TUCS

Dragoş Truşcan

Model Driven Development of Programmable Architectures

Turku Centre *for* Computer Science

TUCS Dissertations
No 84, March 2007

Model Driven Development of Programmable Architectures

Dragoş Truşcan

To be presented – with the permission of the Faculty of Technology at Åbo Akademi University – for public criticisms in Auditorium Gamma at ICT-house, Turku, Finland, on March 16, 2007, at 12 noon.

Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5 B
FIN-20520 Turku, Finland

2007

Supervisor

Professor Johan Lilius
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3-5 B
FIN-20520 Turku
Finland

Reviewers

Professor Tarja Systä
Institute of Software Systems
Tampere University of Technology
P.O.Box 553
FIN-33101 Tampere
Finland

Dr. Bernhard Schätz
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
D-85748 Garching
Germany

Opponent

Dr. Bernhard Schätz
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
D-85748 Garching
Germany

ISBN 978-952-12-1858-3
ISSN 1239-1883
Painosalama Oy – Turku, Finland 2007

To my wife and to my parents

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I start with the supervisor of my PhD studies, Professor Johan Lilius, who with his patience, enthusiasm, and innovative suggestions made this thesis come true. I would also like to thank him for the support and advice that he has offered me throughout my doctoral studies not only in professional, but also in personal matters.

I would also like to thank Professor Tarja Systä, Tampere University of Technology and Dr. Bernhard Schätz, Technische Universität München for reviewing the thesis and for providing me with comments and suggestions for improving this work. Furthermore, I wish to sincerely thank Dr. Bernhard Schätz for accepting to act as opponent at the disputation of this thesis.

My doctoral research has been financially supported by the Turku Centre for Computer Science (TUCS) and by the Center for Reliable Software Technology (CREST) to which I am sincerely grateful. I am also grateful to the leaderships of these institutions (especially to Academy Professor Ralph-Johan Back), and to their technical personnel for providing a high-level research environment and excellent working conditions. I thank the leaders of the CREST laboratories – Professor Kaisa Sere (Distributed Systems Lab.), Academy Professor Ralph-Johan Back and Acting Professor Ivan Porres (Software Construction Lab.), and Professor Johan Lilius (Embedded Systems Lab.) – for providing a high scientific competence within CREST. In addition, I would like to express my gratitude to the HPY and TES research foundations for awarding me grants in support of my research.

Several persons in TUCS and elsewhere have positively influenced my research. I wish to sincerely thank Dr. Seppo Virtanen for fruitful discussions and collaborations on different research topics, and for various kinds of useful information regarding the Finnish language and culture. Assistant Professor João Miguel Fernandes has played an important contribution in my research work by sharing with me his methodical thinking, passion for science and warm attitude. Acting Professor Ivan Porres has also been a catalyst of my doctoral research through his examples of professionalism and efficiency. I am also grateful to Marcus Alanen who throughout my years at TUCS has kindly assisted me in Linux and Coral related technical issues. I wish to thank all these aforementioned researchers for their collaboration on different projects and for coauthoring papers with me.

Special thanks go to the vivid team of the MICAS project. I am grateful to Kim Sandström (Nokia Research Center) for the interesting discussions we had during MICAS meetings in Helsinki and for the practical-oriented perspective on industrial embedded systems he offered me. I thank Johan and Ivan for the freedom they granted me during the project. I also thank the happy *T&T team* (MSc students Torbjörn Lundkvist and Tomas Lillqvist) for their excellent contribution to the MICAS project, and for their collegial and friendly attitude.

The united and enjoyable atmosphere within the Embedded System Laboratory has also had a great influence on my research, especially through its weekly coffee meetings where lively discussions took place on the most diverse topics. I wish to thank in particular to my colleagues Sebastien Lafond and Dag Björklund for our discussions, as well as to Robert Gyllenberg for all the technical and non-technical assistance he provided every time I asked for.

I am grateful to Cristina and Tiberiu Seceleanu, and to João Miguel Fernandes for proofreading this thesis and for their accurate comments. In addition, I am deeply grateful to all my friends that, in one way or another, helped and encouraged me to bring this thesis to an end: Cristina Seceleanu, Tiberiu Seceleanu, Tanja Mărculescu, Ovidiu Mărculescu, Marilena Aldea, Ion Durac, Daniela Durac, Dorina Mărghescu, Luigia Petre, Ion Petre, Diana Preoteasa, Viorel Preoteasa, Elena Czeisler, Eugen Czeisler, Marius Codrea, Mihnea Pretorian and Ştefan Iliescu.

My parents, Maria and Gavril, deserve my warmest appreciation for always encouraging me to enrich my education, and for providing me with their overwhelming love and support. The same amount of gratitude goes to my in-laws Mariana, Mircea, Daniela, Simona and Philippe for their affection and support. I am also deeply grateful to my god-parents, Constanţa and Alexandru, for their constant attempt to cultivate my interest in history, religion, and literature, as well as for their affection and friendship.

Unlimited appreciation and gratitude go to my wife Alina, who has been standing by me at all times throughout this journey. I wish to thank her for her unconditional love and support, for her patience and good advice in hard moments, and most of all, for being my best friend.

February 20, 2007 Dragoş Truşcan
Turku

Abstract

Addressing the conflicting requirements of embedded systems has become, in recent years, an important challenge for designers. On the one hand, there is a need for shorter time-to-market, less costly development cycles for the new products, a goal that has been traditionally achieved using *general purpose processors* (GPPs). Although they can be easily programmed to accomplish specific tasks, GPPs provide low performance as well as high power consumption and large silicon area. On the other hand, the increasing demands for functionality and performance of the current applications require the use of dedicated hardware circuits to boost the performance of the system, while providing low power consumption and blueprint area. However, hardware-based solutions are limited by the difficulty in designing, debugging, and upgrading them.

As a compromise solution, *programmable architectures* have emerged as a flexible, high performance, and cost effective alternative for embedded applications, tailored to process application-specific tasks in an optimized manner. Such architectures are biased to meet specific performance requirements by using dedicated *processing elements* implemented in hardware. A *controller* is used to drive the activity of these processing elements. Using the program code running on the controller, programmable architