specification.

**Integration.** Correcting bugs originated from lack of communication is expensive and might represent a loss of market share. In this case, the engineers have to spend extra time correcting errors.

The authors suggest the following solutions to the previous problems:

**Creation of a single model for the application.** It is necessary to express the solution using a formal language and in an implementation-independent manner. To increase visibility and communication, this description should be at a high level of abstraction.

**Building an executable model of the application.** The execution of models enables early feedback on desired functionality.

**Do not model the structure of the implementation.** A set of mapping rules shall generate the implementation from the executable model and establish the communication mechanisms between its hardware and software components.

The authors define the process of capturing the executable model of an application as building the model using a set of data, states and functions. Then, the designer executes the model independently of its implementation and exercises it with a set of test benches to validate its functionality. During implementation, the designer marks each component with a tag indicating whether the component is synthesized as a software module or a hardware module; thus, it is possible to experiment with different ways of partitioning the implementation into hardware and software by assigning different sets of values to the tags.

### 2.3.2 Mapping from UML to VHDL

Björklund [10] describes a language called Statechart Description Language (SMDL) that can be used as an interface between high-level UML models and formal verification tools, simulators and synthesizers that generate the code of the implementation.



SMDL has formal semantics and incorporates high-level concepts like states, queues and events. A set of structural, operational rules formally defines the semantics of the statements written in this language.

Björklund et al. [11] use SMDL as an intermediate representation to transform UML statecharts into VHDL code. The UML edition tools use a standard representation based on XML, known as XML Metadata Interchange (XMI), to store models. The models in XMI are translated into preliminary SMDL code containing all of the elements in the model, including active states, transitions between states, event emission and orthogonal states. This code is first reduced by removing trivial states and applying scheduling policies. The reduced code is then transformed into software graphs (S-Graphs)<sup>1</sup> on which some optimizations occur like removal of isomorphic nodes<sup>2</sup>. Finally, the ultimate code in VHDL is generated from the optimized S-graphs.

Coyle et al. [15] describe a design flow consisting of a series of transformations going all the way from requirement specifications to hardware/software implementation. In the first phase of the design flow, two sets of UML models are generated from both functional and non-functional requirements. The functional requirements are transformed into UML structural and behavioral diagrams describing the basic functionality of the system to implement. The non-functional requirements are mapped into annotations that extend the functional UML models to indicate performance and timing restrictions. During the second phase, a set of functional UML models are implemented as software components and the rest are synthesized as hardware description language modules by means of a tool called MODCO. Figure 2.2 illustrates the translation process from UML to VHDL performed by MODCO. The UML models are converted to XMI and then processed by XML parsers that extract information that is mapped to VHDL by means of templates.

---

<sup>1</sup>A S-Graph [11] is a directed acyclic graph representing decision trees with assignment statements.

<sup>2</sup>Two nodes are isomorphic if they have the same level and their children are isomorphic.

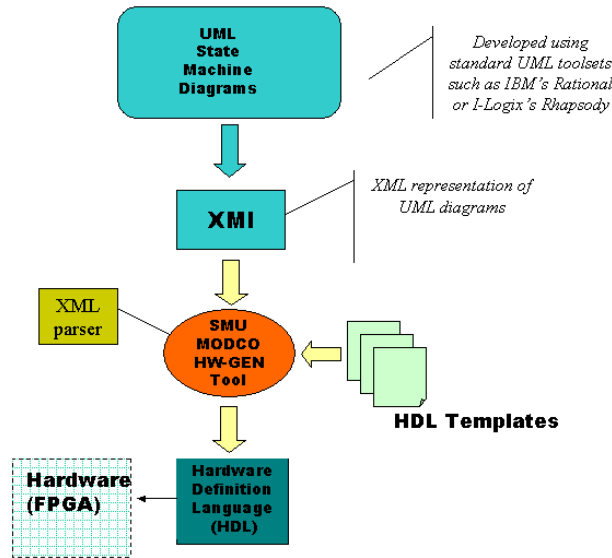


Figure 2.2: MODCO's transformation from UML to VHDL (from [15]).

## 2.4 Discussion

In spite of their advantages, the languages to describe the functionality of digital hardware systems based on C have the following drawbacks:

- The language constructs added to these subsets of C describe digital hardware components, not problem domain abstractions. Since these languages are independent of the problem domain, the designer must pay careful attention to every detail when representing, describing and implementing every abstraction in the problem domain.
- The synthesizers that transform functional descriptions into netlists<sup>3</sup> for the target silicon platforms generate hardware architectures consisting of too many resources. This situation prevents these languages from consideration by some designers of embedded systems.

Now, let us discuss the proposals using UML as the base of the corresponding design flow. The work by Benkermi et al. [9] reports a set of models to describe the

<sup>3</sup>A netlist is a description listing the electronic components that make up the system and the interconnections between them.

hardware and software components of a reconfigurable SoC. A serious limitation of this work is the lack of a transformation from the models into lower level descriptions of the hardware and software components. The OMG's profile for SoC [45] does not describe the mechanisms to implement the models built with it either. However, it is advantageous a standard modeling language to describe SoCs exists.

Mueller et al. [40] and Riccobene et al. [57] are strong advocates of using extensions to UML 2, known as profiles, to model any system at any level of detail. They mention the suitability of the new features in UML 2 to design SoCs. The authors also make the following assertion: "the use of profiles allows moving from a description of a system in a given language to the description of the same system in another language, at the same level of abstraction or lower level of abstraction" [57]. The authors' profile for SystemC facilitates the development of visual descriptions in the SystemC language and the automatic generation of code. However, since the level of abstraction at which the developer conceives the solution is still that of the SystemC language, and not that of the problem domain, there is a risk of low productivity due to the investment of time in describing implementation details. Therefore, this profile is not well-suited to high-level functional description of a SoC, but to the implementation.

Each of the works reported by Coyle et al. [15], Mellor et al. [37] and Riccobene et al. [57] proposes a design flow based on UML aimed at designing and implementing both the hardware and software components of a SoC. These design flows have the following advantages: managing a single source for the whole project, modeling every component of the system using a consistent methodology and a single language, improving the communication between the software and hardware teams, and testing the whole system within the same development environment. The conception of a design flow like this, for both the hardware and software components, is an ambitious project that is beyond the scope of this project. Thus, we focus on the proposals that synthesize lower-level descriptions of the digital hardware components.

Schattkowsky et al. [58] provide valuable recommendations related to the application of UML for digital hardware design in general and SoC design in particular:

- "UML is, by definition, a general-purpose modeling language that cannot be

instantly applied to hardware design. It is necessary to tailor UML in a way that the domain-specific requirements can be met”.

- “The real world things that need to be represented have to be identified and consistently put into the right context as UML elements”.
- “The relation between the concepts used in UML and the real circuits has to be clarified. The application of different diagrams in the design process needs to be clarified”.

This project takes these recommendations seriously and makes them the foundations of the proposed solution.

# Chapter 3

## Proposed framework

This chapter provides an overall description of the proposed framework to design digital hardware systems. This chapter presents the general principles of the model-driven development approach to design software systems and then describes the components of the proposed design framework. This design flow incorporates the general ideas behind MDE and the recommendations made by Schattkowsky [58].

### 3.1 The paradigm of model-driven engineering

MDE is a recent paradigm intended to raise the level of abstraction in the design of software systems even further. The main advantage of this approach is that it encourages the development of solutions (models) in terms of concepts in the problem domain, known well by designers or customers, instead of concepts in the solution domain related to hardware and software technologies. The overall goal is to alleviate the complexity of current hardware/software platforms by automatically translating models into an appropriate implementation that employs the technologies of one of these platforms.

A complete characterization of MDE considers the following features identified by Kent [31]:

**Domain-specific modeling.** The designers use DSMLs that incorporate abstractions, requirements and constraints from the corresponding application domains

to describe models. The syntax and semantics of every DSML must be defined in a formal fashion.

**Modeling dimensions.** In general, abstraction level is only one of multiple criteria that characterize a model. Think of these criteria as orthogonal dimensions that define a modeling space, where every point corresponds to a perspective of the model.

**Mappings or transformations.** Their goals are to produce a target implementation consistent with the information contained in the source model and keep these two entities synchronized whenever the original model changes. A design flow fully based on MDE automates the mapping process and avoids manual development of the target implementation.

**Processes.** For every project, the designers shall define a set of processes to define the order in the creation and coordination of models, establish the division of work among the teams, and define guidelines to produce models.

**Tools.** They allow the designer to build and maintain models, check the correctness of models, perform transformations between models, test the different models, and carry out the management of the whole MDE project.

**Meta-modeling.** No single modeling language is suitable for every application domain, thus the need for defining DSMLs formally. One way to do this is to derive customized languages from a single modeling language, with each version tailored to an application domain. The manipulation or extension of the definition (meta-model) of the original modeling language is necessary to create new specialized DSMLs. More ambitious trends include the automatic generation of customized tools based on a general tool by using *meta-tools*.

*Model-driven architecture* (MDA) [21, 39] is an initiative proposed by the Object Management Group (OMG) that defines a MDE-based framework using the consortium's standards, most notably UML 2.

## 3.2 Model-driven architecture

MDA separates the specification of the requirements and functionality of a software system from the implementation of such functionality using a particular technology. The goals of MDA are to enable the implementation of the same functionality on multiple platforms by means of transformations and allow the integration of different systems by relating the corresponding models. Let us analyze MDA in terms of the features of MDE stated above.

MDA categorizes models according to their level of abstraction, which is the only dimension considered by this realization of MDE. The *platform-independent models* (PIMs) are the high-level models of the solution, and the *platform-specific models* (PSMs) are the concrete derivations, each corresponding to a given hardware/software technology. In a complete scenario, the designer should be able to build, execute, test and interchange PIMs before generating the corresponding PSMs. Figure 3.1 illustrates the transformation of PIMs into PSMs for some software technologies based on Java, .NET and CORBA. Code generation from PSMs is the following transformation step and is often the simpler one.

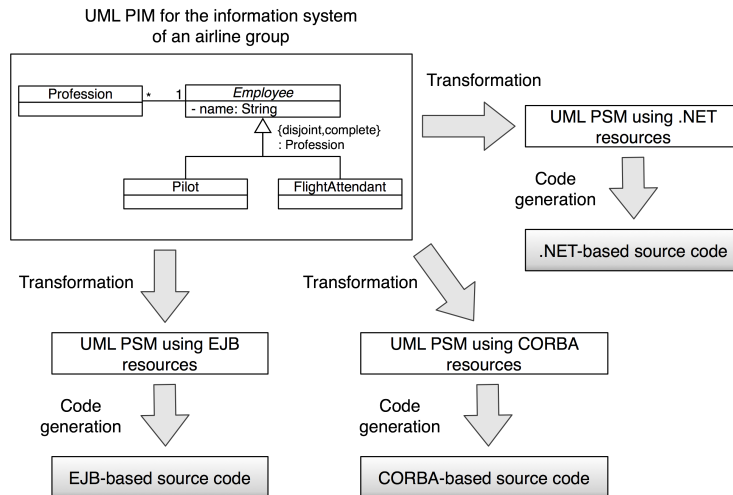


Figure 3.1: A development framework based on MDA: transformation of PIMs into PSMs and code generation (from [21]).

The language of choice to express models in MDA is UML 2, which may be tai-

lored to different application domains by means of its profiling mechanism [47, 48]. The *meta-model of UML 2* defines the structure of the language (abstract syntax) and the relationships between the modeling elements serving as basic blocks of the models. The specification of the meta-model also defines the semantics of the modeling elements, but does not state either how to store the models or what methodologies should be used to build them. The meta-model enables the definition of dialects of UML 2, known as *profiles*, used to build domain-specific models. The XML Metadata Interchange (XMI) standard, developed by the OMG, defines a notation to store, access and interchange UML 2 models between modeling tools [46, 23].

The mapping functions consist of rules that describe every step of the process of transforming a source model into a target model. MDA considers different kinds of transformations, including refinement (PIM to PIM), synthesis (PIM to PSM), re-factoring (PSM to PIM) and platform-dependent refinement (PSM to PSM) [39]. As a consequence, the models described by a given profile may be transformed into models compliant with another profile. The OMG has released two main standards indicating how to define transformations between models and transformations from models to text. The Queries/View/Transformation (QVT) standard describes three transformation languages working on models described by the meta-model of UML 2 [52]. The standard called MOF Model to Text Transformation (MOFM2T) allows the definition of algorithms that generate source code from models described by the meta-model of UML 2 [49].

The OMG does not provide a standard indicating what software process must be used to perform the development tasks using the MDA initiative. In contrast, there are plenty of modeling tools including a representation of the meta-model of UML 2 that allow developers to build diagrams in such language. Finally, there are a number of technologies that implement the standards from the OMG for transformation between models and between models and source code. The following section describes a technology to build digital hardware systems, inspired by the MDA initiative, from domain-specific models.



### 3.3 Description

The proposed design framework differs slightly from a design flow entirely based on MDA in the absence of the platform-specific models. The main principles behind the proposal in this dissertation are the following:

1. A DSML is necessary for the designer to describe functionality more effectively because it allows using terms and abstractions within the problem domain, instead of cumbersome details belonging to the implementation domain. As Mernik et al. state: “by providing notations and constructs tailored toward a particular application domain, the domain-specific languages offer substantial gains in expressiveness and ease of use compared to general-purpose languages for the domain in question, with corresponding gains in productivity and reduced maintenance costs” [38].
2. Transformation algorithms turn functional descriptions in DSML (PIMs in the jargon of MDA) into lower level representations in VHDL to be implemented on either an ASIC or FPGA. An intermediate representation in the form of a PSM is not strictly necessary because all of the information required to generate the VHDL representation can be obtained or deduced from the PIM. Additionally, the transformation between the PIM and the source code is not as complex to require partition into two transformations.
3. UML 2 has two main advantages that make it the best choice to be the foundation of the modeling languages. First, it is extensible and customizable. Second, its standard graphical notation allows a better comprehension degree than a textual language.

Figure 3.2 illustrates the proposed design framework, which consists of a DSML and a transformation tool that generates VHDL code. The models built using the DSML are functional descriptions of algorithms that manipulate blocks of bits and perform operations like block ciphering or compression. The transformation tool validates that the models conform to certain rules and generates code in VHDL. The low-level representation in VHDL describes a digital hardware system that can be implemented on a silicon platform like an ASIC or a FPGA. The implementation phase

can be carried out using other design flows marketed by third-party manufacturers of EDA technologies.

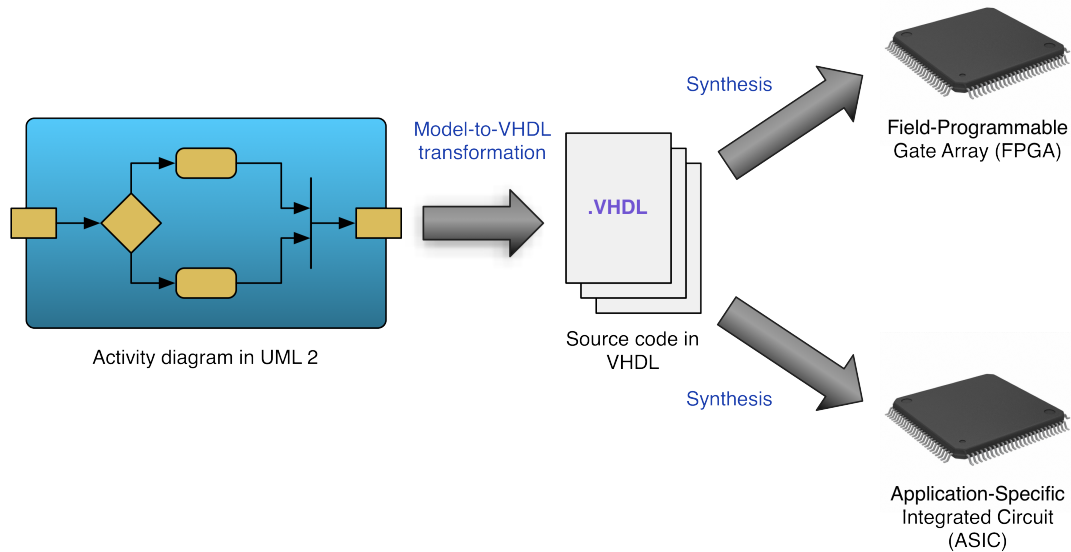


Figure 3.2: Proposed framework based on MDA.

As mentioned previously, the designer employs a version of UML 2 tailored to an application domain to build high-level models. Every model describes how an incoming block of bits flows through a series of operations that eventually produce a resulting block of bits as output. These data-flow models serve as input to the transformation tool built using a technology that transforms models into a textual representation, including source code in an artificial language. The transformation tool generates VHDL code compliant with a reduced version of the formal grammar of such language. The transformation process consists of a number of templates producing the skeleton of the source code and queries validating the models and retrieving information from them to complete the resulting source code. This framework aids in the design of digital hardware accelerators for digital communication devices.

## 3.4 Defining and bounding the application domain

The demand for mobile computing devices capable of communicating with each other and consuming power efficiently has increased notably during the last decade. For this reason, the designers have conceived small and power-efficient digital hardware architectures to enable a computing device to transmit and receive signals through different channels. For instance, Arditti et al. [4] proposed a dedicated processor, consisting of hardware accelerators, to process multiple wireless communication protocols concurrently.

This dissertation applies the proposed design framework to build digital hardware architectures that perform operations involved in digital communication systems. The intention is to contribute to increase the productivity of the designers that implement the digital communication sub-system of modern computer-based devices. The next paragraphs characterize the domain of digital communication systems and identify the areas within it where the proposed framework is to be used.

### 3.4.1 Digital communication systems

The goal of a digital communications system (DCS) is to transmit a digital message from an information source to an information target, or sink, through a transmission channel. Figure 3.3 illustrates the processing stages that the message goes through before the transmitter emits it and after the receiver picks it up.

The format stage samples and quantizes the analogue signal from the source and transforms it into a bitstream consisting of a number of  $k$ -bit groups called message symbols. Three successive algorithms process the  $M = 2^k$  different symbols ( $m_i, i = 1, 2, \dots, M$ ) prior to their modulation. The first algorithm performs source encoding (compression) to represent the symbol with the minimum number of bits. The second algorithm ciphers the information to guarantee privacy and prevent unauthorized access. The third algorithm (channel encoding) adds redundancy bits to the information, so the receiver detects errors and corrects them if possible. The channel encoding phase produces a set of channel symbols ( $u_i, i = 1, 2, \dots, M$ ). A clock signal from a synchronization element, shown in Figure 3.3, controls these signal processing

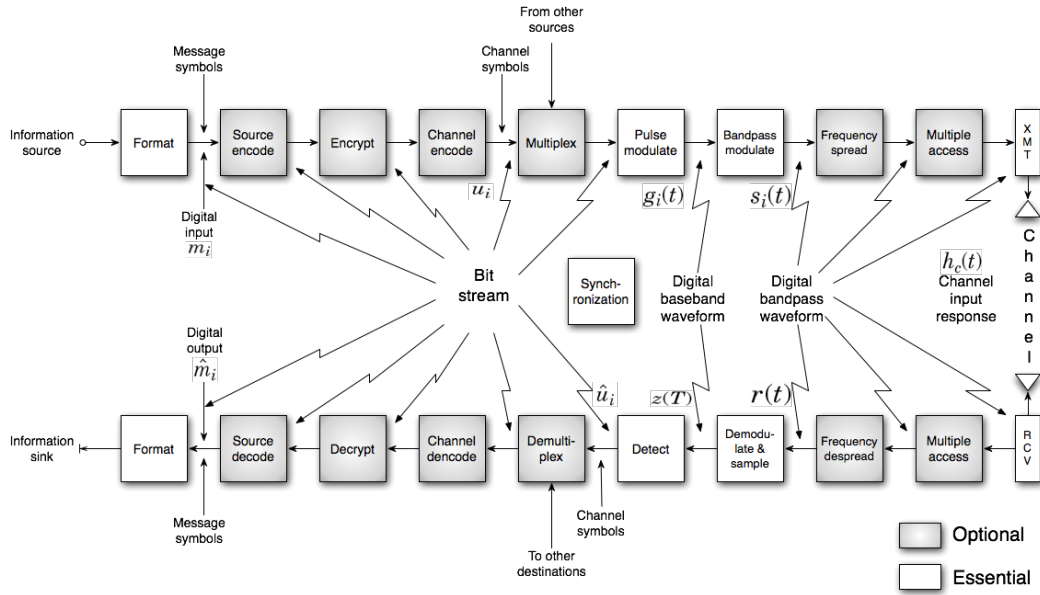


Figure 3.3: Block diagram of a typical digital communications system (from [60]).

phases.

The pulse modulation phase transforms each of the  $M$  possible channel symbols into the corresponding baseband waveform, characterized by its short bandwidth. When there are only two possible channel symbols (0 and 1), there will only be two possible baseband waveforms, known as pulse-code modulation (PCM) waveforms ( $g_i, i = 1, 2, \dots, M$ ). If required by the transmission channel, another modulation phase transforms the baseband waveforms into bandpass waveforms ( $s_i, i = 1, 2, \dots, M$ ). The frequency of the bandpass waveforms is much larger than the spectral content of  $g_i$  due to the effect of a carrier wave.

The phases of multiplexing and multiple-access combine signals from different sources to share a single transmission channel. Frequency spreading (or spread spectrum) refers to a set of methods to spread the signal's frequencies along the frequency domain to produce a signal having a wider bandwidth, an increased resistance to interference and more security in its transmission.

The receiver implements algorithms to carry out the inverse operations of those performed by the transmitter (demodulation, decoding, decipher, decompress, and

demultiplexing). The output of the whole process in the receiver, denoted as  $\hat{m}_i, i = 1, 2, \dots, M$ , is an estimation of the original message symbol transmitted by the source, which could have been affected by noise and fading during transmission through the channel.

Although the data flow shown in Figure 3.3 is standard, there is plenty of room for innovation of new algorithms for each processing stage of a DCS. This innovation also includes new techniques and architectures to implement those algorithms. These implementations may be designed to meet any of the requirements of digital hardware systems (high performance, low area cost or low power consumption) or a trade-off between them.

The problem domain of interest for this project consists of the concepts, abstractions and primitive operations used to build algorithms that perform the following complex operations of a DCS:

1. Source coding:
  - (a) Ciphering: block ciphering, stream ciphering.
  - (b) Compression: lossless, lossy.
2. Channel coding (forward error correction).
3. Arithmetic in finite fields.

This dissertation illustrates how to use the proposed design flow to describe and transform algorithms that carry out ciphering and arithmetic in finite fields. Since the design flow processes hierarchical models, it is also possible to model algorithms performing the other complex operations.

# Chapter 4

## The domain-specific modeling language

This chapter describes the high-level language used to build models, which is the result of tailoring UML 2 by means of its profiling mechanism. This domain-specific profile allows the designer to describe and manipulate the structure of a number of algorithms operating on bit-blocks without modifying their behavior. The intention of the manipulations is to increase the performance or reduce the number of resources consumed by the hardware implementations of the algorithms. The modeling language includes constructs representing operations on bit-blocks, and allows the creation of hierarchical models.

To understand the process of building profiles that adapt UML 2 to different application domains, it is necessary to acquire a basic understanding of the meta-model of UML 2. The meta-model provides the *abstract syntax* of UML 2, which defines the structure of the language, and states the rules to build well-formed and valid models. The meta-model is to UML 2 what a formal grammar is to an artificial language like C, Java, Smalltalk and VHDL. This chapter introduces the concepts of meta-modeling and meta-class, briefly reviews the meta-model of UML 2 and the mechanism to define profiles, and describes the construction of the dialect of UML 2 to model algorithms on bit-blocks.

## 4.1 Overview of meta-modeling

The *domain model* in Figure 4.1 shows the classes defining the entities that make up an airline company, which include different kinds of employees, different kinds of aircrafts and different kinds of facilities. The diagram also shows the existing links between the entities, indicated through relationships between the classes, and possible constraints associated to the entities; see Appendix A for a review of the notation of class diagrams. This conceptual model characterizes the application domain of airline enterprises through object-oriented concepts like classes, attributes, operations, relationships between classes and constraints. This model is a valuable asset when developing information systems that automate crucial processes of an airline company like payroll, aircraft inventory, assignment of employees to facilities and more. However, the *design models* describing the structure of such information systems might be different from the model in Figure 4.1.

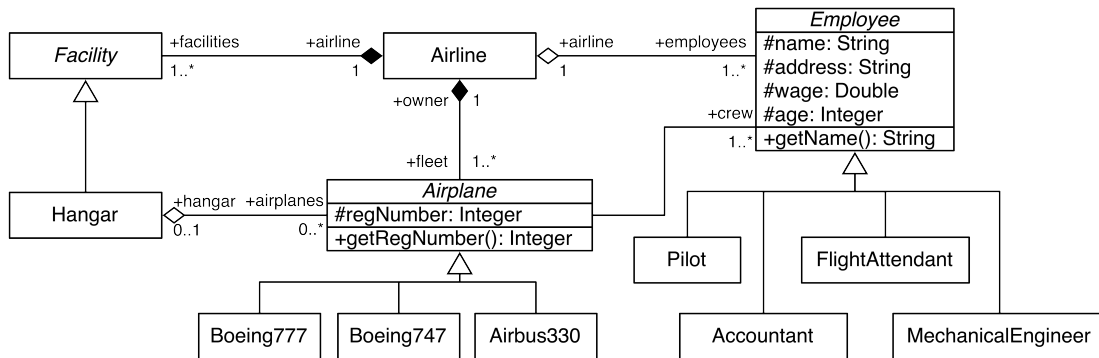


Figure 4.1: Domain model illustrating the organization of an airline company.

The model in Figure 4.2 illustrates the objects  $p_1$ ,  $a_1$ ,  $fa_1$  and  $me_1$ , which are instances of concrete sub-classes of the abstract class **Employee** and represent employees of the airline company. The diagram also shows the links between  $p_1$  and  $ac_1$ , and between  $fa_1$  and  $ac_1$ . The object  $ac_1$  represents an aircraft in the fleet of the company, and the links indicate what employees are part of the crew of such aircraft. This basic example highlights that the domain models in figures 4.1 and 4.2 represent the entities in a domain of interest, and the relationships between them, in

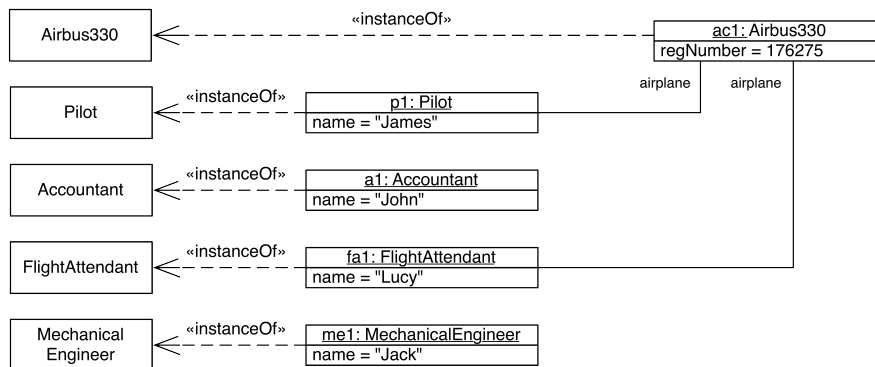


Figure 4.2: A model including instances of sub-classes of **Employee** and their links to an instance of **Airplane**.

an accurate fashion. UML 2 includes constructs to represent classes, association relationships between classes, objects (instances of the classes) and links between objects (instances of the association relationships).

UML 2 also provides a notation to define *meta-classes*, which are classes whose instances are other classes. To understand this concept, imagine that the business group owning the airline company also owns a flight academy that teaches professions like pilot, flight attendant, accountant and mechanical engineer. From the point of view of the flight academy, pilot, flight attendant, accountant and mechanical engineer are instances of the concept “profession”. The list of attributes defining a profession includes the name of the profession, the set of subjects to study to get the corresponding degree, the number of years needed for completion, a reference to the organization that certifies professionals in that area, among others. If we extended the domain model in Figure 4.1 with this information, the concepts pilot, flight attendant, accountant and mechanical engineer would acquire a dual nature. In addition to being classes defining their own instances, these concepts would become instances of a meta-concept or meta-class called **Profession**. Figure 4.3 shows how to indicate that the classes **Pilot**, **FlightAttendant**, **Accountant** and **MechanicalEngineer** are also instances of the meta-class **Profession** in UML 2.

Now, what is the difference between *modeling* and *meta-modeling*? By modeling, the analyst abstracts properties from the real world; by meta-modeling, the analyst



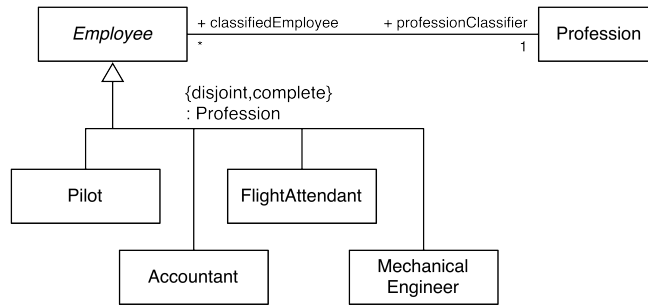


Figure 4.3: An example of the notation used in UML 2 to denote that a number of classes are instances of a specific meta-class.

abstracts properties from the model of the real world. Some authors have published many extensive studies about the fundamentals of meta-modeling [6, 5, 32, 42] and its application to the definition of UML during the last two decades. The meta-model of UML 2 is the model that characterizes the models built using such language [47, 48]; it consists of a number of meta-classes that describe the modeling elements used to build diagrams.

## 4.2 Introduction to the meta-model of UML 2

Figure 4.4 illustrates a framework based on meta-modeling that places software objects, design models in UML 2, and meta-classes from the meta-model of UML 2 at different conceptual layers or *meta-levels*. This framework also defines the dependencies between the elements at each layer. The diagram in Figure 4.4 also illustrates that the specification of the meta-model of UML 2 uses class diagrams to describe classes, meta-classes and the relationships between one another. Thus, the specification defines the abstract syntax (structure) of UML 2 using the concrete syntax (notation) of UML 2 itself.

### 4.2.1 The meta-levels

The bottom meta-level, referred to as M0, contains the software objects that comprise a running software system. Figure 4.4 illustrates that there is a video-decoding

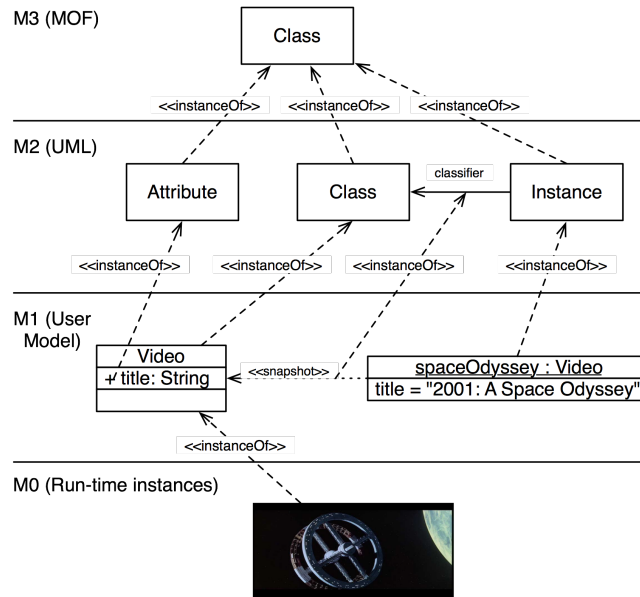


Figure 4.4: The meta-levels in the framework that defines UML 2 (from [47]).

software object, referred to within the system as *spaceOdyssey*, at the meta-level M0. The meta-level M1 contains the models in UML 2 built by the designer to describe the software objects. Figure 4.4 illustrates a model of the class **Video** and a model of the object *spaceOdyssey* itself at the meta-level M1. The software object *spaceOdyssey* at M0 is an instance of the class **Video** at M1, represented by the traditional modeling element for classes in UML 2.

The rules and notation defined by the meta-model of UML 2 allow the designer to build M1 models and recognize its components. The designer of the M1 model in Figure 4.4 employed the class box and the attribute entry to define the class **Video**, and the instance box to represent the object *spaceOdyssey*. Every component of the design model is an instance of a meta-class located at the meta-level M2. This meta-level contains the meta-classes **Class**, **Attribute** and **Instance** that define the modeling elements used to build the M1 model. Figure 4.4 illustrates the “instance of” relationship existing between run-time software objects, modeling elements, and the meta-classes comprising the meta-model of UML 2.

Now, consider the class **Video** at the meta-level M1 in Figure 4.4. On the one

hand, the class **Video** is an instance of the meta-class **Class** from a *lexical point of view* because the modeling element used to represent it is the class box. The meta-class **Class** defines a number of properties present in all of its instances, including the name of the class represented by the instance, and an indication of whether the instance represents an abstract class. The values for such properties of the modeling element representing the class **Video** are “Video” and *false*, respectively. On the other hand, the class **Video** may be conceived as an instance of the meta-class **Medium** from an *ontological point of view*, along with other classes like **Music** and **Image**.

The uppermost meta-level in Figure 4.4, commonly referred to as the meta-meta-model level, or M3 level, contains the Meta-Object Facility (MOF) [43]. MOF is a specification released by the OMG that sets the foundation for the definition of closely related meta-models, being UML 2 one of them. Every modeling tool supporting UML 2 must contain an internal implementation of the meta-model from which the tool instantiates the elements in user models. The modeling tools must also provide the designer with the mechanism to extend the meta-model and define profiles.

### 4.3 Definition of profiles in UML 2

Figure 4.5 shows a diagram consisting of three containers of models called packages. The package **SampleProfile** contains a profile defining new modeling elements affecting the meta-class **Class** in the package **SampleMetamodel**, which represents the meta-model of UML 2. The design model in the package **SampleModel** describes a payroll software system for an airline company that defines the classes **Employee**, **FlightAttendant** and **Pilot**. The intention of the profile is to extend the design model to indicate that the classes **FlightAttendant** and **Pilot** also represent professions taught in a flight academy. The extension initiates when the designer applies the profile in **SampleProfile** to the design model in **SampleModel**, then the modeling tool creates two instances of **Profession**, attaches them to **FlightAttendant** and **Pilot**, and indicates the extension with the keyword «profession» on top of the name of the classes.

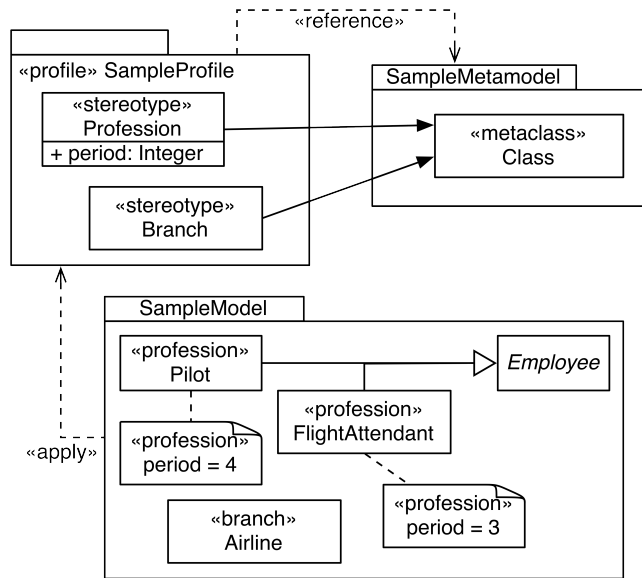


Figure 4.5: Definition of a profile that extends a M2 meta-model and its application to extend a M1 model.

Defining profiles is a meta-modeling task. The distinct meta-class **Profession** in the profile **SampleProfile** extends the meta-class **Class** in the meta-model **SampleMetamodel**. This means that instances of **Profession** are able to extend the modeling elements in **SampleModel** that are instances of **Class**. As a result, **FlightAttendant** and **Pilot** may represent not only specializations of the abstract class **Employee**, but also instances of the concept profession, represented by the stereotype **Profession**<sup>1</sup>. Notice that **Profession** defines an integer property (*period*) that extends the list of properties of every instance of **Class**. Both **FlightAttendant** and **Pilot** receive this property and assign a different value to it. The influence of M2 **Profession** extends only to M1 **Pilot** and M1 **FlightAttendant**, not to M0 instances of such classes.

Figure 4.5 illustrates the extension relationships between each stereotype and

<sup>1</sup>A *stereotype* is a kind of meta-class whose purpose is to extend meta-classes in the meta-model of UML 2 by adding properties and constraints. At modeling time, the instances of stereotypes extend the corresponding modeling elements to represent together concepts in the corresponding application domain.

the meta-class **Class**, the package import relationship between `SampleProfile` and `SampleMetamodel`, and the application relationship between `SampleModel` and the package containing the profile. The keywords «profession» and «branch» on top of the names of the classes in `SampleModel` indicate that the instances of the stereotypes in `SampleProfile` extend the instances of **Class** in the model. The note attached to the stereotyped instance of **Class** indicates the value of the tagged attribute *period* in the corresponding instance of the stereotype **Profession**.

When building a profile, the designer must define a number of elements: stereotypes, tagged attributes, and constraints. The stereotypes extend meta-classes in the meta-model of UML 2 to obtain “new” modeling elements. The tagged attributes defined by the stereotypes extend the list of the properties of the meta-classes extended by the stereotypes. The designer may also define constraints, written in the Object Constraint Language (OCL) [44], to specify invariant rules, preconditions and post conditions for the stereotypes in the profile. Finally, a stereotype may also include an optional icon indicating a new notation for the extended modeling element.

To define a profile the designer first selects the diagrams and modeling elements bearing certain resemblance to the domain-specific abstractions or entities to represent. Then, the designer extends the meta-classes of the selected modeling elements using stereotypes. The stereotypes may include tagged attributes conveying relevant information not available in the extended meta-class, and some expressions in OCL that impose restrictions to the model built upon instances of the meta-classes extended by the profile. The definition of profiles, the construction of models, and the extension of models by profiles are operations performed using a modeling tool supporting UML 2 and its profiling mechanism.

## 4.4 Definition of the domain-specific modeling language

The algorithms in the application domain considered for this project receive a number of bit-blocks as input, transform them through successive operations to generate intermediate results, and produce another bit-blocks as output. This flow of data,

including the operations indicated by the algorithm, can be described in UML 2 using the activity diagram. The activity diagrams consist of a number of different kinds of nodes connected to each other through edges. Please read the description of activity diagrams in Appendix A to acquire a basic understanding of them.

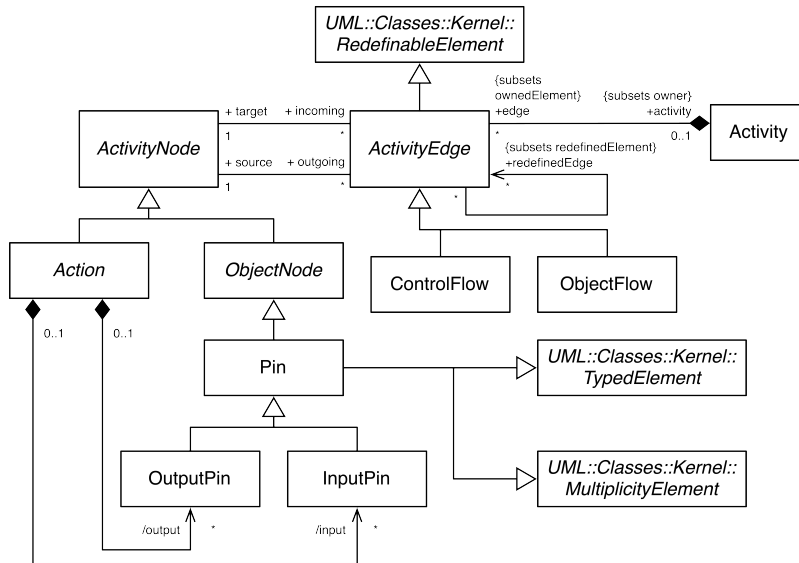
Every activity diagram is a composite object made up of multiple objects assembled together. All of these objects are instances of the meta-classes that populate the meta-model of UML 2. By studying the meta-classes defining the elements of activity diagrams, shown in Figure 4.6, it is possible to determine how to adapt this kind of diagram to model algorithms like block ciphers, compressors and coders in an accurate fashion. The profile described in this section defines stereotypes extending the meta-classes in Figure 4.6, and constraints driving the construction of well-formed activity diagrams.

#### 4.4.1 Adapting activity diagrams to model flows of bit-blocks

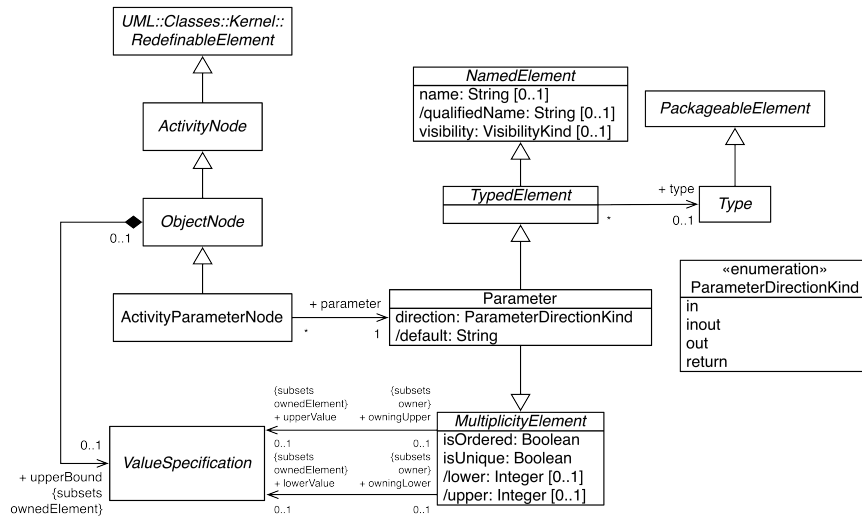
Since activity diagrams are used to describe flows of data in a wide variety of areas, it is necessary to adapt them to the application domain of interest. This adaptation requires representing the abstractions in the application domain using existing modeling elements in the activity diagrams. Very often, it is necessary to restrict the values of the attributes of the modeling elements in the diagrams, and establish how the nodes can be connected to each other. The transformation tool determines whether an input model meets these restrictions before generating the corresponding source code in VHDL. In addition, the expressiveness of the resulting profile can be improved by assigning new graphical notation to the constrained modeling elements.

The following list introduces some of the abstractions in the application domain, and exemplifies the restrictions they must meet. Both the abstractions and the constraints were identified after an examination of different algorithms in the application domain of interest:

1. A well-formed bit-block is an ordered sequence of objects that are instances of the class **Bit**. The length of the sequence must be nonzero and finite.
2. A module describes either a simple operation on input bit-blocks that performs

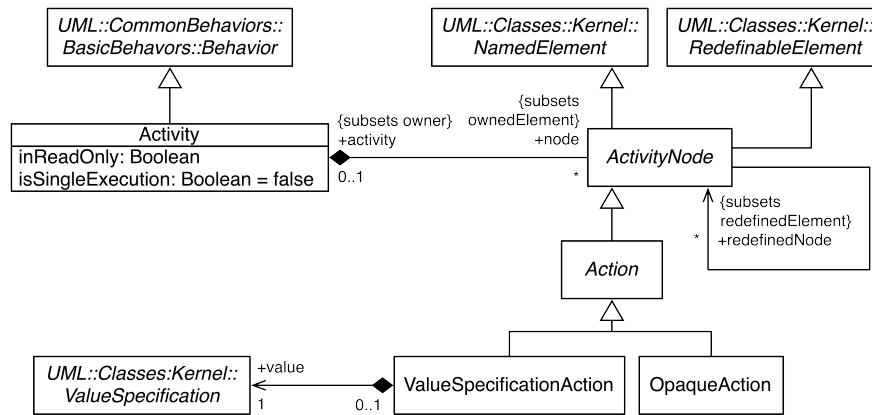


(a) An instance of the meta-class **Activity** owns a number of nodes (instances of the meta-class **ActivityNode**) interconnected by edges (instances of the meta-class **ActivityEdge**)



(b) The nodes representing input and output values are instances of the meta-class **ActivityParameterNode** and are related to instances of the meta-class **Parameter**

Figure 4.6: Extracts of the meta-model of UML 2 defining some modeling elements in an activity diagram.



- (c) An instance of the meta-class **Activity** owns special kinds of nodes indicating operations (instances of concrete sub-classes of the abstract meta-class **Action**)

Figure 4.6: Extracts of the meta-model of UML 2 defining some modeling elements in an activity diagram (cont.)

- primitive operations and produces output bit-blocks, or a complex algorithm involving elaborate flows of bit-blocks and invocations to other modules. This abstraction is represented by an activity in UML 2.
- Every module has parameters (extended instances of the meta-class **ActivityParameterNode**) that send/receive a continuous flow of bit-blocks of a fixed length. This guarantees that the model is always ready to work.
  - A module contains nodes that carry out operations or control the flow of bit-blocks. Most of the nodes receive input operands and produce output operands.
  - Nodes are connected to each other through edges. The only edges allowed in a module are those representing flows of data (extended instances of the meta-class **ObjectFlow**), and are referred to as dataflows.
  - A node cannot be the target of multiple dataflows.
  - Two nodes connected by a dataflow must produce or receive bit-blocks of the same length.



8. Every operand (an extended instance of the meta-class **Pin**) of an operation node (an extended instance of concrete sub-classes of the abstract meta-class **Action**) must receive or produce a continuous flow of well-formed bit-blocks.
9. Every operation node must specify an operation supported by bit-block algorithms (xor, and, or, shift left, shift right, rotate left, rotate right, split, concatenation). In case of nodes specifying constants, these must be integer values.
10. Every operation node must have the appropriate number of input and output operands:
  - (a) An operation node indicating a binary bitwise logic operation (xor, and, or, nand, nor, xnor) must have at least two input operands and only one output operand.
  - (b) An operation node indicating a shift or rotate operation (shift left, shift right, rotate left and rotate right) must have one input operand and one output operand processing bit-blocks of the same length. A second input operand must specify the number of bits to shift/rotate.
  - (c) An operation node indicating a constant value must have a single output parameter and no input parameters.
  - (d) An operation node indicating the split of its single input operand must have multiple output operands, such that the sum of the lengths of the output operands equals the length of the input operand.
  - (e) An operation node indicating the concatenation of multiple input operands must have a single output operand, such that the sum of the lengths of the input operands equals the length of the output operand.
11. The operands must meet other restrictions imposed to them by the owner operation.

The proposed modeling language includes a number of stereotypes that extend the meta-classes shown in Figure 4.6 to closely represent the abstractions in the

application domain, and defines constraints like the ones stated above. The designer uses the declarative language OCL to express these constraints, which use the values of the attributes of the extended meta-class, as well as its relationships with other meta-classes. The transformation tool uses the statements in OCL to validate that an input model conforms to the restrictions (the model is well-formed) prior to generating source code in VHDL.

#### 4.4.2 The organization of the modeling language

The meta-model of UML 2 groups all of its meta-classes into packages. An import relationship from package A to package B indicates that A uses the meta-classes in B to define its own meta-classes. A merge relationship from package A to package B indicates that A may add properties and associations to the meta-classes in B to extend their definitions. Figure 4.7 illustrates that the packages Communications, CompleteActions, CompleteActivities and BehaviorStateMachines in the meta-model of UML 2 are related to several other packages in the meta-model through merge and import relationships. As a result of these relationships, these packages end up containing the meta-classes extended by our profile, called BitBlockFlow.

The package UMLforBitBlockFlow merges the meta-classes from the packages Communications, CompleteActions, CompleteActivities and BehaviorStateMachines into a single repository. The goal is that the stereotypes in the profile BitBlockFlow extend the meta-classes of interest from the same place, instead of extending meta-classes from multiple packages in the meta-model. The reference relationship between BitBlockFlow and UMLforBitBlockFlow indicates that the profile imports all of the meta-classes contained in the package to extend them with stereotypes.

BitBlockFlow consists of seven packages containing the profile's stereotypes, as illustrated in Figure 4.8. The package ModuleInterface contains the stereotypes representing the basic unit of modeling of a bit-block flow (the module) and its interface to the outside world (input and output parameters). The package Types contains the class **Bit**, which is crucial in the definition of the entities processed by the modeling elements in the profile. The package Operations contains stereotypes representing bit-wise logic, shift and rotate operations on bit-blocks of variable length; it also contains

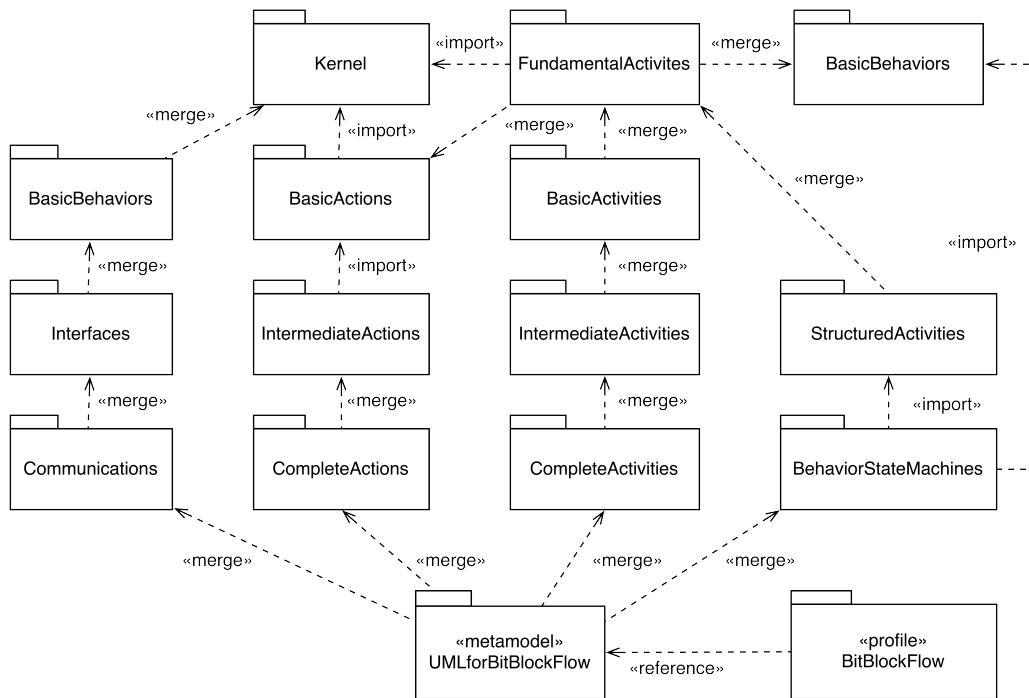


Figure 4.7: The packages in the meta-model of UML 2 extended by the profile.

stereotypes performing other manipulations on bit-blocks. The stereotypes located in the package Edges constraint the edges that transfer objects between nodes in an activity diagram. The elements defined in the package Literals allow including integer values in the model. The packages Control and Synchronization contain stereotypes that direct the flow of data within the model and synchronize output parameters with external signals. The stereotypes in the package StateMachines allow specifying state machines that drive the behavior of switch operations.

The designer uses a modeling tool to build and define the profile, and then applies the stereotypes to instances of the extended meta-classes in an activity diagram. As a result, the modeling elements in the affected models get extended by instances of the stereotypes and become representations of abstractions and operations in the application domain.

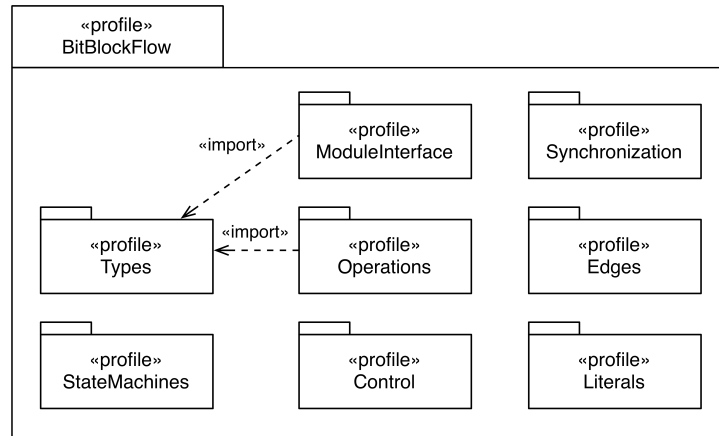


Figure 4.8: The internal structure of the profile BitBlockFlow.

### 4.4.3 Modules and their interfaces

Figure 4.9 illustrates that the stereotype **BBModule** extends the meta-class **Activity**, and that the stereotype **BBParameter** extends the meta-class **ActivityParameterNode** in the meta-model of UML 2. This means that a module is an abstraction that behaves like an activity, but with special features that only make sense when modeling operations in the application domain of interest. The parameters behave like a parameter node in the sense that they are attached to an activity and send or receive objects; however, they are restricted to deal with bit-blocks only. Figure 4.10 illustrates the use of these elements in a simple model. There is no new graphical notation to denote modules and parameters, because the standard concrete syntax is adequate.

The following is a list of the restrictions that instances of **ActivityParameterNode** extended by instances of **BBParameter** must meet:

1. Every parameter must be either an input parameter, or an output parameter, or an input/output parameter.
2. Every module must have at least one input parameter and at least one output parameter; otherwise, at least one input/output parameter.
3. Every parameter must receive/send an unbounded number of well-formed bit-

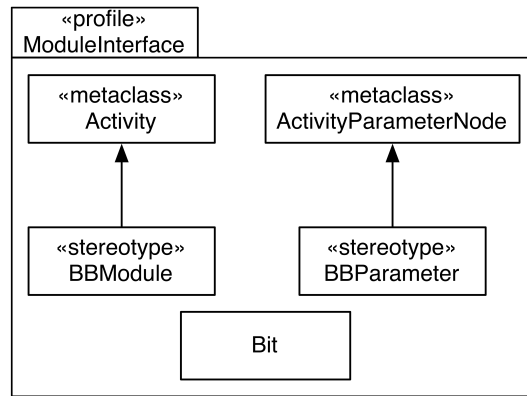


Figure 4.9: The stereotypes in the package ModuleInterface and the meta-classes extended by them.

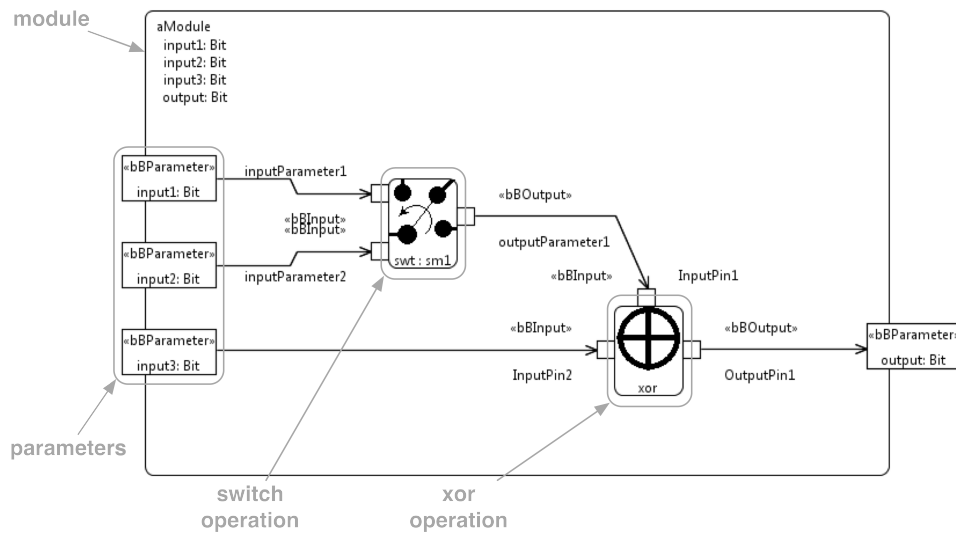


Figure 4.10: A module, its parameters and some operations.

blocks of the same length. However, the length of the bit-blocks processed by different parameters may differ.

4. Every input parameter must have at least one outgoing dataflow and no incoming dataflow. Every output parameter must have only one incoming dataflow and no outgoing dataflow. Every input/output parameter must have at most one incoming dataflow or at least one outgoing dataflow, but not both at the same time.
5. The names of the parameters attached to the same module must be different.

The OCL declarative language is useful to express the previous restrictions in a formal way, so the transformation evaluates them when validating the parameters of a module. The following queries in Acceleo's version of OCL [41] illustrate the evaluation of the restrictions for all of the parameters in a module and for a single parameter:

```

1.  [comment    Determines whether the parameters in the receiving module are well-formed. /]
2.  [query public validateParameters(aModule: Activity): Boolean =
3.      if (aModule.validateNumberParameters()) then
4.          aModule.getParameters()
5.          ->forAll(an: ActivityParameterNode | an.validateParameter())
6.      else
7.          false
8.      endif
9.  /]
10.
11. [comment    Determines whether the receiving parameter is well-formed. /]
12. [query public validateParameter(aParameter: ActivityParameterNode): Boolean =
13.     aParameter.validateType() and
14.     aParameter.validateMultiplicity() and
15.     aParameter.validateUpperBound() and
16.     aParameter.validateDirection() and
17.     aParameter.validateNumberDataflows() and
18.     aParameter.isUnique()
19.  /]

```

The next two queries evaluate the restrictions 1 and 4 stated above, respectively. These queries employ other queries that retrieve information from the elements of the model and use it to compute their returned values. The information provided by the model lies in the values of the attributes of the objects that make up the model, and in the associations of these objects with one another. The queries in OCL are able

to retrieve the values of the attributes and associations of the objects on which they are invoked, and perform simple or complex computations with them.

```

1.  [comment    Determines whether the receiving module has input and output parameters
2.            or input/output parameters. /]
3.  [query public validateNumberParameters(aModule: Activity): Boolean =
4.    (aModule.getNumberInputParameters() > 0 and
5.    aModule.getNumberOutputParameters() > 0) or
6.    aModule.getNumberInoutParameters() > 0
7.  /]
8.
9.  [comment    Determines whether the number of incoming/outgoing dataflows to/from the
10.            receiving parameter conforms to the rules imposed by UML 2, depending on
11.            the direction of the data in the parameter. /]
12. [query public validateNumberDataflows(aParameter: ActivityParameterNode): Boolean =
13.  (aParameter.isInputParameter() and
14.  aParameter.getNumberIncomingDataflows() = 0 and
15.  aParameter.getNumberOutgoingDataflows() > 0) xor
16.  (aParameter.isOutputParameter() and
17.  aParameter.getNumberIncomingDataflows() = 1 and
18.  aParameter.getNumberOutgoingDataflows() = 0) xor
19.  (aParameter.isInoutParameter() and
20.  (aParameter.getNumberIncomingDataflows() > 1 xor
21.  aParameter.getNumberOutgoingDataflows() > 1))
22. /]

```

A module is well-formed if all of its parameters, operations, control nodes and dataflows meet their corresponding restrictions. Also, if a module contains invocations to other modules, the invoked modules must be well-formed. Thus, if a module invokes an invalid module, the invoking module becomes invalid as well. The query that determines whether a module is well-formed or not invokes the queries that validate such property for every kind of node and dataflow in the module.

```

1.  [comment    Determines whether the receiving module is well-formed. /]
2.  [query public validateModule(aModule: Activity): Boolean =
3.    aModule.isBBModule() and
4.    aModule.validateName() and
5.    aModule.validateParameters() and
6.    aModule.validateDataflows() and
7.    aModule.validateLogicOperations() and
8.    aModule.validateIntegerConstants() and
9.    aModule.validateShiftRotateOperations() and
10.   aModule.validateSplitOperations() and
11.   aModule.validateConcatenationOperations() and

```

```

12.     aModule.validateJoins() and
13.     aModule.validateSyncs() and
14.     aModule.validateModuleCalls() and
15.     aModule.validateExtractionOperations() and
16.     aModule.validateSwitchOperations()
17.  /]

```

#### 4.4.4 Operations and their operands

The proposed profile defines a number of operations in the package `Operations`. The modeling language provides stereotypes for bitwise logic operations (and, or, xor, nand, nor, and xnor), shift and rotate operations, operations that split a bit-block into shorter bit-blocks, operations that extract a bit-block from a larger bit-block, operations that concatenate bit-blocks to form a larger bit-block, and operations that zero-extend a bit-block. Figure 4.11 shows that the stereotypes defining the bitwise logic operations, and the stereotypes defining the shift and rotate operations, are sub-classes of the abstract stereotype **BBOperation**. Figure 4.12 illustrates the new concrete syntax assigned to the modeling elements extended by instances of the stereotypes.

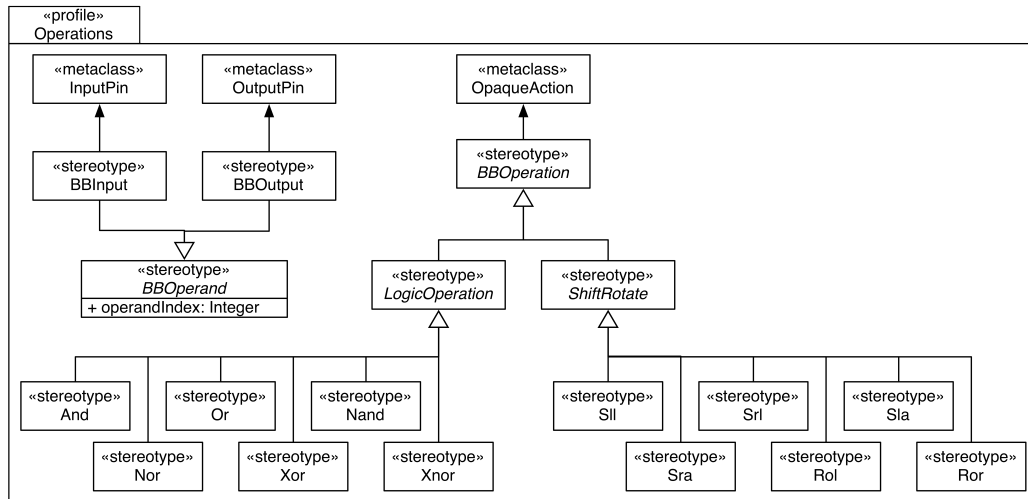


Figure 4.11: The stereotypes in the package `Operations` and the meta-classes they extend.



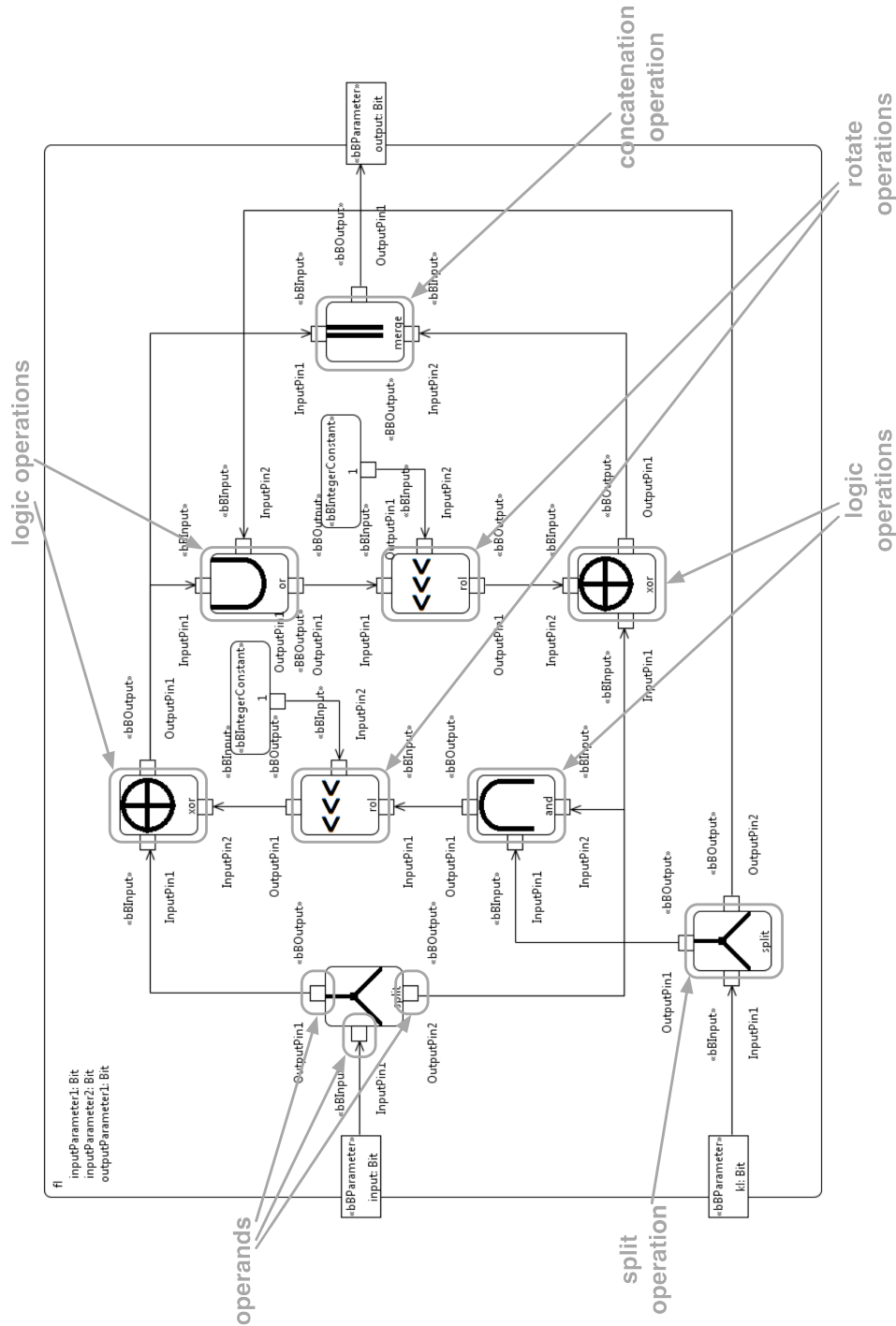


Figure 4.12: Different kinds of operations and their input and output operands.

Notice in Figure 4.12 that every operation has input operands that receive the source data, and output operands that issue the result of the operation. An operand is an abstraction represented by an instance of the abstract meta-class **Pin** that has been extended by an instance of the stereotype **BBInput** or **BBOutput**. Figure 4.11 shows that the stereotypes **BBInput** and **BBOutput** extend the concrete meta-classes **InputPin** and **OutputPin**, and define a new attribute called *operandIndex*, used to denote segments of a bit-block in operations that split or concatenate bit-blocks.

Now, let us examine the restrictions specified for operands by the profile:

1. Every operand must receive/send an unbounded number of well-formed bit-blocks of the same length. However, the length of the bit-blocks processed by different operands may differ.
2. Every input operand must have only one incoming dataflow and no outgoing dataflow. Every output parameter must have at least one outgoing dataflow and no incoming dataflow. Every input/output parameter must have at least one incoming dataflow or at least one outgoing dataflow, but not at the same time.
3. Every kind of operation has its own restrictions on the value of the attribute *operandIndex*, especially the operations that split and concatenate bit-blocks.

The following queries in OCL evaluate these restrictions and use other queries to retrieve information from the activity models and compute the required values:

```

1. [comment    Determines whether the output operands extended by the stereotype
2.            BitBlockFlow::BBOutput of the receiving operation meet the constraints
3.            for their types, their multiplicities and their upper bound. /]
4. [query public validateOutputs(aOperation: Action): Boolean =
5.     aOperation.getOutputs()
6.     ->forAll(op: OutputPin | op.validateOperand())
7. /]
8.
9. [comment    Determines whether the input operands extended by the stereotype
10.           BitBlockFlow::BBInput of the receiving operation meet the constraints
11.           for their types, their multiplicities and their upper bound. /]
12. [query public validateInputs(aOperation: Action): Boolean =

```

```

13.     aOperation.getInputs()
14.     ->forall(ip: InputPin | ip.validateOperand())
15.   /]
16.
17.   [comment    Determines whether the receiving operand meets the constraints
18.           for its type, its multiplicities and its upper bound. /]
19.   [query public validateOperand(anOperand: Pin): Boolean =
20.       anOperand.validateType() and
21.       anOperand.validateUpperBound() and
22.       anOperand.validateMultiplicity() and
23.       anOperand.validateNumberDataflows()
24.   /]

```

In addition to specifying graphical icons for instances of the extended meta-class **OpaqueAction**, the profile also specifies restrictions for the sub-classes of the abstract stereotype **BBOperation**. The restrictions specified for the all of the bitwise logic operations are the following:

1. Every bitwise logic operation must have at least two input operands and one output operand.
2. The operands of every bitwise logic operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, stated previously.
3. Every bitwise logic operation must produce output bit-blocks whose length is greater than or equal to the length of every input operand.

The restrictions specified for the shift and rotate operations are as follows:

1. Every shift or rotate operation must have two input operands and one output operand.
2. The operands of every shift or rotate operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, stated previously.
3. One of the input operands of a shift or rotate operation must be connected to a modeling element specifying an integer constant, and the other input operand

must specify the source bit-block. The integer constant indicates the number of bits the source operand is shifted or rotated.

4. Every shift or rotate operation must produce output bit-blocks whose length is greater than or equal to the length of every input operand.

#### 4.4.5 Dataflows

As stated previously, the dataflows connect nodes to each other, and allow objects to traverse from one operation to another. The stereotype **BBDataFlow** in the package Edges extends the meta-class **ObjectFlow** to adapt it to the application domain of interest. The concrete syntax of edges (a pointing arrow) defined by UML 2 does not require changes and is preserved in the proposed profile. Figures 4.10 and 4.12 illustrate several edges interconnecting nodes. Such instances of **ObjectFlow** have been extended by instances of **BBDataFlow**, but the extensions are not shown for clarity reasons.

The constraints defined for **BBDataFlow** are as follows:

1. Every dataflow must always allow bit-blocks to traverse through it. This is achieved by setting the guard condition associated to every dataflow to true.
2. Every dataflow must allow only one bit-block to traverse through it. This is achieved by setting the weight attribute of every dataflow to one.
3. Every dataflow must connect well-formed operands or parameters belonging to well-formed operations or modules. In addition, the interconnected nodes must process a continuous flow of bit-blocks of the same length.

#### 4.4.6 Invoking modules and switching between bit-blocks

Figure 4.10 illustrates a switch that selects one of several input bit-blocks and transfers it to the output. Every switch in a model has an associated state machine indicating the input to select depending on the current state of the algorithm. Figure 4.13 illustrates the state machine diagram associated to the switch in Figure 4.10. The current state of the algorithm determines the input selected by the switches, and the

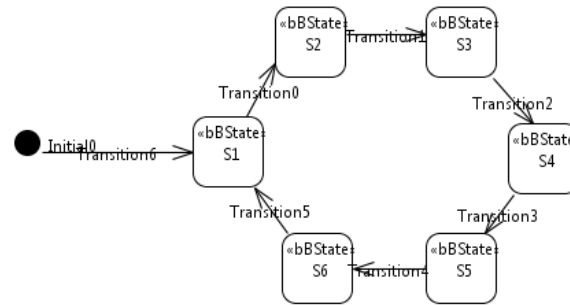


Figure 4.13: An extended state machine diagram in UML 2 associated to a switch.

operands received by the operations connected to the switches. Switches are useful to model iterative algorithms where the result of an operation is fed back to the inputs.

The stereotype **BBSwitch** in the package Controls extends the meta-class **Call-BehaviorAction** to provide the designer with a modeling element that selects one of multiple bit-blocks at the input operands based on the current state of the algorithm. A switch is actually an action that invokes the behavior described by the associated state machine. When the state of the algorithm changes, the switches invoke their state machines, which select the appropriate input depending on the current state and change to the next state. During transition to the next state, the algorithm carries out all of its operations using the values provided by the switches and computes the results.

Every calling action extended by an instance of **BBSwitch** must fulfill the following constraints:

1. The operands of every switch must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, stated previously.
2. The operands of every switch must correspond one-to-one to the parameters of the state machine invoked.
3. Every switch must be synchronous, which means that the the caller module waits for completion of the invoked state machine.

4. The state machine invoked by a switch must fulfill the restrictions specified for this modeling construct.
5. Every switch must be identified by a non-empty string different from the identifier assigned to the state machine invoked.

The profile allows modeling complex algorithms as long as two conditions are met. First, every model must manipulate bit-blocks and use supported operations. Second, a model may invoke other models or not; if it does, the invoked models must meet these two conditions. As a result, it is possible to build hierarchical models made up an arbitrary number of different models. Also, UML 2 provides modeling constructs that enable behaviors to invoke one another in a hierarchical manner; the only limitation may be the extent as to which the modeling tools ease sharing and reusing modeling projects and diagrams.

Figure 4.14 illustrates a module containing modeling elements that invoke other modules. The stereotype **BBModuleCall** extends the meta-class **CallBehaviorAction** to provide the designer with the ability to reuse modules and build hierarchical models. The stereotype defines constraints that adapt instances of **CallBehaviorAction** to the application domain of interest, but preserves the concrete syntax defined for these objects by UML 2, specially the rake symbol ( $\pitchfork$ ). The constraints specified for **BBModuleCall** are as follows:

1. The operands of every module call must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, stated previously.
2. The operands of every module call must correspond one-to-one to the parameters of the module invoked.
3. Every module call must be synchronous, which means that the the caller module waits for completion of the module called.
4. The module called by a call operation must fulfill the restrictions specified for this modeling construct.

5. Every module call must be identified by a non-empty string different from the identifier assigned to the module called.

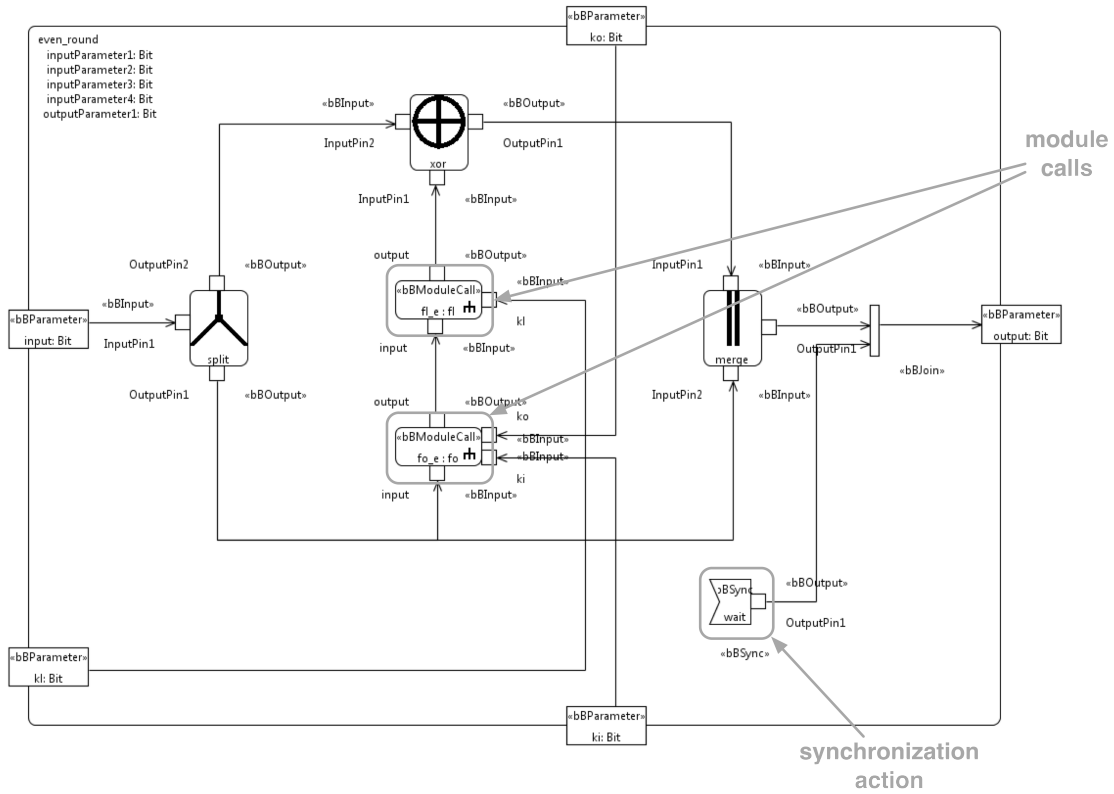


Figure 4.14: A module containing invocations to other modules.

## 4.5 Application of the profile

This section shows practical examples of the use of the proposed domain-specific modeling language to describe different algorithms. First, we show the application of the language to the description of a block cipher algorithm. Second, we show the application of the profile to the description of an algorithm to perform arithmetic in finite fields, which is used by coders and ciphers. The models presented were built using Papyrus, a modeling tool working on top of Eclipse [22].

### 4.5.1 Block ciphering

Consider the block cipher KASUMI used nowadays to implement security functions in modern 3G cellular communication networks [1]. The inputs to the algorithm are a 64-bit plaintext block and a 128-bit key  $K$ , and the output is a 64-bit ciphertext block. Each of the eight rounds of KASUMI's Feistel network carries out a pair of operations called FL and FO, where FO is a Feistel network with three rounds, each invoking a function called FI consisting of four substitution boxes (S9 and S7). The algorithm's key scheduler uses the input key  $K$  to generate eight sets of round keys ( $s_i = \{KL_i, KO_i, KI_i\}, i = 1, 2, \dots, 8$ ), one for every round in the main Feistel structure. The block diagrams in Figure 4.15 illustrate the structure of the components of KASUMI.

The diagrams in Figure 4.15 are neither models in UML 2, nor schematics of a hardware architecture implementing the algorithm. However, it is possible to build models that mimic the structure of KASUMI using UML 2 and the profile described in the previous section, since all of the operations performed by KASUMI are supported by the profile. Figure 4.16 illustrates the hierarchy of modules that make up the model of KASUMI, where each module contains parameters, operations, dataflows, switches, and invocations to other modules. It is possible to manipulate the structure of the model to build simplified descriptions of the algorithm that produce area-efficient hardware architectures.

The designers can modify the structure of KASUMI, without changing its behavior, to obtain implementations of the algorithm that have high performance or save hardware resources. Figure 4.17 illustrates the results of one of such manipulations of the algorithm: a compact structure that ciphers a block after 16 iterations over the same component [8]. The component in Figure 4.17(a) is the result of attaching two FI blocks (see Figure 4.15(c)) by means of sharing the four S-boxes. The component in Figure 4.17(b) is a simplification of the FO function that takes 2 iterations over the dual-input FI function to compute the same result as the Feistel structure shown in Figure 4.15(b). Finally, the component in Figure 4.17(c) computes the same result as the eight-round Feistel structure in Figure 4.15(a) after 16 iterations over the simplified FO function.



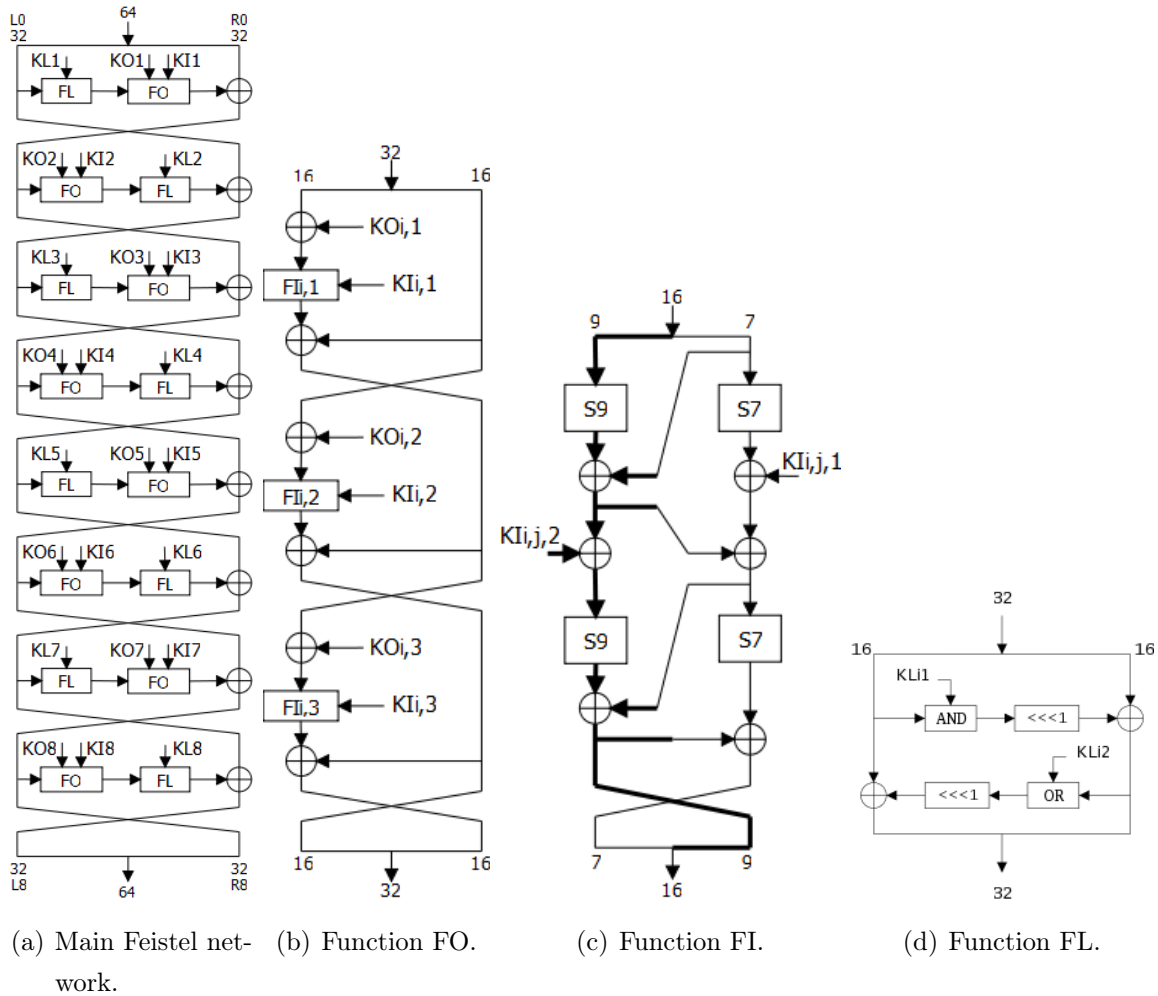
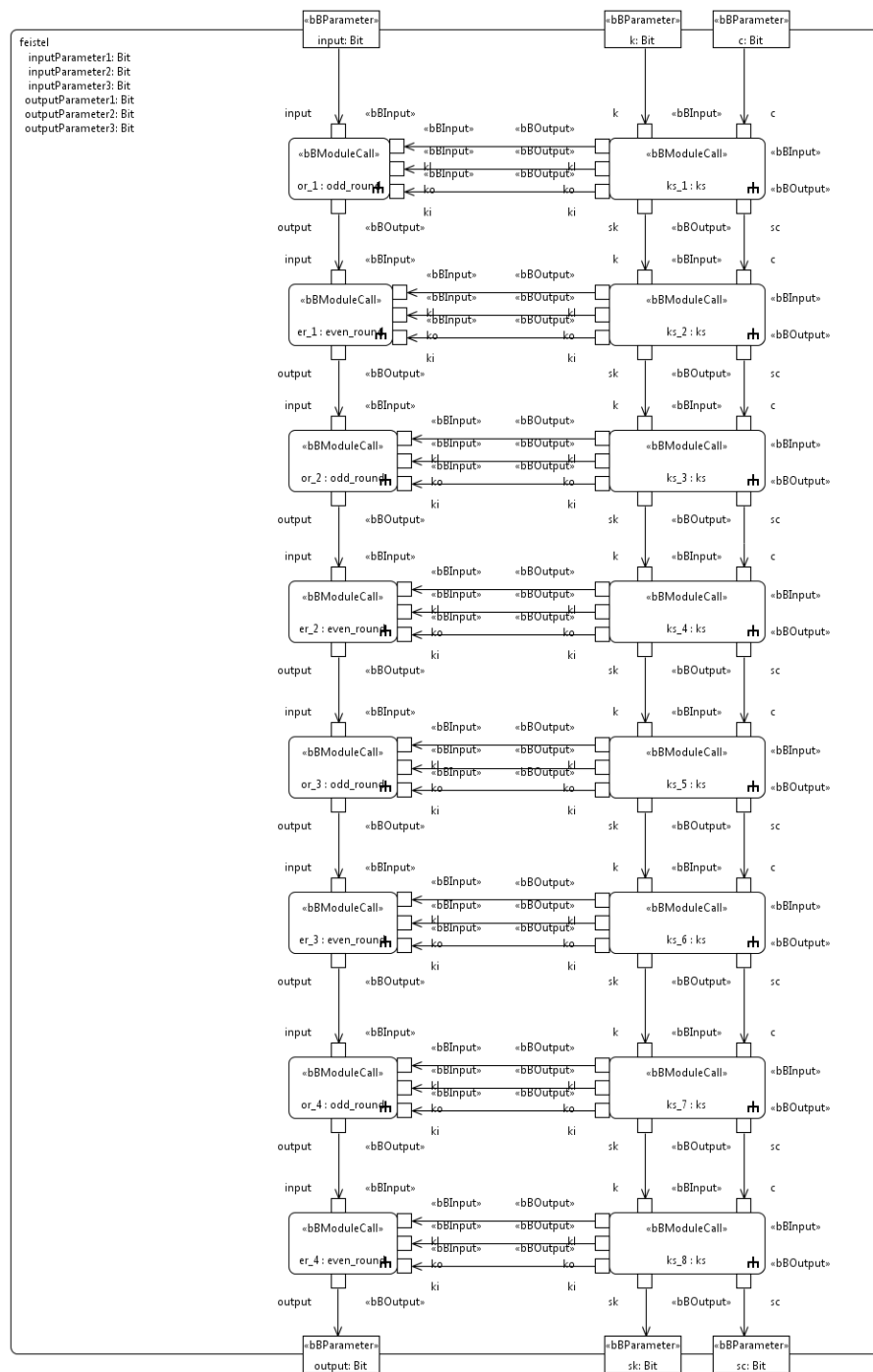


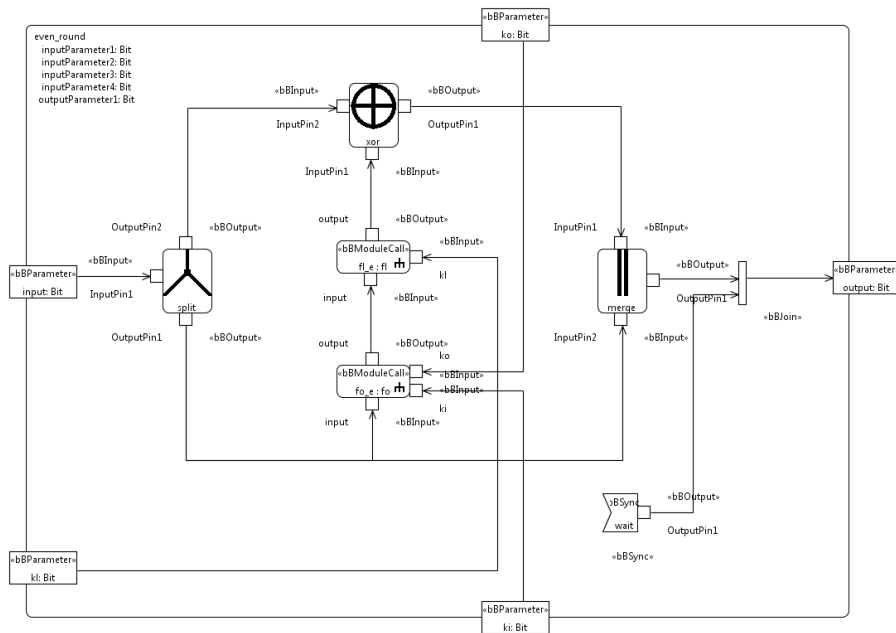
Figure 4.15: The components of the block cipher KASUMI (from [1]).

Figure 4.18 illustrates the model for the previous iterative design. The iterative modules require switches to indicate the feedback of intermediate results, and state machines that control the flow of information during every iteration. The result of implementing this model is a compact design with fewer hardware components than the implementation of the model in Figure 4.15, at the cost of sacrificing performance because of the overhead of 16 iterations to cipher a single 64-bit block. Therefore, the designers always face the challenge of making tradeoffs between high performance and compact implementations. This example illustrates that the proposed profile enables the designer to manipulate the model of an algorithm according to a design strategy,

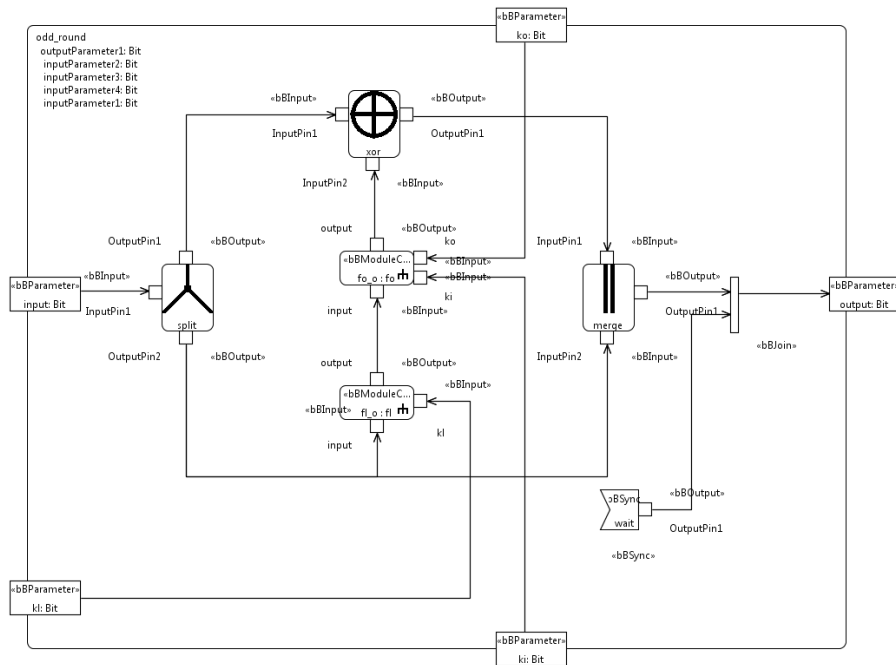


(a) Module for the main Feistel network.

Figure 4.16: The modules comprising the model of KASUMI in UML 2 and the profile BitBlockFlow.

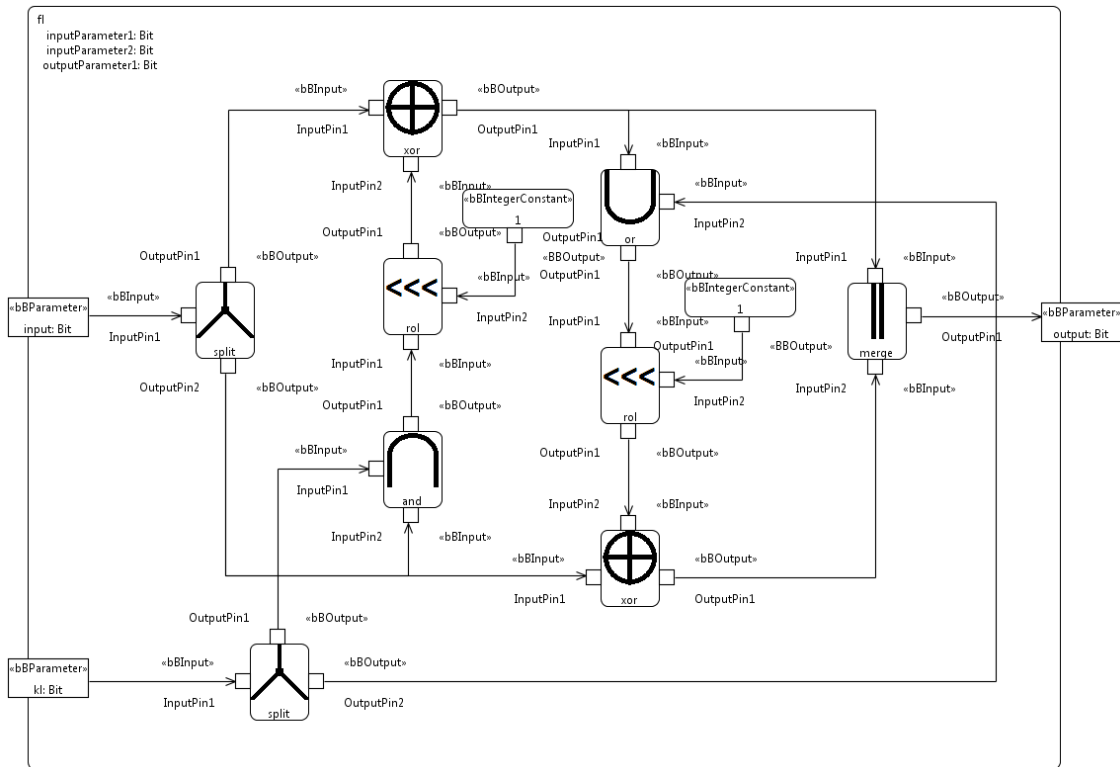


(b) Module for an even round.



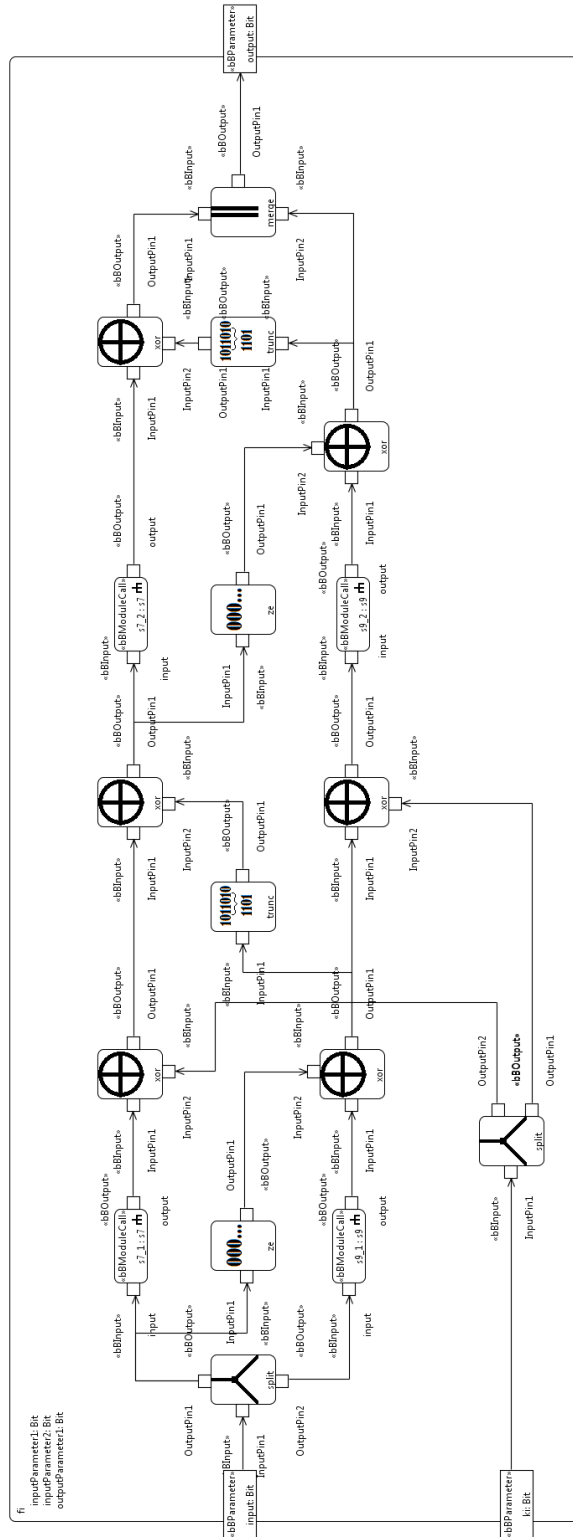
(c) Module for an odd round.

Figure 4.16: The modules comprising the model of KASUMI in UML 2 and the profile BitBlockFlow (cont.)



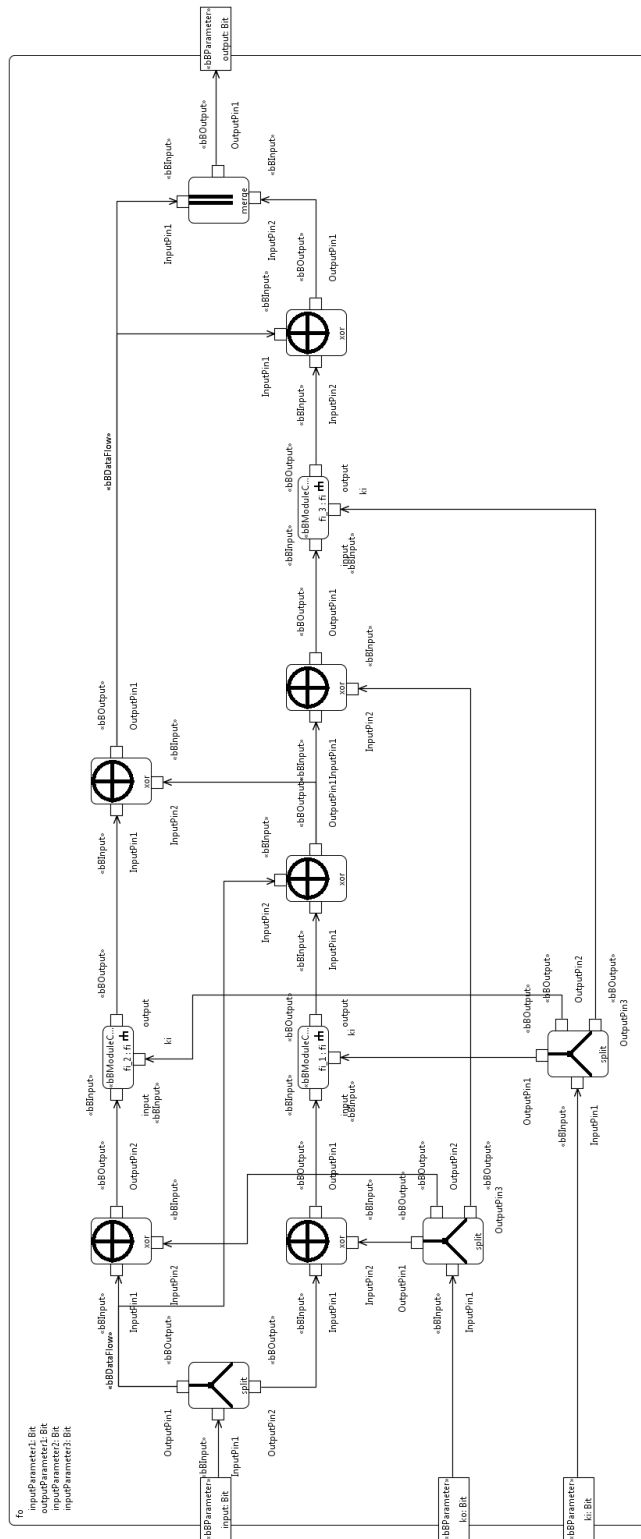
(d) Module for the function FL.

Figure 4.16: The modules comprising the model of KASUMI in UML 2 and the profile BitBlockFlow (cont.)



(e) Module for the function FI.

Figure 4.16: The modules comprising the model of KASUMI in UML 2 and the profile BitBlockFlow (cont.)



(f) Module for the function FO.

Figure 4.16: The modules comprising the model of KASUMI in UML 2 and the profile BitBlockFlow (cont.)

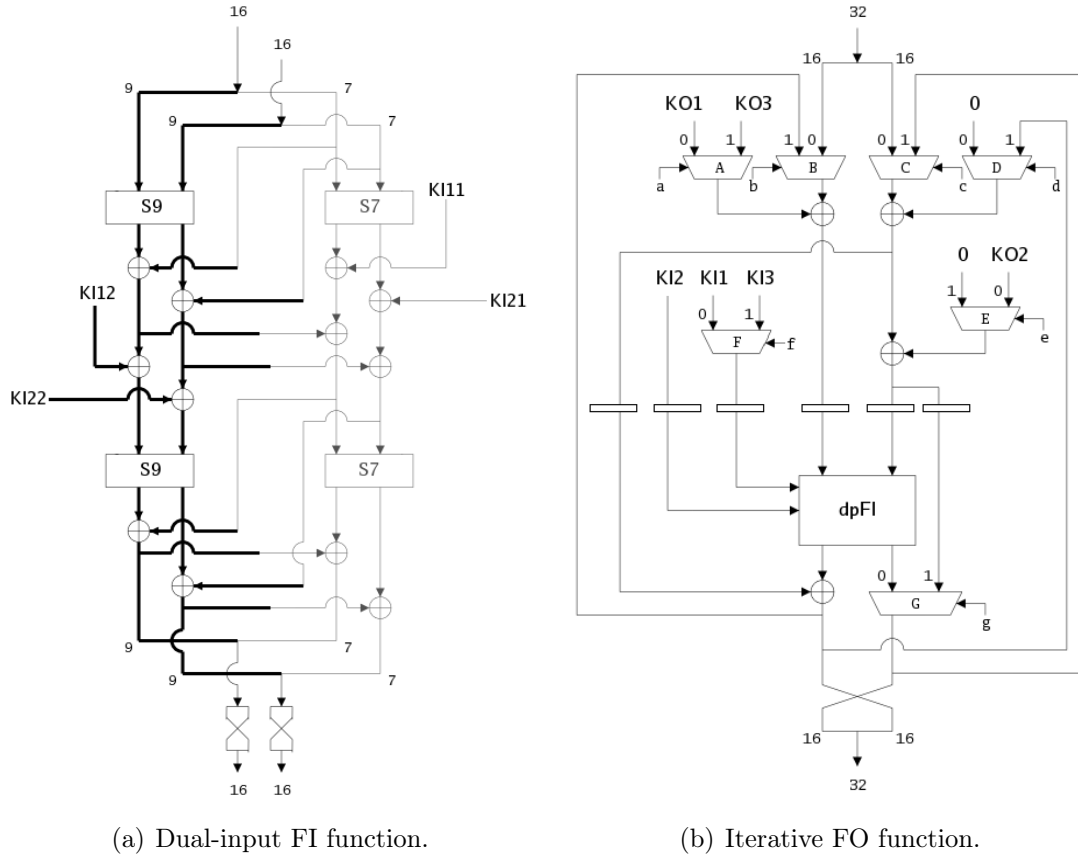


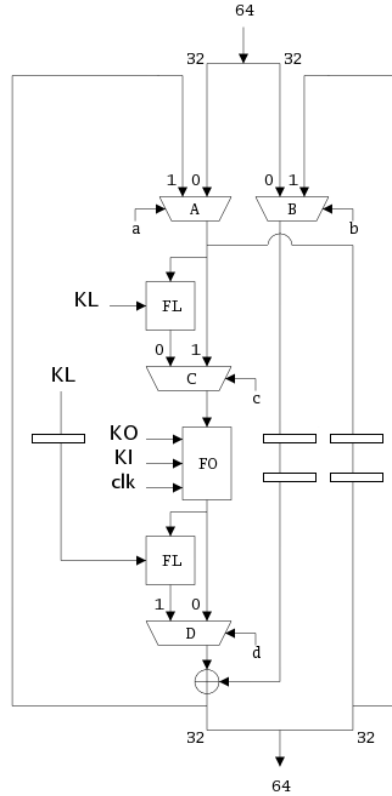
Figure 4.17: The components of the simplified block cipher KASUMI.

and provides the elements to build either iterative and loop-unrolled models.

### 4.5.2 Multiplication in finite fields

Modern algorithms that compute error-correcting codes and perform elliptic curve encryption require multiplication and addition of large integers belonging to a binary field. A binary field, or Galois field, contains a finite number of elements that can be expressed as  $p^k$ , where  $p$  is prime and  $k$  is a positive integer. A commonly used finite field is the binary field, denoted as  $GF(2^m)$ ,  $m > 0$ , that contains  $2^m$  integers represented by  $m$ -bit blocks that when multiplied and added produce an integer that also belongs to  $GF(2^m)$  [13]. Multiplication of large integers in the binary field is not a trivial task, and algorithms performing this operation fast are greatly appreciated.

This clause shows the model of an algorithm that performs fast multiplication of



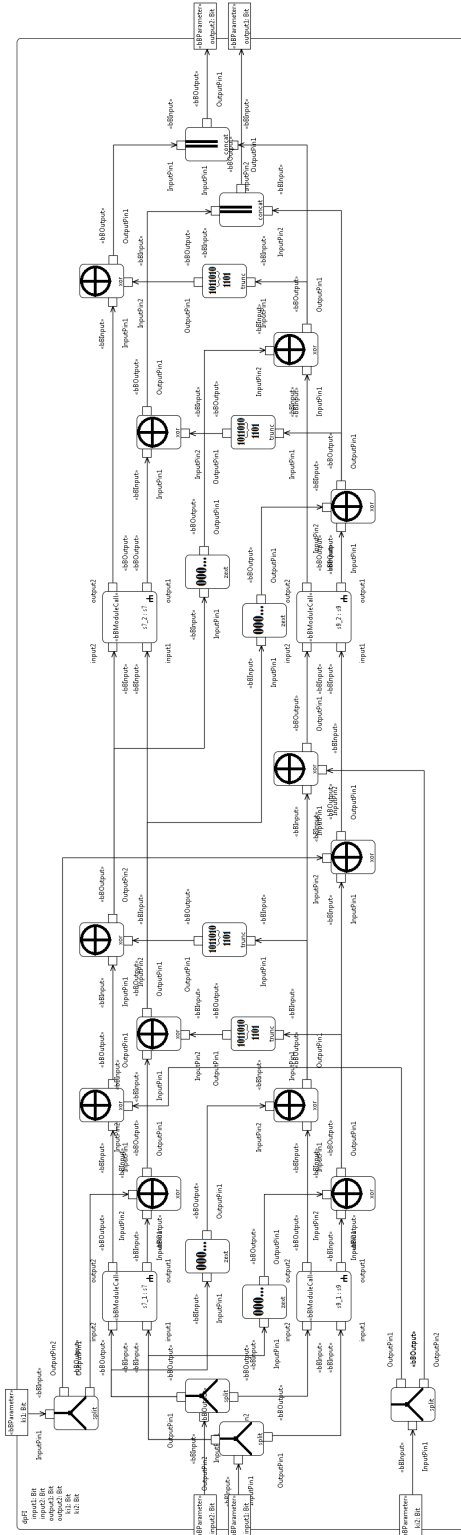
(c) Iterative round.

Figure 4.17: The components of the simplified block cipher KASUMI (cont.)

two 128-bit integers, and produces a 163-bit integer in  $GF(2^{163})$ . The operands can be thought of as 163-bit integers whose 35 most significant bits are set to zero, and their multiplier can be used by a multiplier operating on arbitrary 163-bit integers. The multiplier is based on the Karatsuba-Ofman algorithm (KOA) [29], and uses linear feedback shift registers (LFSRs) to map the result to the binary field  $GF(2^{163})$ . When the length in bits  $m$  of the integers is a power of two,  $m = 2^p, p \leq 0$ , the complexity of KOA is  $O(m^{\log_2 3})$ .

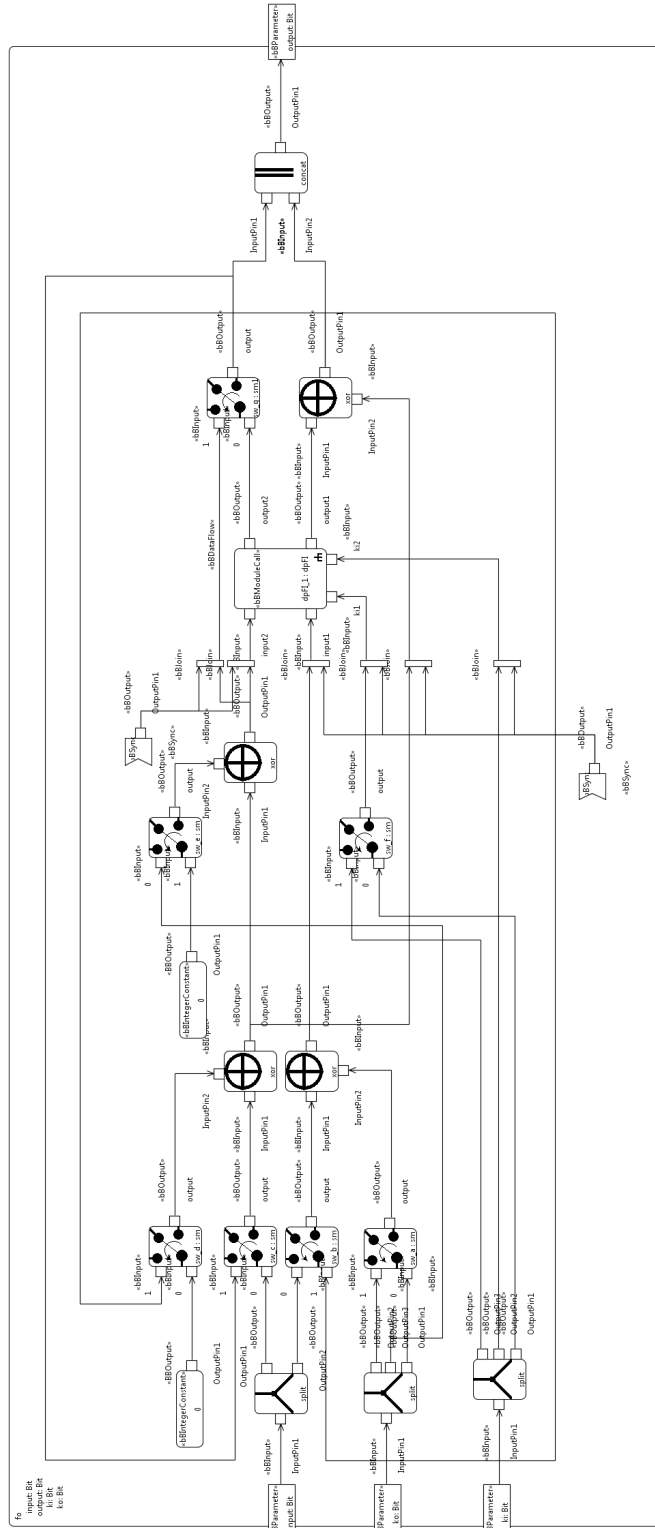
Figure 4.19 illustrates the hierarchical model of the KOA that multiplies two 128-bit integers proposed by Cuevas [16]. The highest module in the hierarchy, illustrated in Figure 4.19(a), splits the input 128-bit blocks into 64-bit blocks and uses them as inputs to a module that performs KOA for 64-bit integers, shown in Figure 4.19(b). The module describing KOA for 64-bit integers splits the input bit blocks into 32-bit





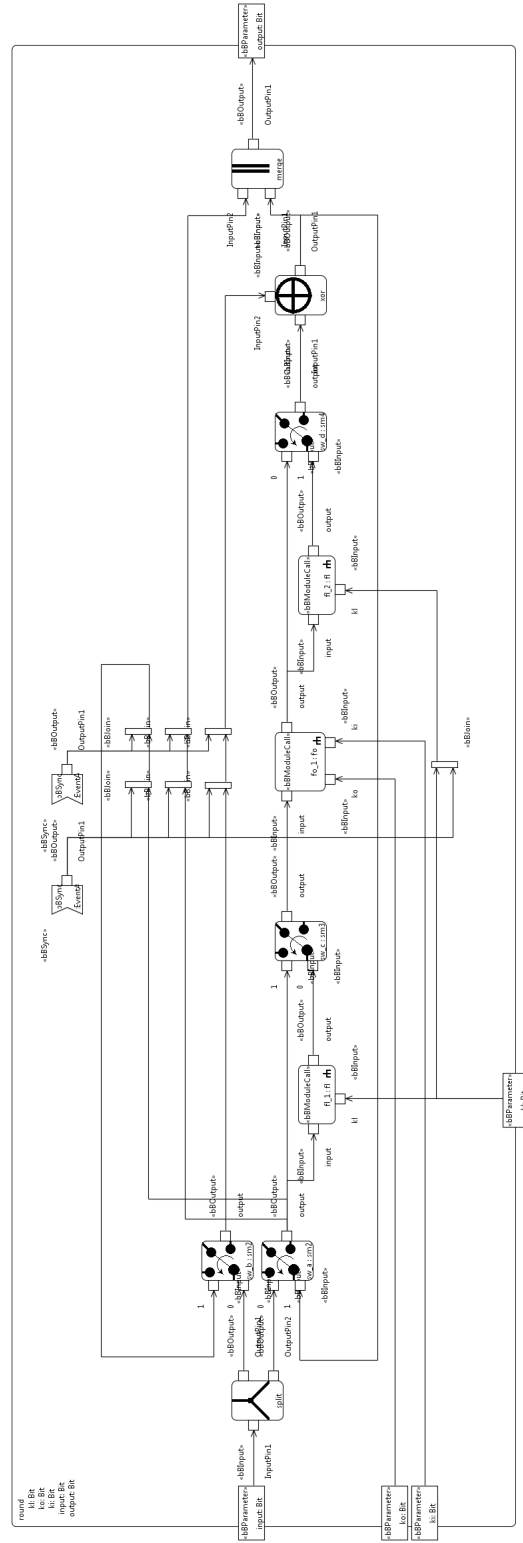
(a) Module for the dual-input FI function.

Figure 4.18: The modules comprising the model of the simplified KASUMI in UML 2 and the profile BitBlockFlow.



(b) Module for the iterative FO function.

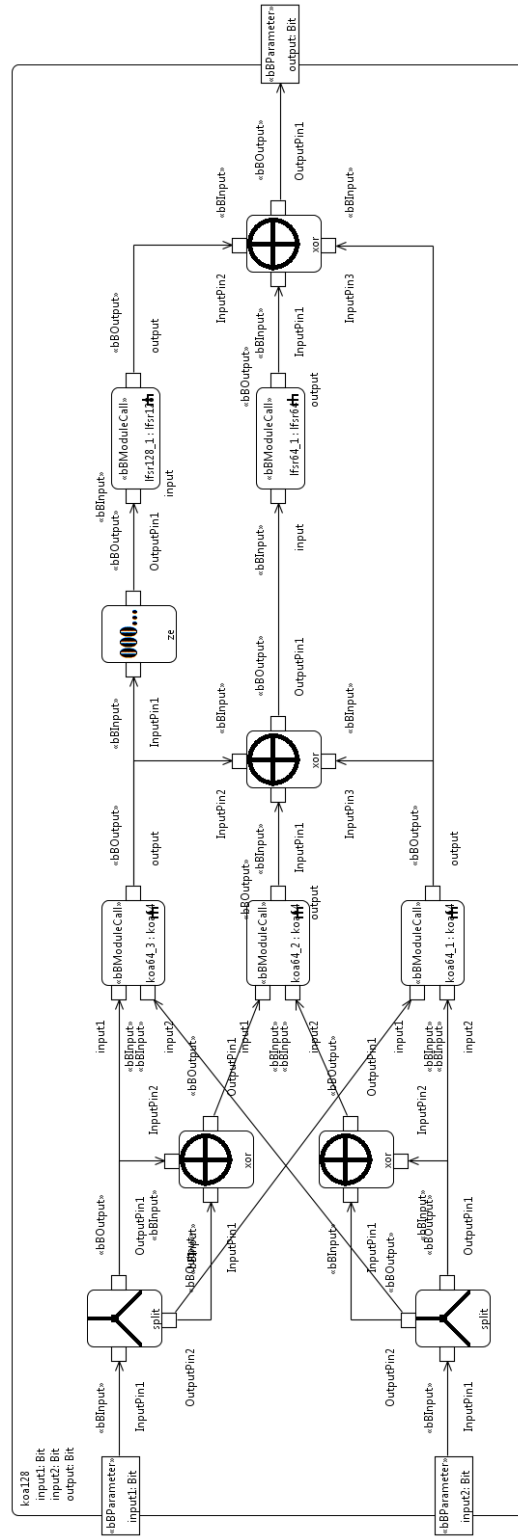
Figure 4.18: The modules comprising the model of the simplified KASUMI in UML 2 and the profile BitBlockFlow (cont.)



(c) Module for the iterative round.

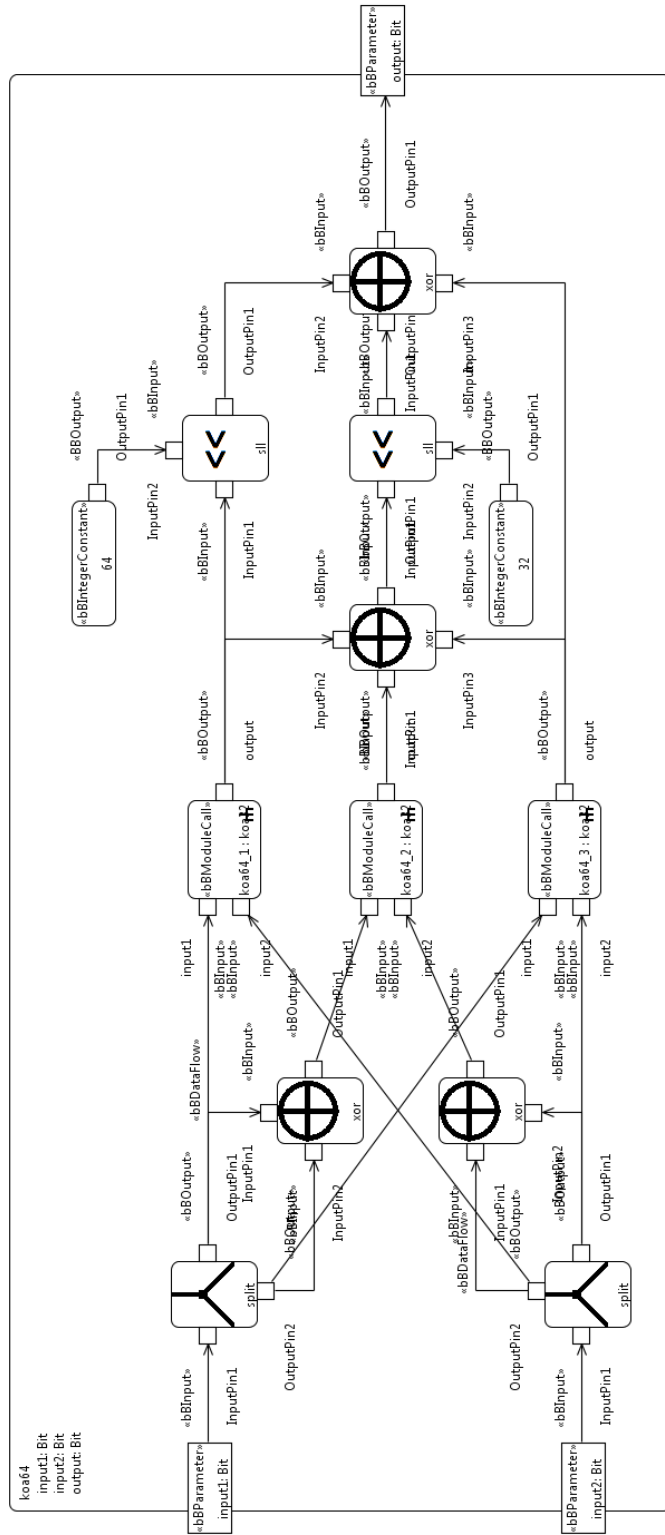
Figure 4.18: The modules comprising the model of the simplified KASUMI in UML 2 and the profile BitBlockFlow (cont.)

blocks and uses them as inputs to a module that performs KOA for 32-bit integers, shown in Figure 4.19(c). Thus, every KOA module splits its  $n$ -bit operands into  $n/2$ -bit blocks and calls a KOA module that multiplies integers of this length. The model contains modules that multiply integers of length 2, 4, 8, 16, 32, 64 and 128. Figures 4.19(d), 4.19(e) and 4.19(f) show the modules describing the linear feedback shift registers that map the result of the multiplication to an element in  $GF(2^{163})$ . These modules employ switches to describe loops and feed back intermediate results.



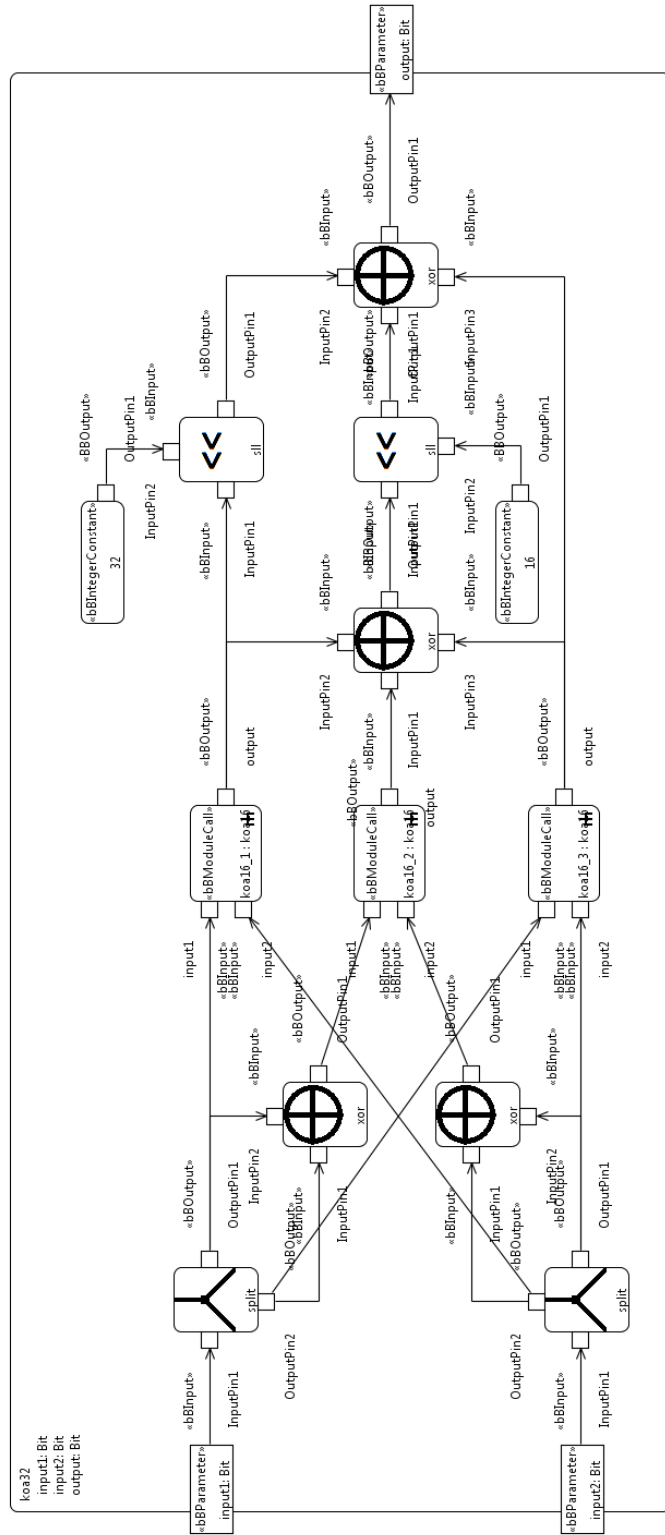
(a) Module for 128-bit KOA.

Figure 4.19: Modules performing multiplication of integers of different lengths according to the Karatsuba-Ofman algorithm.



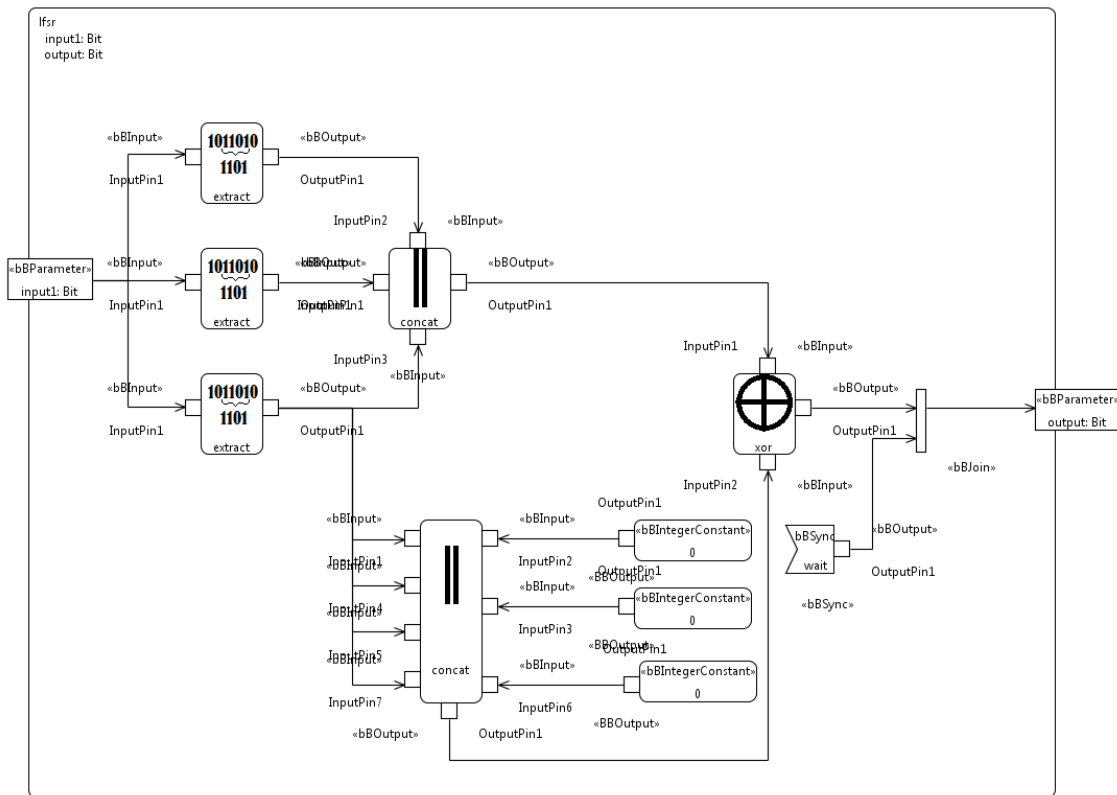
(b) Module for 64-bit KOA.

Figure 4.19: Modules performing multiplication of integers of different lengths according to the Karatsuba-Ofman algorithm (cont.)



(c) Module for 32-bit KOA.

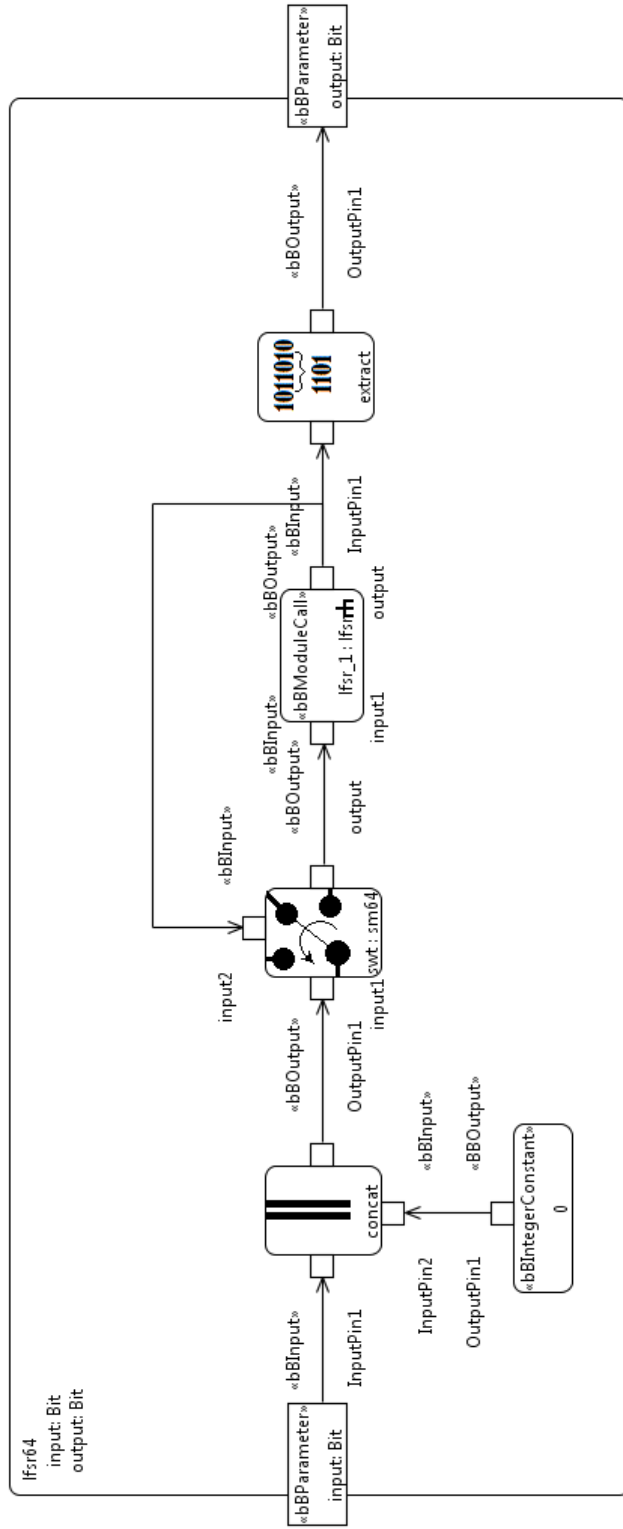
Figure 4.19: Modules performing multiplication of integers of different lengths according to the Karatsuba-Ofman algorithm (cont.)



(d) Module for the basic shift register.

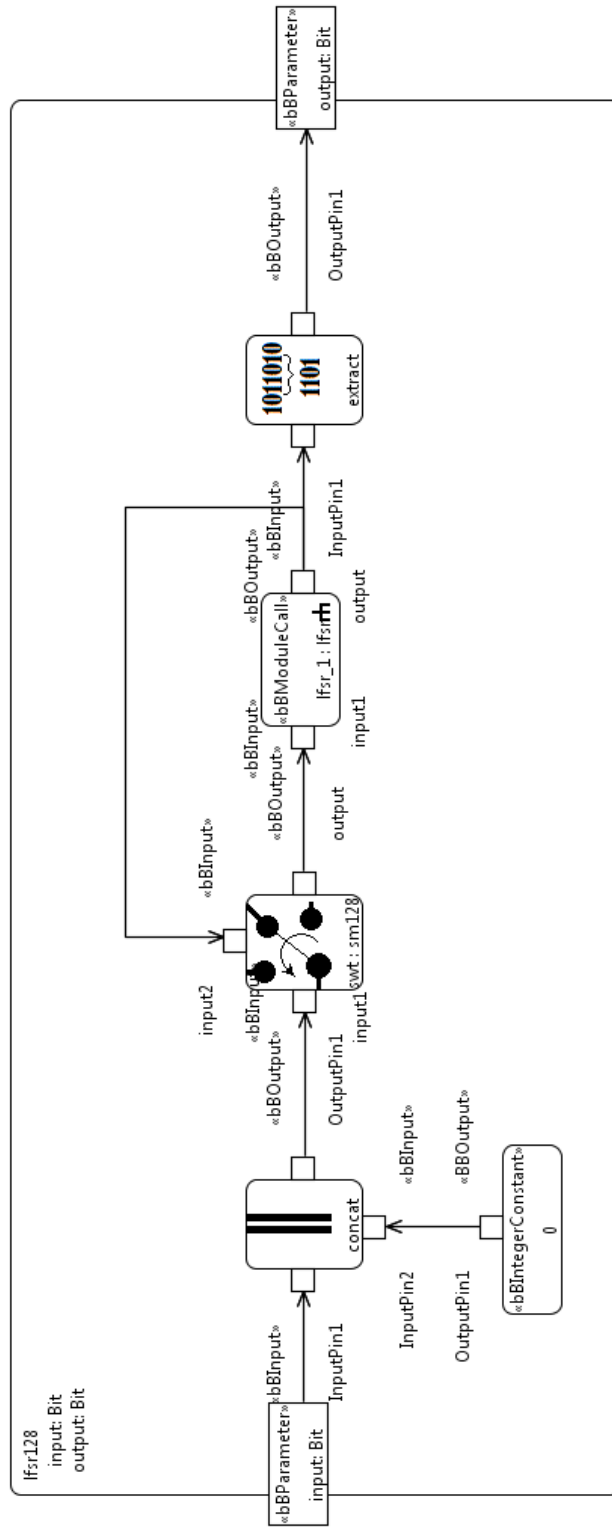
Figure 4.19: Modules performing multiplication of integers of different lengths according to the Karatsuba-Ofman algorithm (cont.)





(e) Module that iterates over the basic shift register 64 times.

Figure 4.19: Modules performing multiplication of integers of different lengths according to the Karatsuba-Ofman algorithm (cont.)



(f) Module that iterates over the basic shift register 128 times.

Figure 4.19: Modules performing multiplication of integers of different lengths according to the Karatsuba-Ofman algorithm (cont.)

# Chapter 5

## The code generator

This chapter describes the implementation of the transformation that generates code in VHDL from the domain-specific data-flow models described in the previous chapter. This transformation was developed using Acceleo [41], a technology built on top of Eclipse that implements the MOFM2T standard from the OMG. This standard is used to describe transformations from models described by meta-models based on MOF to text, including source code in an artificial language [49]. Every transformation specified using MOFM2T consists of a number of *modules* containing *templates* that generate the skeleton of the source code; the templates are filled out with values from properties of the modeling elements or other complex values computed by *queries*.

### 5.1 Overview of MOFM2T and Acceleo

This section presents a simple example that illustrates how to specify transformations using MOFM2T to be implemented in Acceleo. The goal is to generate source code in Java declaring the classes defined by the diagram in Figure 4.1. After a careful examination, we decided to organize this transformation according to the section of the grammar of Java that generates the declaration of classes, whose syntax diagrams are shown in Figure 5.1. The grammar of the target language is the best guide in the process of generating source code conforming to such language, but it must be simplified because not all of the language constructs are considered in this example.

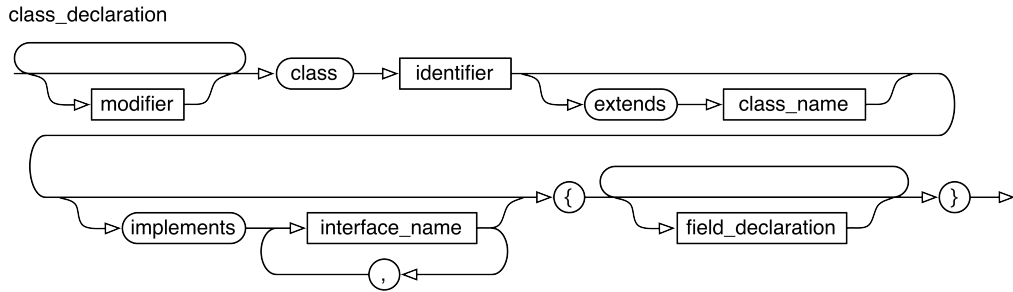
The production rule that starts the generation of the section declaring the class is shown in Figure 5.1(a), it invokes the rules that generate the name of the class, the modifiers of the class, the name of the super-class, the name of the interfaces implemented by the class, and the set of instance variables and methods. Figure 5.1(b) shows the rule that generates the symbols that modify a class, of which we will only consider those that indicate abstract and public classes. The rules that generate identifiers are not considered explicitly in this example because such identifiers are taken directly from the names of the classes in the diagram in Figure 4.1. Figure 5.1(c) shows the rule that generates the declaration of instance variables and methods by invoking the rules in Figure 5.1(d) and Figure 5.1(e), respectively. Again, some of the production rules shown in the diagrams are not explicitly considered during the implementation of the transformation because they generate elements that can be obtained directly from the modeling elements, or that are not considered in this example.

The templates that make up the transformation are written in the language specified by MOFM2T, a declarative language reminiscent of OCL. Each template implements one of the rules in Figure 5.1 by generating the terminal symbols indicated by the rule, and invoking the templates for the invoked rules. The template for the rule in Figure 5.1(a), called `generateClassDeclaration()`, operates on modeling elements that are instances of the metaclass **Class** in the meta-model of UML 2. This template, written in Aceleo, generates the symbols `class`, `{` and `}` in an output file, and invokes other templates that generate the rest of the language constructs involved in the declaration.

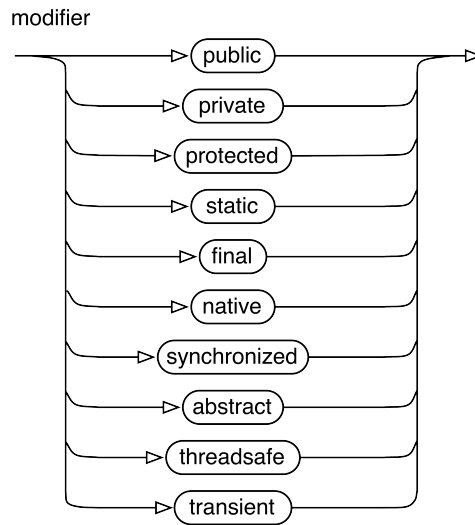
```

1. [template public generateClassDeclaration(aClass: Class)]
2. [aClass.generateModifier()/] class [aClass.generateIdentifier()/] [aClass.generateExtend()/] {
3.     [aClass.generateFieldDeclaration()/]
4. }
5. [/template]
```

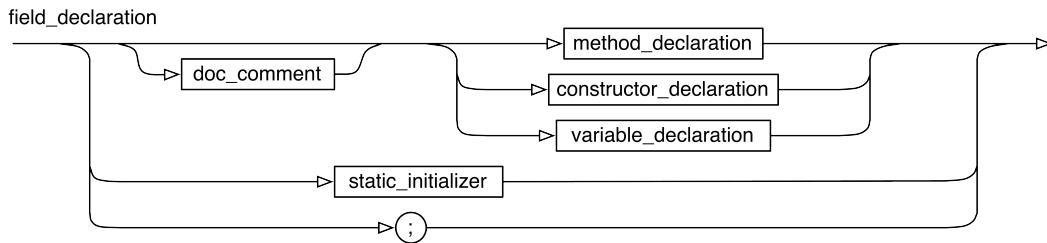
The template implementing the rule in Figure 5.1(b), called `generateModifier()`, generates only two of the ten modifiers for simplicity matters. This template checks the value of the property *isAbstract* in the instance of **Class** on which it is working to determine if such modeling element denotes an abstract class. When the modeling



(a) Syntax diagram for the section that declares a class.

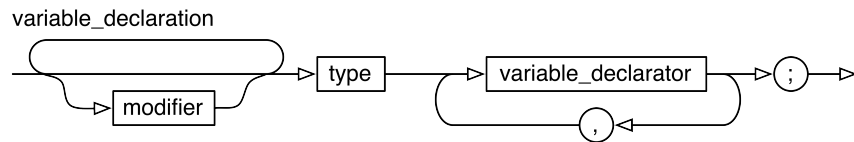


(b) Syntax diagram for the non-terminal symbol that modifies a field.

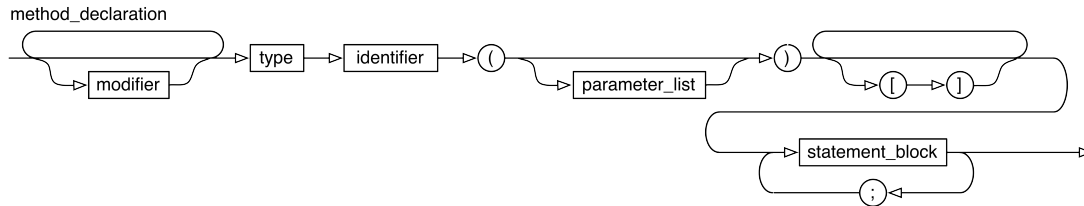


(c) Syntax diagram for the section that declares a field of a class.

Figure 5.1: Syntax diagrams for the declaration of a class in the Java language.



(d) Syntax diagram for the section that declares an instance variable of a class.



(e) Syntax diagram for the section that declares a method of a class.

Figure 5.1: Syntax diagrams for the declaration of a class in the Java language (cont.)

element indicates an abstract class, the template generates the modifier `abstract`, otherwise it generates the modifier `public`. The properties of the metaclass `Class` are documented in the specification of UML 2 [48].

```

1. [template public generateModifier(aClass: Class)]
2. [if (aClass.isAbstract)]
3. abstract
4. [else]
5. public
6. [/if]
7. [/template]

```

The template `generateExtend()` is invoked by `generateClassDeclaration()` to generate the symbol `extends` followed by the name of the super-class of the class being declared. This template uses the query `hasSingleSuperclass()` to validate that the class has one super-class only, and the query `getSuperclass()` to obtain a reference to the modeling element representing the super-class. The first query checks that the number of elements in the collection referred to by the property `generalization` equals one, which means that there is only one generalization relationship departing from the instance of `Class`. The second query first accesses the modeling element representing the generalization relationship, and then returns a reference to the modeling element representing the more general class in the relationship.

```

1. [template public generateExtend(aClass: Class)]
2. [if (aClass.hasSingleSuperclass())]
3. extends [aClass.getSuperclass().name/]
4. [/if]
5. [/template]
6.
7. [query public hasSingleSuperclass(aClass: Class): Boolean =
8.     aClass.generalization->size() = 1
9. /]
10.
11. [query public getSuperclass(aClass: Class): Class =
12.     aClass.generalization
13.     ->asSequence()->first().general.oclAsType(Class)
14. /]

```

The template `generateFieldDeclaration()` implements the rule in Figure 5.1(c) partially, because it only generates the declaration of the instance variables and methods. This template iterates over the collection of properties (instances of the metaclass **Property**), and invokes the template `generateVariableDeclaration()` for every one. The template also iterates over the collection of operations (instances of the metaclass **Operation**), and invokes the template `generateMethodDeclaration()` for every one. These templates correspond to the rules in Figure 5.1(d) and Figure 5.1(e), respectively, and generate the name, type and visibility of the features of the class by accessing the values of the corresponding properties in the instances of **Property** and **Operation**. Also, notice that `generateMethodDeclaration()` also invokes a template that generates an empty body for every method.

```

1. [template public generateFieldDeclaration(aClass: Class)]
2. [for (p: Property | aClass.ownedAttribute)]
3. [p.generateVariableDeclaration()/]
4. [/for]
5.
6. [for (o: Operation | aClass.ownedOperation)]
7. [o.generateMethodDeclaration()/]
8. [/for]
9. [/template]
10.
11. [template public generateVariableDeclaration(aProperty: Property)]
12. [aProperty.generateModifier()/] [aProperty.generateType()/] [aProperty.name/];
13. [/template]
14.
15. [template public generateModifier(aProperty: Property)]

```

```

16. [aProperty.visibility.toString()/]
17. [/template]
18.
19. [template public generateType(aProperty: Property)]
20. [aProperty.type.name/]
21. [/template]
22.
23. [template public generateMethodDeclaration(anOperation: Operation)]
24. [anOperation.generateModifier()/] [anOperation.generateType()/] [anOperation.name/]()
25. [anOperation.generateStatementBlock()/];
26. [/template]
27.
28. [template public generateModifier(anOperation: Operation)]
29. [anOperation.visibility.toString()/]
30. [/template]
31.
32. [template public generateType(anOperation: Operation)]
33. [anOperation.type.name/]
34. [/template]
35.
36. [template public generateStatementBlock(anOperation: Operation)]
37. {
38. }
39. [/template]

```

## 5.2 The simplified grammar of VHDL

Not all of the language constructs of VHDL are required to build a description that complies with activity models like those in figures 4.16 and 4.19. The flow of bit-blocks starting at the input parameters, going through the operations, and ending at the output parameters, can be described by concurrent statements only. These sentences modify the values of bit vectors transported by signals by means of operators and processes. Thus, the transformation tool targets only a subset of VHDL and generates structural descriptions.

The language constructs used by the descriptions in VHDL generated by the transformation tool from the corresponding models are the following:

1. The section that declares the interface (entity) of the design:
  - (a) Ports declaration.



2. Primitive data types:
  - (a) *bit*.
  - (b) *bit\_vector*.
3. The section that declares resources to use by the architecture of the design:
  - (a) Enumeration types.
  - (b) Signals.
4. Binary literals.
5. Range specifiers for bit-vectors or signals.
6. The section that contains concurrent statements.
7. Operators:
  - (a) Bitwise logic.
  - (b) Shift and rotate.
  - (c) Concatenation.
8. The statement that declares a process.
9. The case statement.

The modified grammar listed in Appendix C describes this subset of VHDL, it is a simplification of the full grammar documented in the specification of VHDL [7]. This modified grammar is the base of the transformation; the identifiers of its templates and the way they call each other closely follows the non-terminal symbols and the production rules of the simplified grammar.

### 5.3 The transformation to VHDL

This section describes the implementation of the transformation that generates source code in VHDL from well-formed activity diagrams complying to the abstract syntax

of UML 2 and the constraints defined by the profile BitBlockFlow. This descriptions is focused on the templates that generate source code; the queries that validate well-formedness of the model, introduced in Chapter 4, get evaluated before initiating the generation process. If the model meets the constrains imposed by the profile, the transformation generates a source code file for every module, which defines the two sections mandatory for every design in VHDL: the entity and the architecture.

### 5.3.1 Relevant templates and queries

This clause describes important templates and queries used throughout the transformation to generate language constructs that appear multiple times throughout the definition of the entity and architecture sections. The transformation consists of about 90 templates and 300 queries, which include queries that evaluate the constraints of the modeling elements, queries that retrieve information from the model, and queries computing relevant information.

#### 5.3.1.1 Literals and signals

Figure 5.2 illustrates an operation whose operands are, on the one hand, an integer constant and, on the other hand, the output of another operation received through a dataflow. To determine whether an operand is connected to a modeling element representing an integer literal, the transformation employs a query that traverses to the opposite node and test whether it is the output operand of an element extended by the stereotype **BBIntegerConstant**. If the opposite modeling element represents an integer constant, the query evaluates whether it meets the constraints defined by the stereotype.

```

1.  [comment    Determines whether the receiving node is connected to an integer constant
2.             through a valid dataflow. /]
3.  [query public isConnectedToInteger(aNode: ObjectNode): Boolean =
4.      if (aNode.getIncomingDataflow(1).source.owner.isBBIntegerConstant()) then
5.          aNode.getIncomingDataflow(1).source.owner
6.          .oclAsType(ValueSpecificationAction).validateIntegerConstant()
7.      else
8.          false
9.      endif
10. /]
```

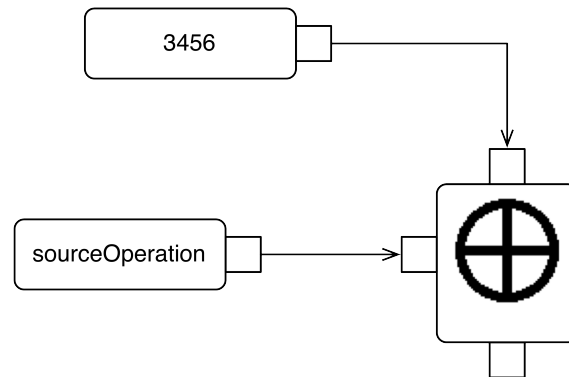


Figure 5.2: An operation whose operands are the output of an operation and an integer literal.

### 5.3.1.2 Reduction of labels of dataflows

Figure 5.3(a) illustrates that the output operand of *sourceOperation* is connected to multiple input operands through dataflows that have their own identifier. The transformation must select only one of these dataflows, map it to a signal in VHDL, and generate its identifier. Additionally, every time the output of *sourceOperation* is required, the transformation must select the same outgoing dataflow. The following template generates the same signal identifier whenever it is invoked on a given node:

```

1.  [comment    Generates the identifier of a signal or a binary literal depending
2.             on whether the receiving node is connected to a constant or a operation.
3.  /]
4.  [template public generateSourceSignalIdentifier(aNode: ObjectNode)]
5.  [if (aNode.isConnectedToInteger())]
6.  [aNode.generateSourceBitVector()/][else]
7.  [aNode.getIncomingDataflow(1).source.getOutgoingDataflow(1).generateSignalIdentifier()/][if]
8.  [/template]

```

The order of the elements in the collection of outgoing edges of a node may be different from an execution of the transformation to another. However, the order of these elements is the same throughout an execution of the transformation; that is why the previous template generates the same signal identifier when invoked on the input operand of any of the target operations. The use of this template discards the

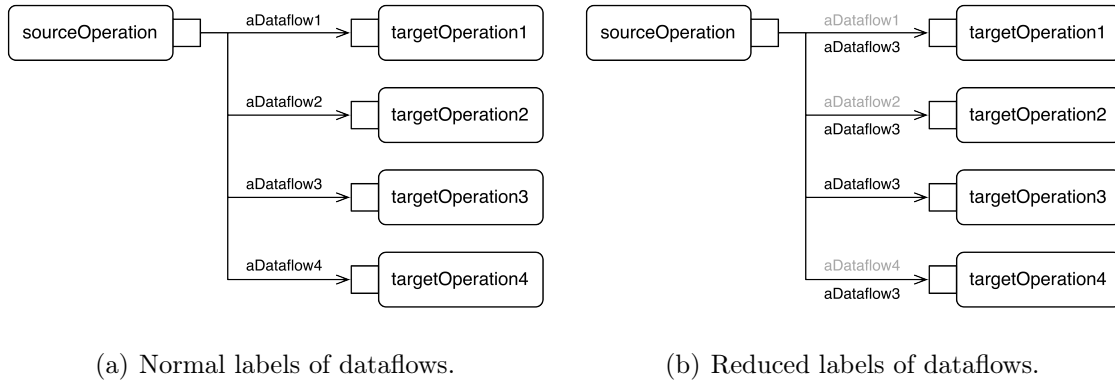


Figure 5.3: Reduction of labels of dataflows.

identifiers of other dataflows departing from the output operand of *sourceOperation*, as illustrated in Figure 5.3(b).

### 5.3.1.3 Recursive generation of expressions

A bitwise logic operation may receive multiple input operands, either integer literals or bit-blocks traversing dataflows. The transformation maps this modeling element to an assignment statement indicating the source signals, the operator and the target signal. Since OCL lacks complex control structures, the generation of the assignment statement requires a template that traverses the sequence of input operands recursively until the end of the sequence. At each call, the template generates a binary literal or the identifier of a signal, depending on whether the first operand of the sequence is connected to an integer literal or to the output operand of another operation; then, the template invokes itself recursively with the remains of the list as argument.

```

1. [template public generateExpression(aSequence: Sequence(InputPin),
2.                                   anOperator: String)]
3. [let firstElement: InputPin = aSequence->first()]
4. [if (aSequence->size() = 1)]
5. [firstElement.generateSourceSignalIdentifier()/]
6. [else]
7. [firstElement.generateSourceSignalIdentifier()/]
8. [anOperator/]
9. [aSequence->excluding(firstElement)->generateExpression(anOperator)/]
10. [/if]

```

- 11. [/let]
- 12. [/template]

### 5.3.2 Generating the entity declaration of the design

According to Bailey et al., “the design entity is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function” [7]. The entity declaration section in a design file defines the interface between the entity and the environment in which it is used. VHDL supports the declaration of multiple elements in the entity declaration section of a design file. However, this project only considers the definition of input, output and input/output ports, which is generated by the following production rules from the simplified grammar of VHDL:

```

<entity_declaration> ::= 'entity' <identifier> 'is'
                        <port_clause>
                        'begin'
                        'end' 'entity' <identifier> ';'

<port_clause> ::= 'port' '(' <port_list> ')' ';'

<port_list> ::= <interface_signal_declaration>
               { ';' <interface_signal_declaration> }

<interface_signal_declaration> ::= <identifier_list> ':' <mode> <subtype_indication>

<identifier_list> ::= <identifier> { ',' <identifier> }

<mode> ::= 'in'
          | 'out'
          | 'inout'

<subtype_indication> ::= <simple_name>
                       | <slice_name>

<simple_name> ::= <identifier>

<slice_name> ::= <simple_name> '(' <range> ')'
```

```

<range>                ::= <integer> <direction> <integer>

<direction>            ::= 'to'
                        | 'downto'

```

The transformation invokes the templates implementing the production rules on every module in the source model to generate the source code declaring the entity and its list of ports. The templates generating the list of ports retrieve the sequence of activity parameter nodes, extended by instances of the stereotype **BBParameter**, of the current module, and maps the nodes to statements declaring the corresponding ports. However, there are two ports that can be included in the list that do not have a matching parameter in the module: the port for the clock signal and the port for the reset signal. The transformation includes the port for the clock signal whenever the output of the current module, or the output of one of the modules it invokes, is synchronized with an external event, or when the current module, or one of the module it invokes, contains a switch. The transformation includes the port for the reset signal whenever the current module, or one of the module it invokes, contains a switch.

The templates below implement closely the production rules stated previously. The first template, `generateEntityDeclaration()`, generates directly five terminal symbols that make up the source code (“entity”, “is”, “begin”, “end”, “entity” and “;”), and generates indirectly other symbols by calling other templates. The second template, `generatePortClause()`, generates two more terminal symbols (“port”, “(” and “)”), and generates indirectly the list of ports by calling `generatePortList()`. The last template traverses the collection of parameters of the module and generates the terminal symbols that make up the declarations of the ports, according to the production rules in the previous grammar. The template `generatePortList()` invokes other templates that retrieve values from the parameters of the module and transfer them to the source code.

```

1.  [template public generateEntityDeclaration(aModule: Activity)]
2.  entity [aModule.generateEntityIdentifier()/] is
3.      [aModule.generatePortClause()/]
4.  begin
5.  end entity [aModule.generateEntityIdentifier()/];

```

```

6.  [/template]
7.
8.  [template public generatePortClause(aModule: Activity)]
9.  port (
10.     [aModule.generatePortList()/]
11.  );
12. [/template]
13.
14. [template public generatePortList(aModule: Activity)]
15. [for (pn: ActivityParameterNode | aModule.getParameters()) separator('; \n')]
16. [pn.generatePortIdentifier()/]: [pn.generateMode()/] [pn.generateSubtypeIndication()/]
17. [/for]
18. [if (aModule.isSynchronized() or aModule.isReseted());]
19.  clk: in bit
20. [if (aModule.isReseted());]
21.  reset: in bit
22. [/if]
23. [/if]
24. [/template]

```

### 5.3.3 Generating the architecture body of the design

According to Bailey et al., “the architecture body specifies the relationships between the inputs and outputs of a design entity and may be expressed in terms of structure, dataflow, or behavior” [7]. Thus, the architecture body section in the design file describes the internals of the entity whose interface was defined in the entity declaration section. The transformation generates architecture bodies describing the internal structure of the entity, and the flow of data between the components of such structure. The architecture body also declares a number of resources used by the statements that describe the flow of data within the entity.

The resources defined in the architecture declarative part of a design file generated by the transformation include: signals transporting information, enumeration types defining literals to denote different states of a module, and the interface of the entities used by the architecture. An architecture body describing a state machine requires the declaration of an enumeration type and some signals, but does not require declaring components. The architecture bodies in the design files for modules in models like those in figures 4.16 and 4.19 do require the declaration of components, and a number of signals.

As an example, consider the templates and queries that generate the declarations of the entities used by the architecture body for the module in progress. Although this module may invoke many other modules many times, the corresponding architecture declarative part needs only one reference to each of the invoked modules. Thus, the main template must retrieve a set of invoked modules, not a collection of invoked modules. For each element in this set, the main template generates terminal symbols and calls the template that generates the list of ports illustrated above. The following snapshot contains the template that generates the declarations and the query that retrieves the set of modules.

```

1.  [template public generateComponentDeclaration(aModule: Activity)]
2.  [for (a: Activity | aModule.getCalledModules()) separator('\n')]
3.  component [a.generateEntityIdentifier()] is
4.      [a.generatePortClause()]
5.  end component;
6.  [/for]
7.  [/template]
8.
9.  [comment    Returns a set of modules that are invoked by the calling operations in
10.           the receiving module.  /]
11. [query public getCalledModules(aModule: Activity): Set(Activity) =
12.     aModule.getModuleCalls()
13.     ->collect(cba: CallBehaviorAction | cba.getModule())
14.     ->asSet()
15.  /]

```

The language constructs in the architecture statement part of a design file include: the instantiation of the components whose interface was declared in the architecture declarative part, processes describing multiplexers or state machines, and concurrent statements that compute and assign values to signals and ports. An architecture body describing a state machine requires a process containing a sequential case statement to switch between states, and a number of concurrent assignment statements. The architecture bodies in the design files for other modules may require instantiating the entities for other components, using processes describing multiplexers, and a number of concurrent assignment statements.

As an example, consider the templates that generate a process statements with a case statement for every switch in the module in progress. These process statements describe the behavior of multiplexers according to the rules suggested by the



Xilinx's Synthesis and Simulation Design Guide [26]. Every process statement uses a case statement to select one of the input signals and assign its value to the output signal, depending on the value of a selector signal, whose value is driven by the entity corresponding to the state machine associated to the switch. Thus, when a module uses a switch, the transformation generates an entity for the state machine associated to the switch, declares and instantiates such entity in the architecture of the module containing the switch, and defines the process statement describing the multiplexer.

```

1.  [template public generateProcessStatement(aModule: Activity)]
2.  [for (cba: CallBehaviorAction | aModule.getSwitchOperations()) separator('\n')]
3.  [cba.getName().concat('_mux_process')/]: process([cba.generateSensitivityList()/]) is
4.  begin
5.      [cba.generateCaseStatement()/]
6.  end process [cba.getName().concat('_mux_process')/];
7.  [/for]
8.  [/template]
9.
10. [template public generateSensitivityList(aSwitch: CallBehaviorAction)]
11. [for (ip: InputPin | aSwitch.getInputs()) separator(', ') after(', ')]
12. [ip.generateSourceSignalIdentifier()/]
13. [/for]
14. [aSwitch.getStateMachine().generateSignalIdentifier()/]
15. [/template]
16.
17. [template public generateCaseStatement(aSwitch: CallBehaviorAction)]
18. case [aSwitch.getStateMachine().generateSignalIdentifier()/] is
19.     [aSwitch.generateCaseStatementAlternative()/]
20. end case;
21. [/template]
22.
23. [template public generateCaseStatementAlternative(aSwitch: CallBehaviorAction)]
24. [for (p: Parameter | aSwitch.getStateMachine().getInputParameters())]
25. when [aSwitch.getStateMachine().generateWaveform(aSwitch.getStateMachine().getIndex(p) - 1)/]
26.     => [aSwitch.getOutput(1).generateTargetSignalIdentifier()/] <=
27.         [p.getMatchingOperand(aSwitch).generateSourceSignalIdentifier()/];
28. [/for]
29. when others
30.     => [aSwitch.getOutput(1).generateTargetSignalIdentifier()/] <=
31.         [aSwitch.getInputs()->last().generateSourceSignalIdentifier()/];
32. [/template]

```

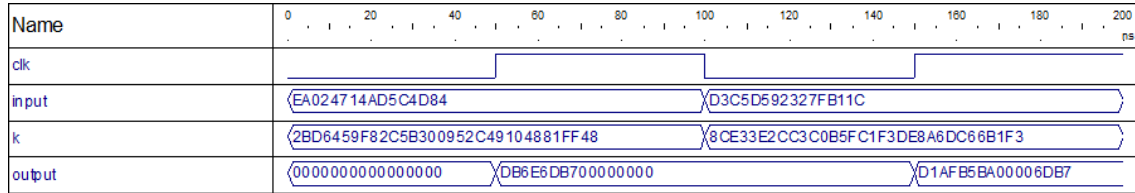
## 5.4 Results of simulation

Figure 5.4 illustrates the results of simulating the descriptions in VHDL corresponding to the modules in Figure 4.16. The main entity in the description corresponds to the module in Figure 4.16(a), and has a pipelined architecture where every stage computes one segment of the key scheduler and one round of the Feistel network. The functional simulation of the description in VHDL for KASUMI uses the standard test vectors provided by the Third Generation Partnership Program (3GPP), which exercise all of the components of the architecture [2].

The pipelined design requires eight clock cycles to encrypt the first 64-bit plaintext block and then issues one ciphertext block every clock cycle. During the first clock cycle (Figure 5.4(a)) the architecture is fed with the first plaintext block and the first key, during the second and third clock cycles (Figure 5.4(a) and Figure 5.4(b)) new data is provided to the architecture. From the eighth clock cycle to the tenth clock cycle (Figure 5.4(d) and Figure 5.4(e)) the architecture generates the resulting ciphertext blocks. These results prove that the transformation tool generated correct hardware descriptions in VHDL from the models in Figure 4.16.

Figure 5.5 illustrates the waveforms resulting of simulating the descriptions in VHDL corresponding to the modules in Figure 4.18. As mentioned in the previous chapter, the compact model of KASUMI describes an iterative variant of the algorithm that takes 16 iterations to cipher an input block. The corresponding implementation in VHDL takes one clock cycle to complete one iteration; thus, it has a latency of 16 clock cycles per block. As a consequence, this implementation does receive new input blocks until it has finished processing the current one. Since this implementation takes two clock cycles to complete one round, it must receive a new set of round keys ( $\{KL_i, KO_i, KI_i\}, i = 1, 2, \dots, 8$ ) every two clock cycles. Finally, notice in figures 5.5(a) to 5.5(h) that the implementation produces intermediate results, which are fed back to the input, until the final result is produced at the sixteenth clock cycle.

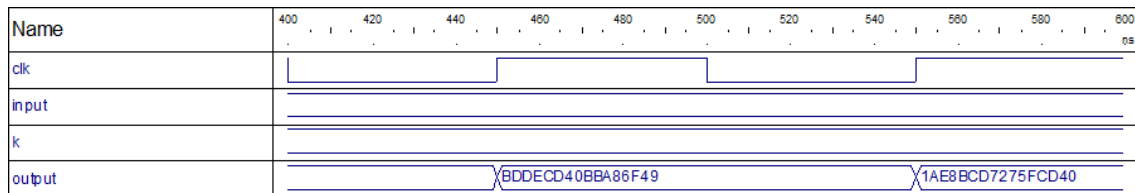
Let us consider the model for 128-bit KOA in Figure 4.19, whose main module is illustrated in Figure 4.19(a). The module in Figure 4.19(a) invokes the modules in figures 4.19(e) and 4.19(f), each iterating over the module representing a shift register shown in Figure 4.19(d). The LFSR in Figure 4.19(e) iterates over the basic



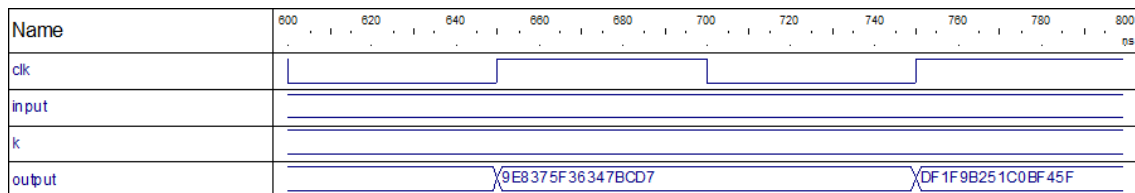
(a)



(b)



(c)

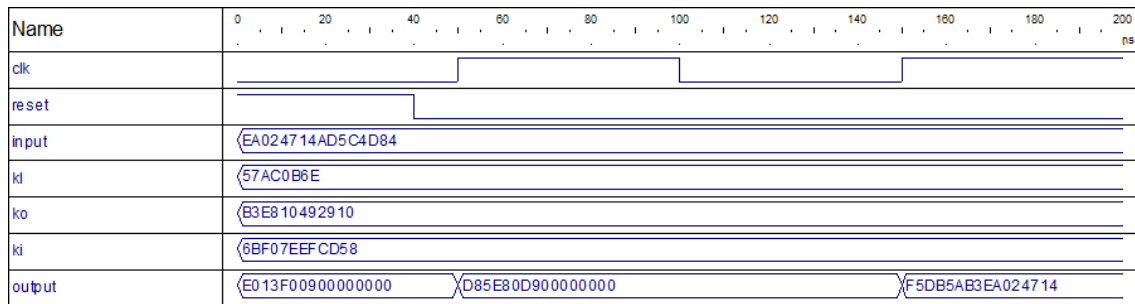


(d)

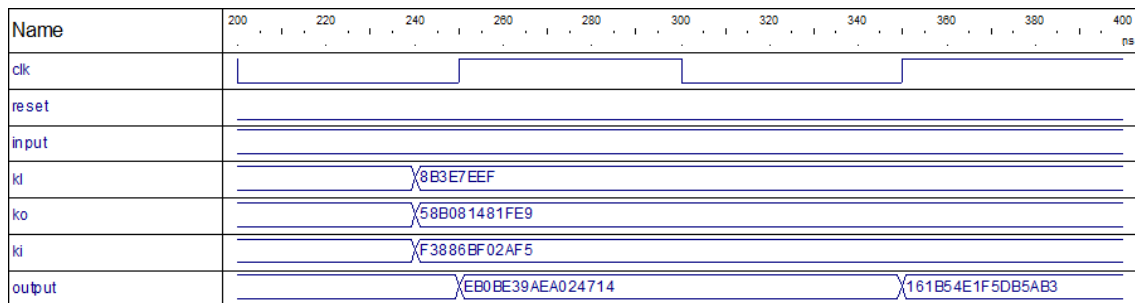


(e)

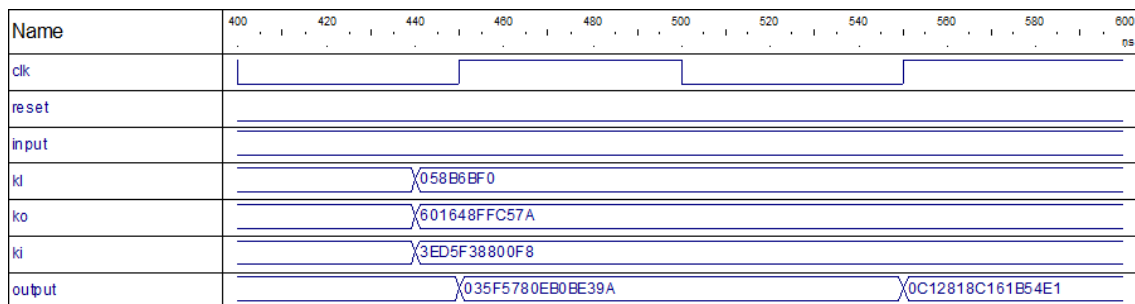
Figure 5.4: Results of the simulation of the description in VHDL implementing KA-SUMI.



(a)

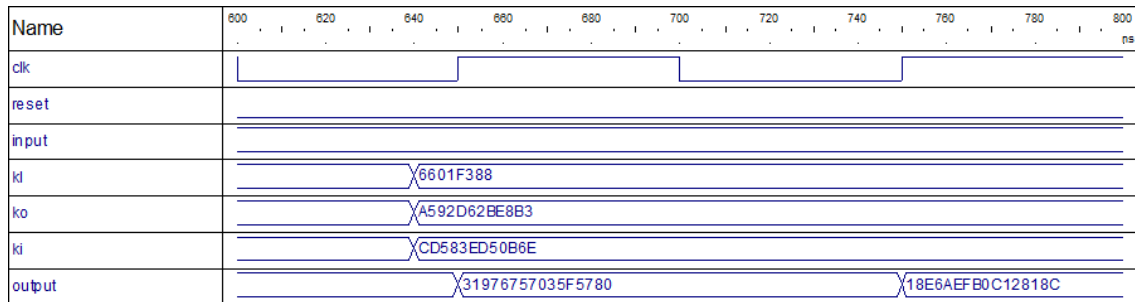


(b)



(c)

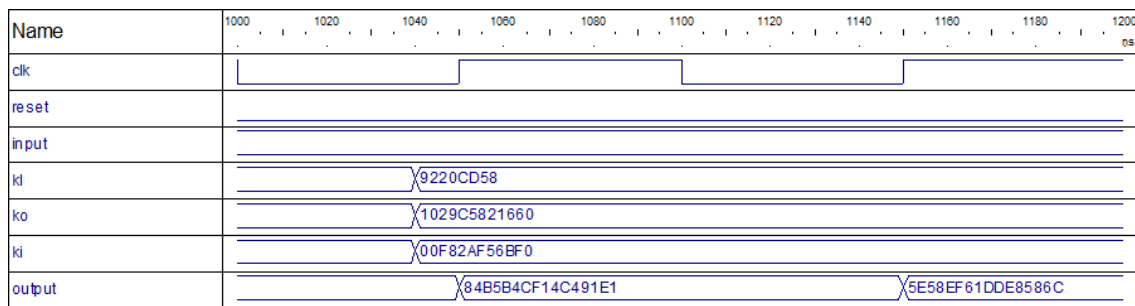
Figure 5.5: Results of the simulation of the description in VHDL implementing the simplified KASUMI.



(d)



(e)

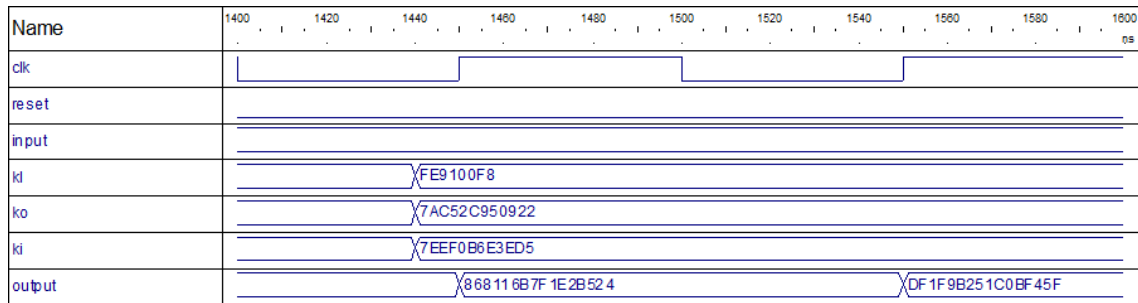


(f)

Figure 5.5: Results of the simulation of the description in VHDL implementing the simplified KASUMI (cont.)



(g)



(h)

Figure 5.5: Results of the simulation of the description in VHDL implementing the simplified KASUMI (cont.)

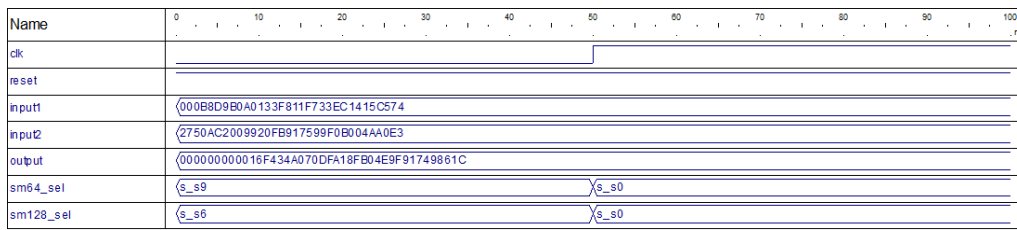
shift register 64 times, and the LFSR in Figure 4.19(f) iterates over the basic shift register 128 times. At the level of abstraction of the model, the module for 128-bit KOA waits for the modules for LFSR-128 and LFSR-64 to complete their iterations, and then computes the final result. At the level of abstraction of the description in VHDL, the entity for 128-bit KOA waits a number of clock cycles for the entities for LFSR-128 and LFSR-64 to complete their operation, and then computes the final result.

Figure 5.6 shows the results of the functional simulation of the entity generated from the model of 128-bit KOA. This entity takes 130 clock cycles to multiply the two 128-bit integer operands, because 130 is the number of clock cycles required to complete the execution of the LFSR with longer latency. Figures 5.6(a) and 5.6(b) illustrate the first two clock cycles of the simulation, during which the state machines of the two LFSR entities are set to their initial states. Figures 5.6(c) and 5.6(d) show the transition of the LFSR entity with shorter latency from its last state to its initial state, meanwhile the other LFSR continues its iteration. Finally, Figure 5.6(e) illustrates completion of iteration of the LFSR entity with longer latency, and the product of the input integers. The values of the input 128-bit integers must be hold during the latency period of the 128-bit KOA entity for it to compute a correct result.

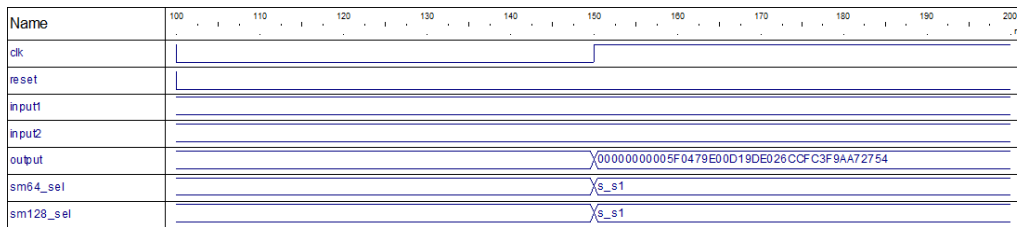
## 5.5 Discussion

The waveforms illustrated in figures 5.4, 5.5 and 5.6 provide evidence that the transformation generates VHDL code whose behaviour matches the functional description indicated by the model. This code is structural because it contains only signal assignment statements, instantiation and interconnection of entities, and processes implementing multiplexors and state machines. Thus, the descriptions in VHDL produced by the design are synthesizable and ready to produce a configuration for a hardware platform like an FPGA. The profile of UML 2 and the transformation tool describe and process arbitrarily complex models describing a hierarchical composition of modules.

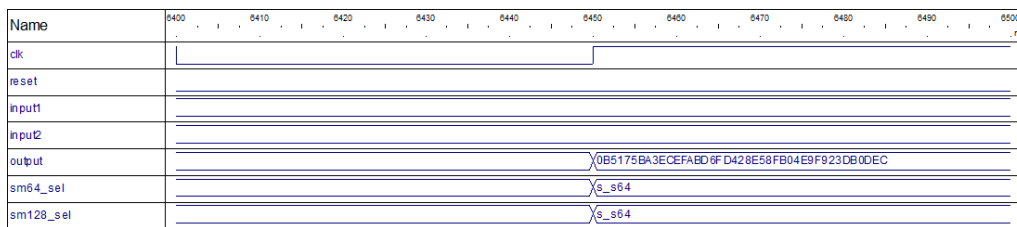
The transformation does not produce optimized VHDL code, which is its main



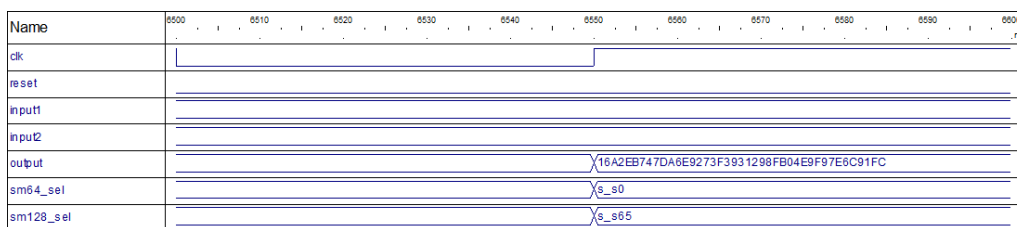
(a)



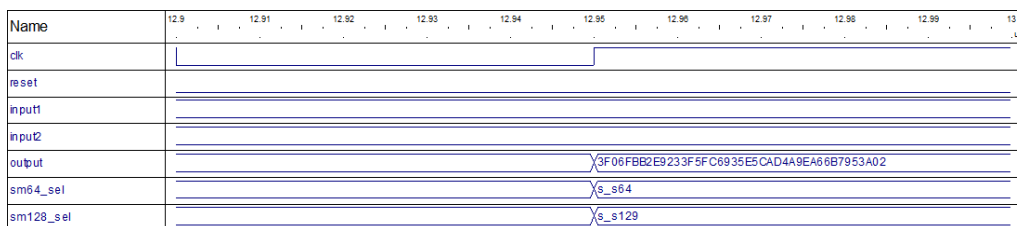
(b)



(c)



(d)



(e)

Figure 5.6: Results of the simulation of the description in VHDL implementing KOA and LFSR.



limitation. Implementing techniques that generate code using as few signals and assignment statements as possible requires more development time and resources, and detect hazardous conditions in the high-level models, like combinational feedback loops. Adding these optimization and analysis mechanisms would produce efficient implementations, and help designers take better decisions based on information provided by an automatic analysis of the source models.

In spite of the limitations, the evidence indicates that the goal of implementing a functional design flow from UML 2 models down to VHDL code was met satisfactorily. The profile of UML 2 allows the designer to represent the flow of information mandated by an algorithm correctly, and the transformation interprets this data flow and generates structural VHDL code effectively. After the successful implementation and testing of the design flow described in the previous chapters, it is time to discuss a possible way to measure its impact in productivity, and its integration with a well-known methodology to design software systems.

# Chapter 6

## Processes, methods and metrics

This dissertation shows that it is possible to use a contemporary technology to engineer software in the design of digital hardware systems. This is feasible due to the similarities between the tasks of developing software systems and describing the functionality of digital hardware systems at the level of abstraction provided by ESL. This situation raises the questions: is it possible to use standard processes, methods and metrics defined for software engineering projects, along with the proposed design framework, in a digital hardware design project? how to adapt such processes, methods and metrics to the domain of digital hardware systems to obtain quality products in a timely manner? This chapter identifies a modern software process that can be extended by the proposed design flow, and used to support the activities of digital hardware designers. In addition, this chapter suggests a methodology to measure the impact of the proposed design flow in the productivity of designers.

### 6.1 It is all about quality

Figure 6.1 shows that any software engineering project is founded on a strong commitment to comply with the highest level of *quality* possible. Pressman defines software quality as “an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it” [54]. The standard 9126 from the International Organization for Standardization

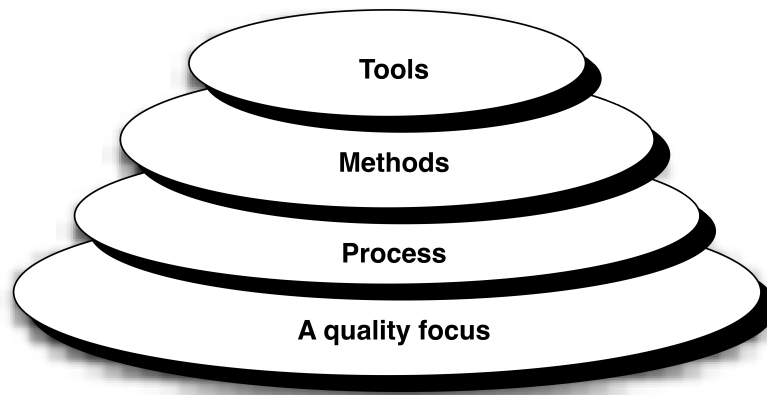


Figure 6.1: The layers of a software engineering task (from [54]).

(ISO) defines quality of software products in terms of a number of attributes that can be measured [27]. The *internal attributes* of a software system are evaluated by measuring intermediate products generated during the process, the *external attributes* are evaluated by measuring parameters during the execution of the system, and *quality in use attributes* are evaluated by measuring the effects of the software product. Internal and external attributes include: functionality, reliability, usability, efficiency, maintainability and portability. The attributes related to quality in use include: effectiveness, productivity, safety and satisfaction.

The *software process* defines the management tasks controlling the development of the project, which indicate what methods to apply and when they are required, what artifacts to produce and when they are needed, what milestones to establish, and how to cope with changes. The *methods* provide technical guidance on how to carry out requirements engineering, design, modeling, implementation, testing, deployment and maintenance. The *tools* are computer programs that aid in every phase of the process and ease the application of methods; it is desirable that all of the tools share information flawlessly and are integrated as much as possible.

Now, what does quality mean in the world of digital hardware design? Rabaey et al. describe the following measurable properties that allow designers to quantify the quality of a digital hardware system: cost, functionality, robustness, performance and energy consumption [56]. Depending on the problem to solve, some of these

properties can be more relevant than the others. The effort to achieve higher levels of design quality led to the foundation of the International Society for Quality Electronic Design (ISQED), which promotes quality and innovation in the design of electronic and electro-mechanical systems. The rest of this chapter discusses how to adapt software processes and metrics to the realm of digital hardware design with the goal of achieving higher levels of quality.

## 6.2 Related work

Protheroe et al. claim that it is possible to quantify quality of a digital hardware system by the ratio of the measure of complexity of the functional description of a system (CD), written in VHDL, to the measure of complexity of the implementation of the system (CI) [55]. This statement is based on the argument that quality is inversely proportional to complexity. The measure of complexity of the functional description depends, in an unrevealed way, on the following data: number of lines of code, the number of branch statements in the flow of control, and the number of input/output ports and signals in the description. The data on which the measure of complexity of the implementation depends are: the number of hardware resources consumed by the implementation in the target platform, the cost of testing, and the cost incurred as a result of failure of the design. The authors conclude that the quality improves as the complexity of the implementation decreases, and that it is possible to predict the cost in terms of hardware resources (a parameter in CI) from the parameters defining the complexity of the description (CD). This document does not suggest methods and tools to collect the required measures, and does not indicate which software process could be adapted to evaluate quality for digital hardware systems either.

Dias et al. state that the quality of a digital hardware system is “a measure of the fulfillment of a given set of valued characteristics” [18]. The valued characteristics considered by the authors are: *modularity*, *testability* and *diagnostic capability*, which are evaluated for the functional description of the system, and for its implementation. The authors emphasize the importance of describing the functionality of systems according to a “design for testability” spirit, and propose their own methodology to

optimize the functional descriptions for modularity and testability. At the implementation level, the authors' methodology estimates the effectiveness of the optimizations at the functional description level through tests, and evaluates the quality of the testing process by using metrics of the defects detected in the implemented system. This work does not take into account software processes, but highlights the importance of applying metrics related to object-oriented design during the development of the functional descriptions.

## 6.3 Proposal for a software process and metrics

This section describes the proposed modifications to a standard software process to adapt it to the functional description of digital hardware systems with model-driven technologies. The description of the proposal considers the implementation of the digital hardware systems using only FPGAs because FPGAs allow fast prototyping and short development cycles. The modification of the software process to support design flows for VLSI ASICs is beyond the scope of this project. The analysis presented in this section is based on the work carried out by Watts S. Humphrey that led to the definition of the Personal Software Process [24].

### 6.3.1 Principles of the proposal

Defining a new software process is by far a complex task, and an unnecessary one if considering the inclusion of the modeling language and the synthesis tool described in the previous chapters in a development process. Thus, to devise a software process where the proposed design flow plays a key role, we can take an existing one and suggest extensions or modifications. Now, what of the existing software processes is the best candidate for extension? The best candidate is the one that meets a number of requirements defined in terms of the current status of this project, and the resources required to implement a process using an incipient technology.

The requirements that the candidate software process must meet are:

**Personal.** The process shall provide the mechanisms and metrics to evaluate success at the personal level, not only at the team or organisational level.

**Adaptable.** The process must not be tied to a programming language or design flow. It must be possible to apply it using any language or methodology.

**Proven.** The process must have been used in industry for a while.

**Simple.** The documentation of the process must be understandable and relatively short.

The process that meets the previous requirements is the Personal Software Process, released in 1993 and widely adopted by the software industry, specifically by companies like Microsoft and Motorola.

The design flow described in previous chapters does not imply a radical deviation from the methods and techniques used to design and build software systems. However, there are a number of design phases to perform to implement a description in VHDL in a FPGA platform: synthesis of the functional description in VHDL, place and route, back annotation, post-synthesis simulation, and testing. The methods, logs and metrics of the extended PSP may take into account the information gathered during each of these phases using appropriate metrics. In this manner, the extended PSP for the design and development of digital hardware systems may reach its goal of generating products with the minimum number of defects on time and within the budget planned.

### 6.3.2 Introduction to the Personal Software Process

According to Humphrey, “when the engineers use PSP, the recommended process goal is to produce zero-defect products on schedule and within planned costs”. PSP recognizes that the performance of every engineer is different, and can be evaluated by measuring the time the engineer spends on each step, the defects that the engineer introduces and fixes, and the size of the engineer’s final products. PSP also recognizes that planning plays a crucial role, and encourages the engineers to plan their work before committing to or starting on a job. The engineers base their plans on personal information from previous projects and estimates for the current project. PSP has a strong commitment to quality, and encourages engineers to feel responsible for the

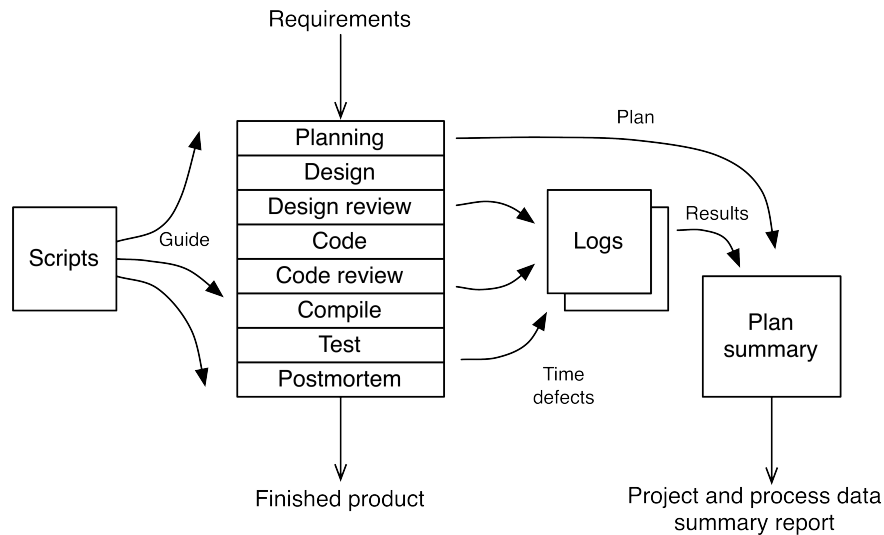


Figure 6.2: Process flow for Personal Software Process (from [24]).

quality of their products, measure and track product quality, and analyze the results of the project to improve their personal processes.

Figure 6.2 illustrates the phases of PSP and the data generated from them. The process starts, as any engineering process, with a requirements specification that is the input to the first phase: planning. The engineers perform every phase by following the instructions of the corresponding script, and record data about the time it takes to complete tasks, and the defects detected and fixed, in the corresponding log. The last phase, postmortem, is very important because it allows the engineers to summarize data about time and defects, measure the size of the resulting product, and use this information to detect and correct flaws in the performance of every engineer. The engineers store this information in a report that can be retrieved during the planning process of a future project to compute estimates for that project.

PSP defines quality in terms of the number of defects found and corrected in the software product. Thus, the main objective of this process is to find and fix as many defects as possible before every compilation or test of the software system. Phases design review and code review, illustrated in the diagram in Figure 6.2, allow the engineers to inspect their work personally, before compiling and testing, to find

defects that are so common that they are injected repeatedly. Well-trained engineers are able to find and correct many defects per hour, and understand their designs thoroughly to avoid making mistakes.

Humphrey indicates that collecting information about the size of the software system produced, amount of time required by every task of the process, and number of defects detected and fixed enables the engineers to compute a set of measures of quality. No single measure characterizes the overall quality of a system, but the full set of measures does provide engineers with more reliable information about quality. The principal metrics of quality considered by PSP are: defect density, review rate, development time ratios, defect ratios, yield, defects per hour, defect removal leverage, and appraisal to failure ratio (A/FR).

## 6.4 Evaluation of productivity

It is necessary to measure the productivity of the employees of an organization to improve it. The process of developing software systems has many commonalties with the process of describing the functionality of digital hardware systems; thus, the discussion about the evaluation of productivity when using the proposed design flow relies on estimations of productivity defined for software development processes. This clause indicates that the same principles apply when evaluating productivity during the process of designing digital hardware systems.

### 6.4.1 Measures of productivity

A software development process receives a set of requirements as inputs and produces a software system as its output after consuming a number of resources. According to Card and Yu et al. [62], the conceptual software development productivity is the ratio of the measurement of the output produced and the measurement of the resources consumed:

$$productivity = \frac{output\ produced}{resources\ consumed} \quad (6.1)$$

It is necessary to define, in a clear and objective way, what the measurement of output and resources are, and how to compute them, to make software productivity



measurements meaningful.

A common metric of the output produced by the software development process is the estimation of the size of the system released and delivered to the customer. There are two criteria to compute the size of a software system:

**Lines of code.** The number of uncommented sentences in the source code of the system released. These sentences of production source code can be classified into three classes:

- New code.
- Ported or reused code.
- Code modified as a result of changes required and bug fixes.

The total size equals a weighted sum of the sizes of the different classes of source code.

The support code, employed to build and test production code, is not released to customers. It is not usually considered for productivity evaluation either, but if it is, the considerable resources needed to develop it must be taken into account in the denominator of Expression 6.1 [62].

**Amount of functionality.** The amount of functionality provided by a software system can be estimated by a breakdown of its principal data inputs and outputs [3].

The general approach is to list and count the number of external user inputs, user inquiries to the system, outputs, and files generated by the software system. These elements are the external manifestation of the system, and cover all of its functionality. The inputs and outputs are counted individually, and weighted with values indicating the importance of the function that consumes the input, or generates the output. The weighted sum of the counters of inputs and outputs is a *function point*.

The resources consumed during the development process include effort, computing and networking costs, and compensation of personnel [62]. The metrics employed to express the amount of resources consumed are:

- For development effort, it is possible to use any of the typical man-year, man-month, man-day, man-hour, staff-year, staff-month, or staff-day, staff-hour measures.
- For cost a monetary unit like dollars (USD).

Each organization must select the metrics for the parameters of the Expression 6.1 based on the availability of reliable measurement methods and tools [62], and construct a suitable indicator that considers the multiple factors that affect productivity. For instance, Figure 1.1 illustrates the increase in the rate of productivity in the semiconductor industry along the last 20 years. The computation of this rate uses the transistors per staff-month metric derived from Expression 6.1 when the metric of size of an electronic system is set to its number of transistors.

### 6.4.2 Proposed methodology to measure productivity

Previous chapters identified design complexity as a factor having a significant impact on the productivity of the development team of digital hardware systems. Each development team in an organization must identify the set of factors that prevent it from reaching higher levels of productivity; if design complexity is one of them, the team should consider using a high level design flow. In addition, the team must adopt appropriate mechanisms to alleviate the impact of the rest of the factors.

The evaluation of productivity during the development process of a digital hardware system is required to prove that a design flow based on MDA helps to alleviate design complexity. The proposal to evaluate productivity uses Expression 6.1, where the *output\_produced* parameter provides a measurement of functionality of the system, and the *resources\_consumed* parameter provides a measurement of the effort required to describe the functionality of the system using the man-day unit. The reasons for selecting these parameters are the following:

- The designer's primary concern is to describe the system's functionality using models instead of writing code, which will be automatically generated. The customer's primary concern is that the system meets the functional requirements and be delivered on time.

- A typical resource that can be consumed during a research project is human effort. Money is not always available as required.

The proposed strategy to measure productivity is as follows:

1. Select two digital communication systems whose functionality can be modeled using the proposed design flow and an RTL design flow.
2. Set a development environment where the major factor impacting productivity is design complexity, and the impact of the remaining factors, like people experience, is minimum.
3. For each of the systems selected use a conventional RTL design flow, and the proposed design flow based on MDA, to describe and implement them. The output of both processes should be a validated VHDL code of the systems.
4. Use function points to compute the amount of functionality implemented every day and keep records. Additionally, keep records of the effective time the designer spent on describing such functionality; do not include breaks. This recording requires a software tool that keeps track of the relevant information, and automates the computation of productivity.
5. Define a scale that ranks productivity measurements and classifies them as low, satisfactory, good or high.
6. Compare the productivity measurements after completion of the two design flows. The conditions under which the two design processes are executed must be the same to guarantee fair comparisons. There is no sense in comparing productivity estimations for the proposed design flow with estimations from other people that executed different development processes under different conditions.

## 6.5 Discussion

Evaluation of productivity requires a careful and controlled planning of tests, software tools, procedures and metrics. In addition, it requires a pilot team of designers with

a certain level of experience and skills. Finally, evaluation of productivity requires time during which the designers perform tests, exercise the development tools, and gather data. Unfortunately, it was not possible to set up the infrastructure and gather the human resources required to perform accurate evaluations for this project. However, the software process and methodology described in previous clauses can still be adapted to work well in projects to design complex digital hardware systems.

Adapting the processes and methodologies of software engineering to the realm of digital hardware design makes sense because of the nature of current development tools and languages. Of course, there are issues related to the implementation of electronic circuits that are not present during the development of software systems, but both development processes share similarities to a large extent. Adapting and unifying the processes and methodologies in such a way that they work with appropriate changes for both worlds may be an attractive field of study. Among the variations are the use of different parameters to estimate productivity.

# Chapter 7

## Conclusions

This dissertation described a domain-specific modeling language that allows the designer to describe algorithms processing bit-blocks using graphical high-level abstractions, and a synthesis tool that transforms such descriptions into VHDL design files that can be implemented in a hardware platform. This chapter discusses results, indicates future challenges, and summarizes the contributions of this project.

### 7.1 Concluding remarks

This dissertation provided evidence that it is possible to apply the paradigm of model-driven engineering to the development of digital hardware systems. This dissertation described a design flow consisting of a domain-specific modeling language and a transformation tool that isolates the designer from low-level abstractions in the digital circuit world. In spite of its early stage, the design flow can be used to describe a number of algorithms, synthesize VHDL code from them, and simulate these descriptions using standard test vectors. Finally, this dissertation contributed to the state of the art of EDA by providing designers with a tool that raises the level of abstraction when describing the functionality of a digital hardware system. According to an anonymous reviewer of one of our publications, the proposal is commendable because “the effort of using a high level abstraction language in describing a hardware design has indeed become necessary, as engineering problems are getting more complicated

continuously”.

Since model-driven engineering is an incipient technology, the supporting software tools and specifications are subject to improvements. On the one hand, although software technologies allowing software engineers to write model-to-model and model-to-text transformations have reached a functional state, their performance must be improved. On the other hand, some specifications also require refinement to make them more manageable and powerful. For instance, UML 2 supports the construction of different kinds of detailed diagrams at the cost of having a complex meta-model populated by hundreds of interrelated meta-classes. Also, Acceleo inherited the limitations of OCL to perform complex operations on collections.

The material in this document can be used as a starting point for other projects that intend to exploit the paradigm of model-driven development. This dissertation provided the reader with basic concepts like meta-modeling, the meta-model of UML 2, profiles and model-to-text transformations. Interested readers may expand this knowledge further by examining the existing literature and experimenting with the software tools available. The interested developer can also experiment further with the software tools, based on Eclipse and EMF, to build models and write model-to-model [28] and model-to-text transformations [41].

The last chapter of this dissertation provided suggestions on extending an existing software process with the proposed design flow and useful metrics. The application of software processes to management of digital hardware system development may be an attractive area of research; an idea supported by the growing number of ESL technologies available at the EDA market currently. The unification resulting from applying software processes to the world of digital hardware systems may result in a meta-process that can be instantiated to different applications domains, not only software engineering or hardware engineering. These modeling and meta-modeling tasks can be supported by Software Process Engineering Metamodel (SPEM) version 2, a profile of UML 2 [50].

## 7.2 Future work

There are a number of directions that can be addressed to extend this project. First, analyzing the convenience of using a different meta-model as the base for the domain-specific modeling language. The meta-model of UML 2 is difficult to follow due to the large number of meta-classes and associations it contains. However, the advantages of UML 2 are its standardization, its openness, its widespread use by students and practitioners, and the availability of many software tools supporting it. Other commercial meta-models, like SCADE [61] and MetaEdit+ [30], lack one or more of these advantages.

Second, building a stand-alone integrated design environment (IDE) that gathers the modeling tool, the proposed synthesizer, the compiler of VHDL, and the simulator. It is necessary to have all of the development tools available in a single environment, but it is not necessary to build this environment from scratch. The modeling and development engines work on top of Eclipse, but not the VHDL compiler and simulator. There are plug-ins for VHDL that work on top of Eclipse and can be used to integrate all of the components within a single environment. Some issues that require resolution are the exchange of modeling projects between installations of the same development environment in different platforms, and between different development environments. The experience is that exchanging models between tools from different vendors through XMI does not work well.

Third, executing and validating models before synthesis. It is desirable to validate the system at modeling phase, not only at a phase after synthesis to VHDL. In this respect, the expectation from the developer is that the models are able to get executed, receive inputs and produce outputs transparently. Stephen Mellor proposed the idea of adding an action language to UML that allowed the model to create instances of classes, establish associations, perform operations on attributes, or call state events [36]. As a consequence, the OMG released a specification for precise semantics for an executable subset of UML 2, or Foundational UML (fUML), on February 2011 [51]. The specification of the action language for this subset of UML 2, or Action Language for fUML (Alf), is still in beta form. Therefore, no implementation is ready yet.

Fourth, building a transformation tool that generates source code that includes

components and intellectual property (IP) cores for a specific family of FPGAs. Thus, it is possible to develop a set of domain-specific modeling languages for different application domains, and write a family of transformation tools, each transforming models in a modeling language to an implementation for the corresponding platform.

## 7.3 Contributions

The main contributions of this project, as stated previously, are a domain-specific modeling language, and a transformation tool from the high-level models to VHDL. The proposed profile is suitable for standardization, so other people can use it to build their own transformations to other languages and platforms, or extend it to support a related application domain. This dissertation, to a lesser extent, also intends to provide the reader with a tutorial on adapting the meta-model of UML 2 to an application domain through profiles, and develop model-to-text transformations with Aceleo.

## 7.4 Publications

During the last years, we published one paper in the proceedings of an international conference, a book chapter and a paper in a journal indexed by the Journal Citations Reports. These documents are the following:

1. Balderas-Contreras, T., Rodríguez-Gómez, G. and Cumplido, R. On Model-Driven Engineering of Reconfigurable Digital Control Hardware Systems. *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*. Editors: Khalgui, M. and Hanisch, H. pp. 190–208. IGI Global. 2011. DOI: 10.4018/978-1-60960-086-0.ch008.
2. Balderas-Contreras, T., Rodríguez-Gómez, G. and Cumplido, R. A UML 2.0 Profile to Model Block Cipher Algorithms. In Proceedings of the 6th European Conference on modeling Foundations and Applications (ECMFA 2010). pp. 20–31. Lecture Notes in Computer Science, Volume 6138. 2010. DOI: 10.1007/978-3-642-13595-8\_4.



3. Balderas-Contreras, T., Cumplido, R. and Rodríguez-Gómez, G. Synthesizing VHDL from Activity Models in UML 2. *International Journal of Circuit Theory and Applications*. 2012. DOI: 10.1002/cta.1874.

Additionally, the following documents were submitted for review to different journals indexed by the Journal Citations Reports:

1. Balderas-Contreras, T., Cumplido, R. and Rodríguez-Gómez, G. Using Model-Driven Development to Educate Software Engineers in the Design of Digital Hardware Systems. Submitted for review to *International Journal of Electrical Engineering Education*.
2. Balderas-Contreras, T., Rodríguez-Gómez, G. and Cumplido, R. Programming with Domain-Specific Models in the Unified Modeling Language Version 2. Submitted for review to *Software: Practice and Experience*.

# Bibliography

- [1] 3GPP. Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification. Technical Report 3GPP TS 35.202 version 9.0.0 Release 9, 3rd Generation Partnership Program, 2009.
- [2] 3GPP. Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 3: Implementors' Test Data. Technical Report 3GPP TS 35.203 version 9.0.0 Release 9, 3rd Generation Partnership Program, 2009.
- [3] A. J. Albrecht and J. E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Trans. Softw. Eng.*, 9(6):639–648, November 1983.
- [4] David Arditti Ilitzky, Jeffrey D. Hoffman, Anthony Chun, and Brando Perez Esparza. Architecture of the Scalable Communications Core's Network on Chip. *IEEE Micro*, 27:62–74, September 2007.
- [5] Colin Atkinson and Thomas Kühne. Model-driven Development: A Metamodeling Foundation. *IEEE Software*, 20:36–41, September 2003.
- [6] Colin Atkinson and Thomas Kühne. A Tour of Language Customization Concepts. *Advances in Computers*, 70:105–161, 2007.
- [7] Stephen A. Bailey, Peter J. Ashenden, J. Bhasker, Dennis Brophy, Patrick K. Bryant, Ernst Christen, Wolfgang Ecker, Masamichi Kawarabayashi, Robert H. Klenke, Satoshi Kojima, Jim Lewis, Paul J. Menchini, Jean P. Mermet, Gregory D. Peterson, Lance G. Thompson, Alain Vachoux, and John Willis. IEEE

- Standard VHDL Language Reference Manual. Technical Report IEEE Standard 1076-2002, IEEE Computer Society, 2002.
- [8] Tomás Balderas-Contreras. Hardware/Software Implementation of the Security Functions for Third Generation Cellular Networks. Master's thesis, Instituto Nacional de Astrofísica Óptica y Electrónica, Tonantzintla, Puebla. MEXICO, Dec 2004.
- [9] Imène Benkermi, Mohamed El Amine Benkhelifa, Daniel Chillet, Sébastien Pillement, Jean-Christophe Prévotet, and François Verdier. System-Level Modelling for Reconfigurable SoCs. In *Proceedings of the 20th Conference on Design of Circuits and Integrated Systems (DCIS)*, Lisboa, Portugal, November 2005.
- [10] Dag Björklund. The SMDL Statechart Description Language: Design, Semantics and Implementation. Master's thesis, Åbo Akademi University, Turku, Finland, Nov 2001.
- [11] Dag Björklund and Johan Lilius. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In *Proceedings of the Proceedings of the 20th IEEE Norchip Conference, NORCHIP02*, 2002.
- [12] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented Analysis and Design with Applications, Third Edition*. Addison-Wesley Professional, third edition, 2007.
- [13] Jorge Castiñeira Moreira and Patrick Guy Farrell. *Essentials of Error-Control Coding*. John Wiley & Sons, Ltd, Chichester, West Sussex, England, 2006.
- [14] International Roadmap Committee. Design. In *International Technology Roadmap for Semiconductors*. 2007.
- [15] Frank P. Coyle and Mitchell A. Thornton. From UML to HDL: a Model Driven Architectural Approach to Hardware-Software Co-Design. In *Proceedings of Information Systems: New Generations Conference, ISNG*, pages 88–93, 2005.

- [16] Eduardo Cuevas-Farfán. Reducción Polinomial Mediante LFSR para Multiplicador KOA en  $GF(2^{163})$ . Technical report, Instituto Nacional de Astrofísica Óptica y Electrónica, Tonantzintla, Puebla. MEXICO, May 2012.
- [17] Douglas Densmore, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. A Platform-Based Taxonomy for ESL Design. *IEEE Design and Test of Computers*, 23:359–374, September 2006.
- [18] O. P. Dias, J. Semião, M. B. Santos, I. M. Teixeira, and J. P. Teixeira. Quality of Electronic Design: From Architectural Level to Test Coverage. In *Proceedings of the 1st International Symposium on Quality of Electronic Design*, ISQED '00, pages 197–202, Washington, DC, USA, 2000. IEEE Computer Society.
- [19] Stephen A. Edwards. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design and Test of Computers*, 23:375–386, September 2006.
- [20] Michael J. Flynn and Patrick Hung. Microprocessor Design Issues: Thoughts on the Road Ahead. *IEEE Micro*, 25:16–31, May 2005.
- [21] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [22] Sébastien Gérard. Papyrus User Guide Series. About UML Profiling. Version 1.0.0. Technical report, CEA LIST Institute, 2011.
- [23] Timothy J. Grose, Gary C. Doney, and Stephen A. Brodsky. *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
- [24] Watts S. Humphrey. The Personal Software Process (PSP). Technical Report CMU/SEI-2000-TR-022, Carnegie Mellon Software Engineering Institute, 2000.
- [25] Agility Design Solutions Inc. Handel-C Language Reference Manual. Technical Report RM-1003-4.4, Agility Design Solutions Inc., 2007.
- [26] Xilinx Inc. Synthesis and Simulation Design Guide. Technical Report UG626, Xilinx Inc., 2011.

- [27] ISO/IEC. Software Engineering - Product Quality - Part 1: Quality Model. Technical Report ISO/IEC 9126-1:2001, International Organization for Standardization/International Electrotechnical Commission, 2001.
- [28] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.
- [29] Anatolii Karatsuba and Yuri Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.
- [30] Steven Kelly, Kalle Lyytinen, and Matti Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In *Proceedings of the 8th International Conference on Advances Information System Engineering*, pages 1–21, London, UK, 1996. Springer-Verlag.
- [31] Stuart Kent. Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.
- [32] Thomas Kühne. Contrasting Classification with Generalisation. In *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, APCCM '09, pages 71–78, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.
- [33] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006.
- [34] Gérard Le Lann. An Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective. In *Proceedings of the 1997 International Conference on*

- Engineering of Computer-based Systems*, ECBS'97, pages 339–346, Washington, DC, USA, 1997. IEEE Computer Society.
- [35] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, San Francisco, CA, USA, 2007. 488p.
- [36] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [37] Stephen J. Mellor, John R. Wolfe, and Campbell McCausland. Why Systems-on-Chip needs More UML like a Hole in the Head. In Grant Martin and Wolfgang Müller, editors, *UML for SOC Design*, pages 17–36. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [38] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37:316–344, December 2005.
- [39] Joaquin Miller and Jishnu Mukerji. Model Driven Architecture (MDA). Draft ormsc/2001-07-01, Architecture Board ORMSC, July 2001.
- [40] W. Mueller, A. Rosti, S. Bocchio, E. Riccobene, P. Scandurra, W. Dehaene, and Y. Vanderperren. UML for ESL Design: Basic Principles, Tools, and Applications. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pages 73–80, New York, NY, USA, 2006. ACM.
- [41] Jonathan Musset, Étienne Juliot, and Stéphane Lacrampe. *Acceleo 2.6: User Guide*. Technical report, Obeo, 2008.
- [42] Antoni Olivé. *Conceptual Modeling of Information Systems*. Springer Publishing Company, Incorporated, Secaucus, NJ, USA, 2007.

- [43] OMG. Meta Object Facility (MOF) Core Specification. Version 2.0. Technical Report formal/06-01-01, Object Management Group, 2006.
- [44] OMG. Object Constraint Language. OMG Available Specification. Version 2.0. Technical Report formal/06-05-01, Object Management Group, 2006.
- [45] OMG. UML Profile for System on a Chip (SoC). Version 1.0.1. Technical Report formal/06-08-01, Object Management Group, 2006.
- [46] OMG. MOF 2.0/XMI Mapping, Version 2.1.1. Technical Report formal/2007-12-01, Object Management Group, 2007.
- [47] OMG. OMG Unified Modeling Language (OMG UML) Infrastructure. Version 2.1.2. Technical Report formal/2007-11-04, Object Management Group, 2007.
- [48] OMG. OMG Unified Modeling Language (OMG UML) Superstructure. Version 2.1.2. Technical Report formal/2007-11-02, Object Management Group, 2007.
- [49] OMG. MOF Model to Text Transformation Language. Version 1.0. Technical Report formal/2008-01-16, Object Management Group, 2008.
- [50] OMG. Software & Systems Process Engineering Metamodel Specification (SPEM). Version 2.0. Technical Report formal/2008-04-01, Object Management Group, 2008.
- [51] OMG. Documents Associated with Semantics of a Foundational Subset for Executable UML Models (FUML). Version 1.0. Technical Report formal/11-02-01, Object Management Group, 2011.
- [52] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1. Technical Report formal/2011-01-01, Object Management Group, 2011.
- [53] Vaughan R. Pratt. Anatomy of the Pentium Bug. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 97–107, London, UK, 1995. Springer-Verlag.

- [54] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.
- [55] Dave Protheroe and Francesco Pessolano. An Objective Measure of Digital System Design Quality. In *Proceedings of the 1st International Symposium on Quality of Electronic Design*, ISQED '00, pages 227–233, Washington, DC, USA, 2000. IEEE Computer Society.
- [56] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2003.
- [57] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 704–709, Washington, DC, USA, 2005. IEEE Computer Society.
- [58] Tim Schattkowsky. UML 2.0 - Overview and Perspectives in SoC Design. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 832–833, Washington, DC, USA, 2005. IEEE Computer Society.
- [59] Patrick Schaumont and Ingrid Verbauwhede. A Component-Based Design Environment for ESL Design. *IEEE Design and Test of Computers*, 23:338–347, September 2006.
- [60] Bernard Sklar. *Digital Communications: Fundamentals and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.
- [61] Esterel Technologies. SCADE Suite 6.2: Technical Data Sheet. Technical Report Whitepaper SC-TDS-6.2, Esterel Technologies, 2011.
- [62] Weider D. Yu, D. Paul Smith, and Steel T. Huang. Software Productivity Measurements. In *International Computer Software and Applications Conference*, 1991.



# Appendix A

## Review of the notation of UML 2

The purpose of this appendix is to introduce the reader to the notation of UML 2 to understand the diagrams presented in the previous chapters of this dissertation [48]. Only the class diagram, the activity diagram, and the object diagram will be reviewed because they are the ones used throughout this document. The next sections describe the most relevant features of these diagrams, including their goals, their semantics, and their constituting elements.

### A.1 Class diagrams

Of all of the UML 2 diagrams, the class diagram is the most commonly used and known. It defines the types of the objects present in an application domain or in a software system, depending on whether the diagram represents a domain model or a design model, respectively. It also shows the *relationships* existing between these types, which model the links existing between the objects in the application domain or the software system. The types just mentioned are called *classes* and their definitions include *properties* that characterize the state of the instances of the class, and *operations* that describe the behavior of such instances.

### A.1.1 Classes and relationships

Figure A.1 illustrates the graphical representation for each of the concepts highlighted above. The figure shows the different types of classes, properties and relationships that can be included in a class diagram. Let us describe these modeling elements in detail.

**Class.** A class is depicted in diagrams using a box with three compartments, with the name of the class specified in the uppermost compartment. To indicate that the class is an *abstract class*<sup>1</sup>, its name shall be italicized. If the name of the class is not italicized, then it represents a *concrete class*<sup>2</sup>. Figure A.1 shows that **Employee** and **Airplane** are abstract classes, whereas the others are concrete classes.

**Property.** A property in a class represents a datum that is present in every instance of such class. The set of properties defined by a class forms the status of every instance, which typically changes as time passes. The properties are listed one after another in the second compartment of the corresponding class box. Each property has its own attributes that can be specified along with its name, including its type, its visibility (private, public or protected), its multiplicity, a default value, and some strings representing restrictions. The class diagrams in this document show only the type of properties.

**Operation.** An operation denotes a specific behavior that can be invoked on every instance of a class. The operations are listed one after another in the third compartment of the corresponding class box. Each operation has its own attributes that can be specified along with its name, including the type of the returned value, its multiplicity, the name and type of each parameter, and some strings representing restrictions. The class diagrams in this document show only the type of the value returned by the operations.

---

<sup>1</sup>An abstract class defines attributes that are shared by all of its sub-classes, and a set of empty operations that are to be implemented also by its sub-classes. This kind of class cannot be instantiated directly.

<sup>2</sup>A concrete class can be instantiated.

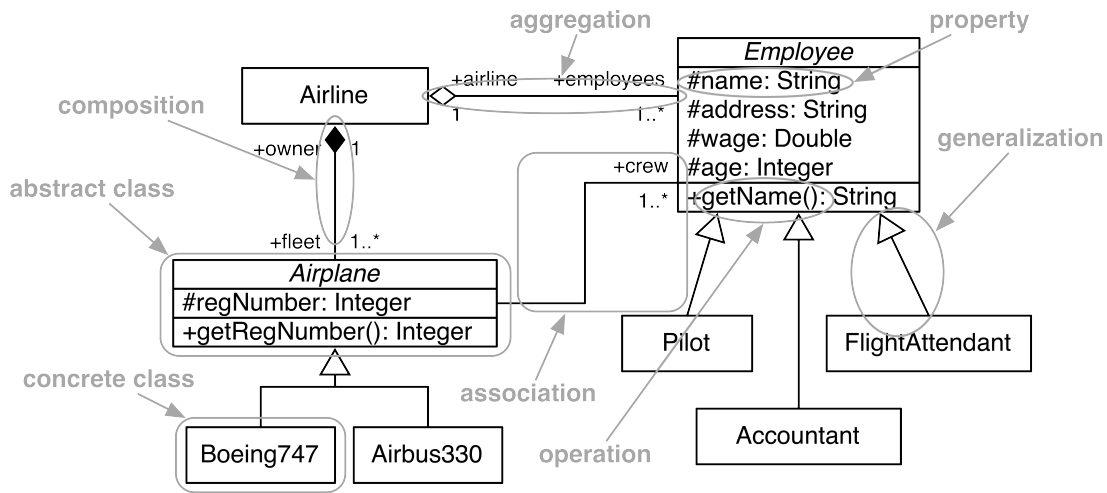


Figure A.1: A class diagram in UML 2 and its components.

Now, let us describe three relationships between classes that can be present in a class diagram.

**Generalization.** A taxonomic relationship between a more general class and a more specific class. It is denoted by a line departing from the specific class (or subclass) and arriving to the general class (or superclass), which is pointed to by an empty triangle. As a consequence of this relationship, every instance of the specific class is also an indirect instance of the general class. Also, the specific class inherits the features (properties and operations) defined by the general class. Figure A.1 shows that **FlightAttendant**, **Accountant** and **Pilot** are more specific classes than the general class **Employee** and that the specific classes inherit all of the attributes defined by the general class.

**Association.** Whenever there is a link between two different objects in an application domain or software system, there is an association relationship between the corresponding classes in a class diagram. The association relationship is expressed by a line between two classes, which models a link between the instances of such classes. In Figure A.1, the generalization relationships arriving to **Airplane** and **Employee** and the association between these two classes imply that there is also an association between the concrete classes **Boeing747**

and **Pilot**, and between **Boeing747** and **FlightAttendant**. As a result, every instance of **Boeing747** may be linked to all of the members of its crew.

**Composition.** This is a “part of” relationship that indicates that an instance is a component of only one owner instance. This relationship is denoted with a line joining two classes, and a filled diamond next to the owner class. The diagram in Figure A.1 indicates that an instance of a concrete sub-class of **Airplane** is owned by only one instance of the class **Airline**. Normally, an airplane cannot be used by more than two airlines at the same time.

**Aggregation.** Another “part of” relationship that intends to model that an instance may be part of multiple owner instances. This relationship is denoted with a line joining two classes, and a hollow diamond next to the owner class. The diagram in Figure A.1 indicates that an instance of the class **Airline** has several instances of concrete sub-classes of **Employee** aggregated to it. This relationship is useful to model that a freelance accountant can work for different airlines at a given time.

The properties placed at the ends of the relationships in the diagram, in Figure A.1, are known as *roles*. A role located at a specific end of a relationship belongs to the class located at the opposite end. Every role in the diagram shows its multiplicity, which is an attribute of a property that indicates how many objects the property refers to. The multiplicities shown in the diagram are: 1..\*, which indicates an unbounded number of instances greater than one, and 1, which indicates a single instance.

### A.1.2 Derived unions

The concept of *derived union* is used extensively in the meta-model of UML 2, let us explain it using a familiar example. An airline owns different kinds of facilities like offices to issue tickets, hangars to maintain aircrafts, and office buildings for managerial operations. The class diagram in Figure A.2 indicates that every instance of the concrete class **Hangar** shall be linked to two sets of instances. The first collection is accessed through the property *engineers* and represents the set of mechanical engineers performing maintenance activities in hangar. The second collection is accessed

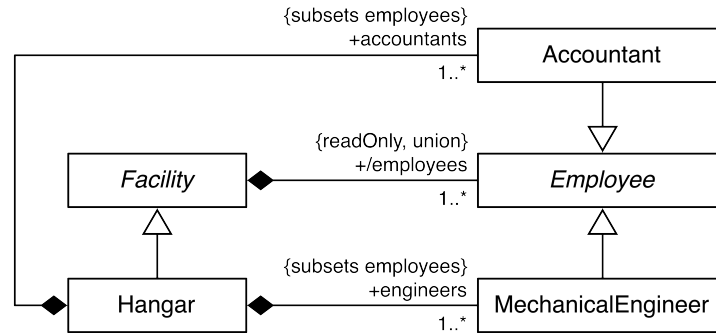


Figure A.2: A class diagram illustrating the concept of derived union.

through the property *accountants* and denotes the set of accountants carrying out the managerial and logistic operations required in the hangar.

The concrete class **Hangar** inherits the public property *employees* from the abstract class **Facility**, which is a reference to any collection of employees (indirect instances of the abstract class **Employee**). Thus, *employees* is perfectly able to refer to the collection that results from merging the set of accountants (referred to by *accountants*), and the set of engineers (referred to by *engineers*). The string `{readOnly,union}` next to *employees*, and the fact that such property is *derived*<sup>3</sup> indicate that the value of *employees* is computed by performing a union on the values of other properties. What other properties? Those labeled by the string `{subsets employees}` next to their names (*accountants* and *engineers*). Therefore, the values of *accountants* and *engineers* are merged to compute the value that is assigned to *employees*, which is a reference to the whole set of employees working at the facility in question.

### A.1.3 Keywords

A keyword in UML 2 is an identifier that is enclosed in guillemets (`« »`) and modifies the meaning of an existing modeling element in UML 2. For instance, the diagram in

<sup>3</sup>A derived property is indicated by the symbol `/` next to its name. The value of a derived property is always computed from the values of other properties. When implementing the model in software, derived properties might be translated to methods instead of actual instance variables.

Figure A.3 shows that the class box commonly used to define classes can be reused to represent interfaces. Since the concept of class is similar to the concept of interface, it is possible to modify the class box to represent an interface by adding the keyword «interface». The designers of UML 2 decided not to overload the language with too many symbols, and introduced keywords that mark modeling elements to indicate something different. Every modeling element that represents an interface can be related to another through a generalization relationship, and can be related to classes through two new relationships: dependency and implementation, as illustrated in Figure A.3.

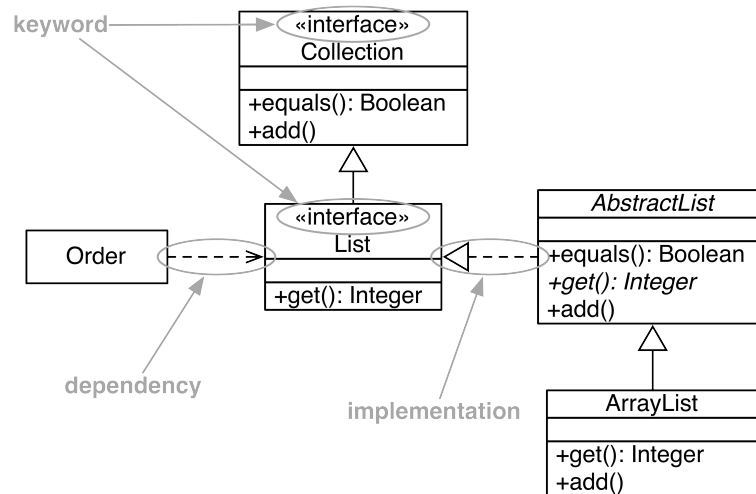


Figure A.3: The definition of an interface in UML 2 requires modifying a class with a keyword.

It is important not to confuse keywords with property strings. In Figure A.2, the property strings `{readOnly,union}` and `{subsets employees}` in the diagram do not change the meaning of the property to which they are attached, but provide more information about it and its values.

## A.2 Activity diagrams

The activity diagram in UML 2 describes operations performed by a software system or behaviors occurring in an application domain. Unlike the class diagram, which focuses on the static structure of a software system or application domain, the activity diagram focuses on the dynamic aspects of the system or domain to model. This diagram allows modeling control flows, as does a flowchart, and describing parallel control flows and data flows. This section discusses only activity diagrams that describe data flows, receive input data and produce output data.

### A.2.1 Activities, nodes and edges

Figure A.4 shows a simple activity diagram of the kind used in this dissertation. This diagram contains a single activity, called “anActivity”, consisting of a number of activity nodes and actions interconnected by edges. UML 2 classifies nodes into *object nodes* and *control nodes*, and edges into *object flows* and *control flows*. All of the edges in Figure A.4 are object flows because they allow data objects pass through them. Some nodes are object nodes that consume and/or produce data objects, and others are control nodes that coordinate the flow of control. Also, an activity may contain many kinds of actions serving different purposes.

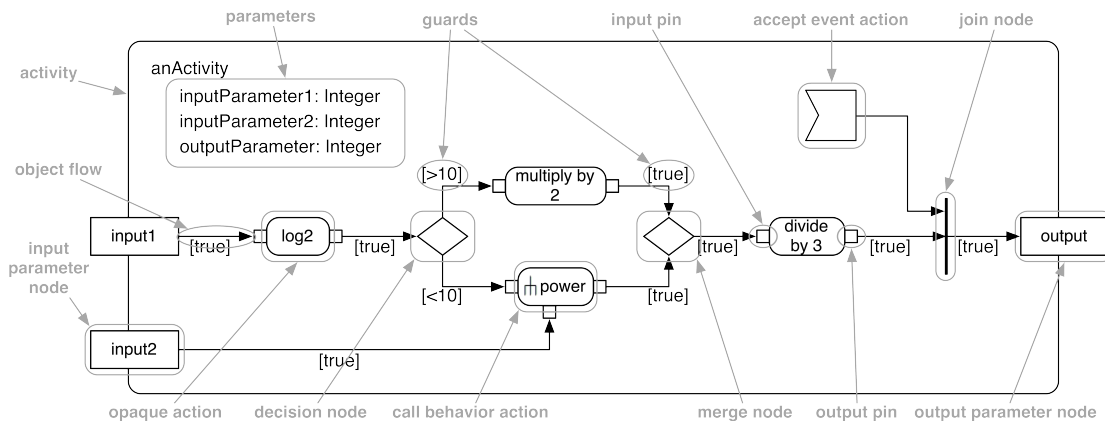


Figure A.4: An activity diagram in UML 2 and its components.

Let us consider the nodes and edges making up by the activity in Figure A.4:

**Activity.** Denoted by a rectangle with rounded corners, an activity describes behaviors in terms of other subordinate behaviors whose execution is coordinated by means of a data or control flow. All of the activities in this dissertation are parameterized and initiate their execution when new data objects are available in their input parameters.

**Activity parameter node.** Is an object node that represents input or output data. It is represented by a rectangle on the border of the activity. There is no graphical distinction between an input parameter node and an output parameter node; this distinction is made when setting an attribute in the parameter associated to the node.

**Parameter.** Every activity parameter node has a parameter associated to it. This modeling element specifies the type of the objects received/sent by the parameter node, the multiplicity of such data, and the direction of the data (input, output, input/output). A parameter is usually listed in the upper right corner of the activity. In Figure A.4, the parameter called “inputParameter1” receives integers and is associated to the parameter node called “input1”, and the parameter called “outputParameter” is associated to the parameter node “output”. This language construct is the same that is used to model the parameters of operations in class diagrams.

**Decision node.** A control node represented by a diamond with an incoming edge and multiple outgoing edges. The data object received through the incoming edge is offered to each of the outgoing edges, but traverses only one of them. This edge is selected by evaluating the conditions specified by the guards of the edges, and determining which guard evaluates to true. For every decision node, there should be a corresponding merge node, a control node that serves as a convergence point of multiple data/control flows.

**Object flow.** Is represented by an arrow that interconnects nodes and actions within the activity. This kind of edge allows data objects pass along it whenever the associated guard is evaluated to true. The guard is a condition attached to the edge and specified by enclosing it in brackets ([ ]).



## A.2.2 Actions

The action is the fundamental unit of execution when modeling behavior, it represents a transformation or other kind of processing. UML 2 defines many kinds of actions to model a wide variety of situations and operations. The diagram in Figure A.4 contains three different kinds of actions, and other modeling elements generally associated to actions, the input and output pins.

**Opaque action.** The semantics of this kind of action, denoted by a rectangle with round corners, is implementation-specific. The diagram in Figure A.4 shows that this activity is used to indicate three operations performed on integer arguments.

**Calling behavior action.** This kind of action, denoted by a rectangle with round corners and a rake symbol ( $\pitchfork$ ) within, invokes an external activity. The arguments of the action are available to the invoked behavior for use during its execution. In Figure A.4, the call behavior action invokes an activity called “power” and passes its two integer arguments to such activity as inputs.

**Accept event action.** This action is denoted by a concave pentagon and waits for an event that meets a specific condition. The join node in Figure A.4 prevents the integer coming from the opaque action “divide by 3” from reaching the output parameter node until the event occurs. Once the event occurs, the node lets the integer object pass and reach the output parameter node.

**Pins.** A pin, denoted by a small rectangle attached to an action, is an object node that receives incoming data objects (arguments) and sends outgoing data objects (return values). It also specifies the type and multiplicity of the data object for the action. In Figure A.4, all of the pins receive/send integer objects.

## A.3 Object diagrams

The object diagram shows representations of instances of classes defined in a class diagram, the links between them, and the values of some or all of their properties.

The object diagram is a snapshot of the software system at a specific point in time, and allows the designer to understand links that are not comprehensible in the corresponding class diagram. Figure A.5 illustrates the components of an object diagram derived from the class diagram in Figure A.1.

**Instance specification.** This modeling element, denoted by a box with its name and class underlined, represents an object in a software system or application domain. The name of the instance and the name of the class are separated by a colon (:), and sometimes the name of the object may be omitted. The second compartment of the box contains a list of the properties of the instance along with its values. Notice that an instance specification is a representation of an object, not the object itself, and shows the state of the object at a specific point.

**Link.** Is denoted by a straight line joining two instance specifications. As mentioned previously, a link is an instance of the association joining the classes that describe the instances represented by the specifications. The roles in the association establish the link and may be located at the appropriate end of the link.

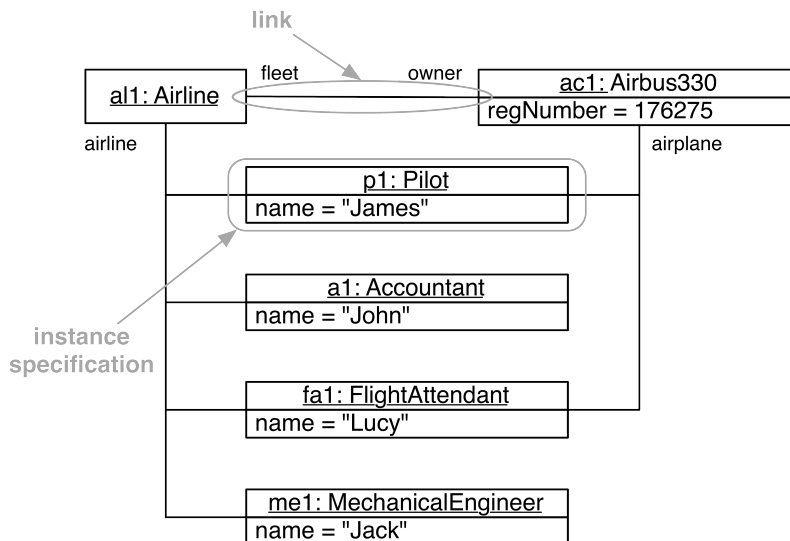


Figure A.5: An object diagram in UML 2 and its components.

# Appendix B

## The profile BitBlockFlow

This appendix documents the most relevant stereotypes in the profile BitBlockFlow. This documentation provides details about the structure of the profile, and allows the designer to understand the extensions provided to adapt activity diagrams to the application domain of interest. Each description includes the name of the stereotype, an indication of whether the stereotype is an abstract or concrete class, the name of the package the stereotype belongs to, the meta-class extended by the stereotype, an enumeration of the constraints defined for the stereotype, and a brief introduction to the responsibilities of the stereotype.

<b>Name:</b>	<b>BBModule.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	ModuleInterface.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>Activity.</b>
<b>Description:</b>	A module is a self-contained unit of behavior describing either a simple operation or a complex algorithm. The module receives input bit-blocks from its environment, and produces output bit-blocks that transfers back to the environment. The internals of the module describe the flow of information from the inputs to the outputs, and consist of nodes representing operations, invocations to other modules, and control nodes. The interface of the module consists of input parameters, output parameters, or input/output parameters. A module may invoke one or more modules, which may invoke one or more modules, and so on. In this way, it is possible to build arbitrarily complex hierarchies of modules, and model any composite algorithm.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. The identifier string must not be empty.</li> <li>2. The parameters of the module must be well-formed.</li> <li>3. The nodes controlling the flow of bit-blocks must be well-formed.</li> <li>4. The nodes representing logic and shift/rotate operations must be well-formed.</li> <li>5. The nodes representing operations that manipulate bit-blocks (split, concatenation, extraction and copy) must be well-formed.</li> <li>6. The dataflows joining nodes must be well-formed.</li> <li>7. The nodes representing switches must be well-formed.</li> <li>8. The module must invoke well-formed modules using well-formed calls.</li> </ol>
<b>Concrete syntax:</b>	The same as the one defined by the extended meta-class.

<b>Name:</b>	<b>BBParameter.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	ModuleInterface.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>ActivityParameterNode.</b>
<b>Description:</b>	A parameter is either a node that receives input bit-blocks and drives them to the internals of the module, or a node that receives bit-blocks from the internals and transfers them to the environment of the module, or a node that performs both of the previous tasks. The parameters of a module are restricted to send or receive sequences of instances of the class <b>Bit</b> (bit-blocks) whose length must be finite and greater than zero. The multiplicity of the parameter indicates the length of the bit-blocks processed.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every parameter must be either an input parameter, or an output parameter, or an input/output parameter.</li> <li>2. Every module must have at least one input parameter and at least one output parameter; otherwise, at least one input/output parameter.</li> <li>3. Every parameter must receive/send an unbounded number of well-formed bit-blocks of the same length. However, the length of the bit-blocks processed by different parameters may differ.</li> <li>4. Every input parameter must have at least one outgoing dataflow and no incoming dataflow. Every output parameter must have only one incoming dataflow and no outgoing dataflow. Every input/output parameter must have at least one incoming dataflow or at least one outgoing dataflow, but not at the same time.</li> <li>5. The names of the parameters attached to the same module must be different.</li> </ol>
<b>Concrete syntax:</b>	The same as the one defined by the extended meta-class.

<b>Name:</b>	<b>BBDataflow.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Edges.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>ObjectFlow.</b>
<b>Description:</b>	A dataflow connects nodes to each other and allows bit-block to traverse through them to reach a target node from a source node.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every dataflow must always allow bit-blocks to traverse through it. This is achieved by setting the guard condition associated to every dataflow to true.</li> <li>2. Every dataflow must allow only one bit-block to traverse through it. This is achieved by setting the weight attribute of every dataflow to one.</li> <li>3. Every dataflow must connect well-formed operands or parameters belonging to well-formed operations or modules. In addition, the interconnected nodes must process a continuous flow of bit-blocks of the same length.</li> </ol>
<b>Concrete syntax:</b>	The same as the one defined by the extended meta-class.


<b>Name:</b>	<b>BBOperand.</b>
<b>Is abstract:</b>	Yes.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>Pin.</b>
<b>Description:</b>	An operand is an element attached to an owning operation. The operation receives the bit-blocks on which it is performing its duties through input operands. The operation produces a bit-block as a result of its execution, and sends this result back to its environment through an output operand. The operations that split and concatenate bit-blocks require multiple output operands and input operands, respectively, that are ordered. The tagged attribute <i>operandIndex</i> defines the order of operands for such operations.
<b>Tagged attributes:</b>	<ul style="list-style-type: none"> <li>• <i>operandIndex</i>. Indicates the index of the operand when the owning operation is a split operation or a concatenation operation. Its default value es 0. The value of this tagged attribute is not considered for other operations.</li> </ul>
<b>Constraints:</b>	Defined by the subclasses.
<b>Concrete syntax:</b>	This is an abstract stereotype. The subclasses of this stereotype may define new concrete syntaxes for the instances of <b>Pin</b> .


<b>Name:</b>	<b>BBInput.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>BBOperand.</b>
<b>Extends:</b>	<b>InputPin.</b>
<b>Description:</b>	An input operand is an element attached to an owning operation. The input operands of an operation receive bit-blocks from other nodes through its incoming dataflow. The number of input operands for every operation must be defined by the stereotype defining the operation.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every input operand must receive an unbounded number of well-formed bit-blocks of the same length. However, the length of the bit-blocks processed by different input operands may differ.</li> <li>2. Every input operand must have only one incoming dataflow and no outgoing dataflow.</li> <li>3. Every kind of operation has its own restrictions on the value of the attribute <i>operandIndex</i>, inherited from the superclass, especially the operations that split and concatenate bit-blocks.</li> </ol>
<b>Concrete syntax:</b>	The same as the one defined by the extended meta-class.
<b>Name:</b>	<b>BBOutput.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>BBOperand.</b>
<b>Extends:</b>	<b>OutputPin.</b>
<b>Description:</b>	An output operand is an element attached to an owning operation. The output operands of an operation send bit-blocks to other nodes through its outgoing dataflow. The number of output operands for every operation must be defined by the stereotype defining the operation.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every output operand must send an unbounded number of well-formed bit-blocks of the same length. However, the length of the bit-blocks processed by different output operands may differ.</li> <li>2. Every output operand must have only one outgoing dataflow and no incoming dataflow.</li> <li>3. Every kind of operation has its own restrictions on the value of the attribute <i>operandIndex</i>, inherited from the superclass, especially the operations that split and concatenate bit-blocks.</li> </ol>
<b>Concrete syntax:</b>	The same as the one defined by the extended meta-class.


<b>Name:</b>	<b>BBOperation.</b>
<b>Is abstract:</b>	Yes.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	This stereotype defines the objects in an activity model affected by BitBlockFlow that extend instances of the meta-class <b>OpaqueAction</b> to represent an operation. This operation may be a bitwise logic operation, a shift/rotate operation, or an operation that manipulates bit-blocks (split, concatenation, extraction and copy).
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	Defined by the subclasses.
<b>Concrete syntax:</b>	This is an abstract stereotype. The subclasses of this stereotype may define new concrete syntaxes for the instances of <b>OpaqueAction</b> .


<b>Name:</b>	<b>LogicOperation.</b>
<b>Is abstract:</b>	Yes.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>BBOperation.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	This stereotype defines the objects in an activity model affected by BitBlockFlow that extend instances of the meta-class <b>OpaqueAction</b> to represent bitwise logic operations.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every bitwise logic operation must have at least two input operands and one output operand.</li> <li>2. The operands of every bitwise logic operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>3. Every bitwise logic operation must produce output bit-blocks whose length is greater than or equal to the length of every input operand.</li> </ol>
<b>Concrete syntax:</b>	This is an abstract stereotype. The subclasses of this stereotype may define new concrete syntaxes for the instances of <b>OpaqueAction</b> .




<b>Name:</b>	<b>And.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>LogicOperation.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent bitwise AND operations on bit-blocks at the input operands that produce a new bit-block at the output operand.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	

<b>Name:</b>	<b>Or.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>LogicOperation.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent bitwise OR operations on bit-blocks at the input operands that produce a new bit-block at the output operand.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	

<b>Name:</b>	<b>Xor.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>LogicOperation.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent bitwise XOR operations on bit-blocks at the input operands that produce a new bit-block at the output operand.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	

<b>Name:</b>	<b>Split.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> extended by instances of this stereotype represent operations that divide their single input bit-block into $n$ output bit-blocks. The length of the input bit-block is given by the multiplicity of the single input operand. The lengths of the output bit-blocks are given by the multiplicities of the output operands. As long as the input bit-block is long enough to be split into the the number of output bit-blocks, there is no limit to the number of output operands.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every split operation must have a single input operand and at least one output operand.</li> <li>2. The operands of every split operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>3. The length of the bit-block at the input operand must be equal to the sum of the lengths of the bit-blocks at the output operands.</li> <li>4. The output operands must be indexed according to an increasing sequence of integers starting at 1. This indexing of operands allows the split operation to identify what segment of the input signal it will assign to what output operand.</li> </ol>
<b>Concrete syntax:</b>	

<b>Name:</b>	<b>ZeroExtension.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> extended by instances of this stereotype represent operations that zero-extend the input bit-block to produce the output bit-block. If the length of the input bit-block is $n$ and the length of the output bit-block is $m$ , such that $m \geq n$ , then the $m - n$ most significant bits of the output bit-block are all set to zero.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every zero-extension operation must have a single input operand and a single output operand.</li> <li>2. The operands of every zero-extension operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>3. The length of the input bit-block must be less than or equal to the length of the output bit-block.</li> </ol>
<b>Concrete syntax:</b>	<b>000...</b>

<b>Name:</b>	<b>Concatenation.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> extended by instances of this stereotype represent operations that join their $n$ input bit-blocks to produce a single output bit-block. The length of the output bit-block is given by the multiplicity of the single output operand. The lengths of the input bit-blocks are given by the multiplicities of the input operands.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every concatenation operation must have at least one input operand and a single output operand.</li> <li>2. The operands of every concatenation operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>3. The sum of the lengths of the bit-blocks at the input operands must be equal to the length of the bit-block at the input operand.</li> <li>4. The input operands must be indexed according to an increasing sequence of integers starting at 1. This indexing of operands allows the concatenation operation to identify the order in which the input operands are concatenated to generate the output operand.</li> </ol>
<b>Concrete syntax:</b>	

<b>Name:</b>	<b>Extraction.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> extended by instances of this stereotype represent operations that copies a segment of the input bit-block to the output bit-block. The length of the output bit-block is given by the multiplicity of the single output operand. The length of the input bit-block is given by the multiplicity of the input operand.
<b>Tagged attributes:</b>	<ul style="list-style-type: none"> <li>• <i>lowerIndex</i>. Indicates the position of the least significant bit of the block to extract.</li> <li>• <i>upperIndex</i>. Indicates the position of the most significant bit of the block to extract.</li> </ul>
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every extraction operation must have a single input operand and a single output operand.</li> <li>2. The operands of every extraction operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>3. The tagged values must meet that <math>0 \leq lowerIndex \leq upperIndex</math> and upperIndex must be less than or equal to the length of the input bit-block.</li> <li>4. The length of the output bit-block must be equal to <math>upperIndex - lowerIndex + 1</math>.</li> </ol>
<b>Concrete syntax:</b>	<pre> 1011010   ~~~~~ 1101 </pre>

<b>Name:</b>	<b>ShiftRotate.</b>
<b>Is abstract:</b>	Yes.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>BBOperation.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	This stereotype defines the objects in an activity model affected by BitBlockFlow that extend instances of the meta-class <b>OpaqueAction</b> to represent shift/rotate operations.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. Every shift or rotate operation must have two input operands and one output operand.</li> <li>2. The operands of every shift or rotate operation must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>3. One of the input operands of a shift or rotate operation must be connected to a modeling element specifying an integer constant, and the other input operand must specify the source bit-block. The integer constant indicates the number of bits the source operand is shifted or rotated.</li> <li>4. Every shift or rotate operation must produce output bit-blocks whose length is greater than or equal to the length of every input operand.</li> </ol>
<b>Concrete syntax:</b>	This is an abstract stereotype. The subclasses of this stereotype may define new concrete syntaxes for the instances of <b>OpaqueAction</b> .

<b>Name:</b>	Sll.
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>ShiftRotate.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent shift-left logical operations on the input bit-block by the number of positions given by the operand connected to an integer literal.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	<<


<b>Name:</b>	Srl.
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>ShiftRotate.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent shift-right logical operations on the input bit-block by the number of positions given by the operand connected to an integer literal.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	>>

<b>Name:</b>	Rol.
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>ShiftRotate.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent rotate-left operations on the input bit-block by the number of positions given by the operand connected to an integer literal.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	<<<

<b>Name:</b>	<b>Ror.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	<b>ShiftRotate.</b>
<b>Extends:</b>	<b>OpaqueAction.</b>
<b>Description:</b>	The instances of <b>OpaqueAction</b> in an activity model extended by instances of this stereotype represent rotate-right operations on the input bit-block by the number of positions given by the operand connected to an integer literal.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	The constraints for this stereotype are inherited from stereotype <b>LogicOperation.</b>
<b>Concrete syntax:</b>	<b>&gt;&gt;&gt;</b>

<b>Name:</b>	<b>BBModuleCall.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Operations.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>CallBehaviorAction.</b>
<b>Description:</b>	The instances of <b>CallBehaviorAction</b> in an activity model extended by instances of this stereotype invoke other modules. This operation plays a key role in the definition of hierarchical and complex modules.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. The operands of every module call must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter.</b></li> <li>2. The operands of every module call must correspond one-to-one to the parameters of the module invoked.</li> <li>3. Every module call must be synchronous, which means that the the caller module waits for completion of the module called.</li> <li>4. The module called by a call operation must fulfill the restrictions specified for this modeling construct.</li> <li>5. Every module call must be identified by a non-empty string different from the identifier assigned to the module called.</li> </ol>
<b>Concrete syntax:</b>	The same as the one defined by the extended meta-class.



<b>Name:</b>	<b>BBSwitch.</b>
<b>Is abstract:</b>	No.
<b>Package:</b>	Controls.
<b>Superclass:</b>	None.
<b>Extends:</b>	<b>CallBehaviorAction.</b>
<b>Description:</b>	The instances of <b>CallBehaviorAction</b> in an activity model extended by instances of this stereotype select one of multiple bit-blocks at the input operands based on the current state of the algorithm. A switch invokes the behavior described by the associated state machine. When the state of the algorithm changes, the switches invoke their state machines, which select the appropriate input depending on the current state and change to the next state. During transition to the next state, the algorithm carries out all of its operations using the values provided by the switches and computes the results.
<b>Tagged attributes:</b>	None.
<b>Constraints:</b>	<ol style="list-style-type: none"> <li>1. The operands of every switch must meet the restrictions related to the number of bit-blocks processed, and the number of dataflows they are connected to, defined for the stereotype <b>BBParameter</b>.</li> <li>2. The operands of every switch must correspond one-to-one to the parameters of the state machine invoked.</li> <li>3. Every switch must be synchronous, which means that the the caller module waits for completion of the invoked state machine.</li> <li>4. The state machine invoked by a switch must fulfill the restrictions specified for this modeling construct.</li> <li>5. Every switch must be identified by a non-empty string different from the identifier assigned to the state machine invoked.</li> </ol>
<b>Concrete syntax:</b>	

# Appendix C

## Modified version of the grammar of VHDL

The following is the list of production rules that describe the sub-set of VHDL generated by the transformation described in this dissertation. The rules are described in Backus-Naur Form (BNF), with the initial non-terminal symbol being *design\_file*.

```
<design_file> ::= <context_clause> <library_unit>
<context_clause> ::= <library_clause> <use_clause>
<library_clause> ::= 'library' <logical_name_list> ';'
<logical_name_list> ::= <logical_name> { ',' <logical_name> }
<logical_name> ::= <identifier>
<use_clause> ::= 'use' <selected_name> { ',' <selected_name> } ';'
<selected_name> ::= <prefix> '.' <suffix>
<prefix> ::= <identifier>
           | <selected_name>
<suffix> ::= <identifier>
           | 'all'
<library_unit> ::= <primary_unit> <secondary_unit>
```

$\langle \text{primary\_unit} \rangle ::= \langle \text{entity\_declaration} \rangle$   
 $\langle \text{entity\_declaration} \rangle ::= \text{'entity' } \langle \text{identifier} \rangle \text{'is'}$   
 $\quad \langle \text{port\_clause} \rangle$   
 $\quad \text{'begin'}$   
 $\quad \text{'end' 'entity' } \langle \text{identifier} \rangle \text{';'}$   
 $\langle \text{port\_clause} \rangle ::= \text{'port' '(' } \langle \text{port\_list} \rangle \text{' )' ';'}$   
 $\langle \text{port\_list} \rangle ::= \langle \text{interface\_signal\_declaration} \rangle$   
 $\quad \{ \text{';' } \langle \text{interface\_signal\_declaration} \rangle \}$   
 $\langle \text{interface\_signal\_declaration} \rangle ::= \langle \text{identifier\_list} \rangle \text{' : ' } \langle \text{mode} \rangle \langle \text{subtype\_indication} \rangle$   
 $\langle \text{identifier\_list} \rangle ::= \langle \text{identifier} \rangle \{ \text{' , ' } \langle \text{identifier} \rangle \}$   
 $\langle \text{mode} \rangle ::= \text{'in'}$   
 $\quad | \text{'out'}$   
 $\quad | \text{'inout'}$   
 $\langle \text{subtype\_indication} \rangle ::= \langle \text{simple\_name} \rangle$   
 $\quad | \langle \text{slice\_name} \rangle$   
 $\langle \text{simple\_name} \rangle ::= \langle \text{identifier} \rangle$   
 $\langle \text{slice\_name} \rangle ::= \langle \text{simple\_name} \rangle \text{'(' } \langle \text{range} \rangle \text{' )'}$   
 $\langle \text{range} \rangle ::= \langle \text{integer} \rangle \langle \text{direction} \rangle \langle \text{integer} \rangle$   
 $\langle \text{direction} \rangle ::= \text{'to'}$   
 $\quad | \text{'downto'}$   
 $\langle \text{secondary\_unit} \rangle ::= \langle \text{architecture\_body} \rangle$   
 $\langle \text{architecture\_body} \rangle ::= \text{'architecture' } \langle \text{identifier} \rangle \text{'of' } \langle \text{identifier} \rangle \text{'is'}$   
 $\quad \langle \text{architecture\_declarative\_part} \rangle$   
 $\quad \text{'begin'}$   
 $\quad \langle \text{architecture\_statement\_part} \rangle$   
 $\quad \text{'end' 'architecture' } \langle \text{identifier} \rangle \text{';'}$   
 $\langle \text{architecture\_declarative\_part} \rangle ::= \langle \text{component\_declaration} \rangle \langle \text{type\_declaration} \rangle$   
 $\quad \langle \text{signal\_declaration} \rangle$

$\langle \text{component\_declaration} \rangle ::= \text{'component' } \langle \text{identifier} \rangle \text{'is'}$   
 $\qquad \qquad \qquad \langle \text{port\_clause} \rangle$   
 $\qquad \qquad \qquad \text{'end' 'component' } \langle \text{identifier} \rangle \text{' ;'}$

$\langle \text{type\_declaration} \rangle ::= \text{'type' } \langle \text{identifier} \rangle \text{'is' } \langle \text{enumeration\_type\_definition} \rangle \text{' ;'}$

$\langle \text{enumeration\_type\_definition} \rangle ::= \text{'(' } \langle \text{enumeration\_literal} \rangle \{ \text{' , ' } \langle \text{enumeration\_literal} \rangle \}$   
 $\qquad \qquad \qquad \text{' )'}$

$\langle \text{enumeration\_literal} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{signal\_declaration} \rangle ::= \text{'signal' } \langle \text{identifier\_list} \rangle \text{' : ' } \langle \text{subtype\_indication} \rangle \text{' ;'}$

$\langle \text{architecture\_statement\_part} \rangle ::= \{ \langle \text{concurrent\_statement} \rangle \}$

$\langle \text{concurrent\_statement} \rangle ::= \langle \text{process\_statement} \rangle$   
 $\qquad \qquad \qquad | \langle \text{component\_instantiation\_statement} \rangle$   
 $\qquad \qquad \qquad | \langle \text{concurrent\_signal\_assignment\_statement} \rangle$

$\langle \text{process\_statement} \rangle ::= \langle \text{identifier} \rangle \text{' : ' 'process' ' ( ' } \langle \text{sensitivity\_list} \rangle \text{' ) ' 'is'}$   
 $\qquad \qquad \qquad \text{'begin'}$   
 $\qquad \qquad \qquad \langle \text{case\_statement} \rangle$   
 $\qquad \qquad \qquad \text{'end' 'process' } \langle \text{identifier} \rangle \text{' ;'}$

$\langle \text{sensitivity\_list} \rangle ::= \langle \text{identifier} \rangle \{ \text{' , ' } \langle \text{identifier} \rangle \}$

$\langle \text{case\_statement} \rangle ::= \text{'case' } \langle \text{identifier} \rangle \text{'is'}$   
 $\qquad \qquad \qquad \langle \text{case\_statement\_alternative} \rangle$   
 $\qquad \qquad \qquad \{ \langle \text{case\_statement\_alternative} \rangle \}$   
 $\qquad \qquad \qquad \text{'end' 'case' ' ;'}$

$\langle \text{case\_statement\_alternative} \rangle ::= \text{'when' } \langle \text{choice} \rangle \text{' => ' } \langle \text{signal\_assignment\_statement} \rangle$

$\langle \text{choice} \rangle ::= \langle \text{bit\_string\_literal} \rangle$   
 $\qquad \qquad \qquad | \text{'others'}$

$\langle \text{bit\_string\_literal} \rangle ::= \text{'B' ' " ' } \langle \text{bit\_value} \rangle \text{' " '}$

$\langle \text{bit\_value} \rangle ::= \langle \text{binary\_digit} \rangle \{ \langle \text{binary\_digit} \rangle \}$

$\langle \text{binary\_digit} \rangle ::= \text{'1'}$   
 $\qquad \qquad \qquad | \text{'0'}$

$\langle \text{signal\_assignment\_statement} \rangle ::= \langle \text{identifier} \rangle \text{'<='} \langle \text{identifier} \rangle \text{' ;'}$   
 $\langle \text{component\_instantiation\_statement} \rangle ::= \langle \text{identifier} \rangle \text{' : ' 'component' } \langle \text{port\_map\_aspect} \rangle \text{' ;'}$   
 $\langle \text{port\_map\_aspect} \rangle ::= \text{'port' 'map' '(' } \langle \text{port\_association\_list} \rangle \text{' )'}$   
 $\langle \text{port\_association\_list} \rangle ::= \langle \text{association\_element} \rangle \{ \text{' , ' } \langle \text{association\_element} \rangle \}$   
 $\langle \text{association\_element} \rangle ::= \langle \text{port\_name} \rangle \text{' => ' } \langle \text{signal\_name} \rangle$   
 $\langle \text{port\_name} \rangle ::= \langle \text{identifier} \rangle$   
 $\langle \text{signal\_name} \rangle ::= \langle \text{identifier} \rangle$   
 $\langle \text{concurrent\_signal\_assignment\_statement} \rangle ::= \langle \text{identifier} \rangle \text{' <= ' } \langle \text{waveform} \rangle \text{' ;'}$   
 $\langle \text{waveform} \rangle ::= \langle \text{unconditional\_expression} \rangle$   
 $\quad \quad \quad | \langle \text{conditional\_expression} \rangle$   
 $\langle \text{unconditional\_expression} \rangle ::= \langle \text{add\_expression} \rangle$   
 $\quad \quad \quad | \langle \text{logic\_expression} \rangle$   
 $\quad \quad \quad | \langle \text{shift\_expression} \rangle$   
 $\langle \text{add\_expression} \rangle ::= \langle \text{term} \rangle \text{' \& ' } \langle \text{term} \rangle$   
 $\langle \text{term} \rangle ::= \langle \text{identifier} \rangle$   
 $\quad \quad \quad | \langle \text{bit\_string\_literal} \rangle$   
 $\langle \text{logic\_expression} \rangle ::= \langle \text{term} \rangle [ \langle \text{logic\_operator} \rangle \langle \text{term} \rangle ]$   
 $\langle \text{logic\_operator} \rangle ::= \text{'and'}$   
 $\quad \quad \quad | \text{'or'}$   
 $\quad \quad \quad | \text{'xor'}$   
 $\quad \quad \quad | \text{'nand'}$   
 $\quad \quad \quad | \text{'nor'}$   
 $\quad \quad \quad | \text{'xnor'}$   
 $\langle \text{shift\_expression} \rangle ::= \langle \text{simple\_expression} \rangle \langle \text{shift\_operator} \rangle \langle \text{integer} \rangle$   
 $\langle \text{shift\_operator} \rangle ::= \text{'sll'}$   
 $\quad \quad \quad | \text{'srl'}$   
 $\quad \quad \quad | \text{'sla'}$

| 'sra'  
| 'rol'  
| 'ror'

$\langle \text{conditional\_expression} \rangle ::= \langle \text{identifier} \rangle \text{ 'when' } \langle \text{condition} \rangle$

# Appendix D

## List of acronyms

### 3

**3GPP** Third Generation Partnership Program

### A

**Alf** Action Language for fUML

**ASIC** Application-Specific Integrated Circuit

### B

**BNF** Backus-Naur Form

### C

**CORBA** Common Object Request Broker Architecture

### D

**DCS** Digital Communications System

**DSML** Domain-Specific Modeling Language

### E

**EDA** Electronic Design Automation

**EJB** Enterprise JavaBeans

**ESL** Electronic System Level

**F**

**FPGA** Field Programmable Gate Array

**fUML** Foundational UML

**G**

**Gbps** Giga-bits per second

**H**

**HDL** Hardware Description Language

**I**

**IDE** Integrated Development Environment

**IP** Intellectual Property

**ISO** International Organization for Standardization

**ISQED** International Society for Quality Electronic Design

**K**

**KOA** Karatsuba-Ofman Algorithm

**L**

**LFSR** Linear Feedback Shift Register

**LSI** Large-Scale Integration

**M**

**MBD** Model-Based Development



**Mbps** Mega-bits per second

**MDA** Model-Driven architecture

**MDE** Model-Driven Engineering

**MOF** Meta-Object Facility

**MOFM2T** MOF Model to Text Transformation

**MSI** Medium-Scale Integration

**O**

**OCL** Object Constraint Language

**OMG** Object Management Group

**P**

**PIM** Platform-Independent Model

**PSM** Platform-Specific Model

**PSP** Personal Software Process

**Q**

**QVT** Queries/View/Transformation

**R**

**RTL** Register-Transfer Level

**S**

**SMDL** Statechart Description Language

**SoC** System-on-a-Chip

**U**

**UML 2** Unified Modeling Language version 2

**V**

**VHDL** Very High Speed Integrated Circuit Hardware Description Language

**VLSI** Very Large Scale Integration

**X**

**XMI** XML Metadata Interchange

**XML** Extensible Markup Language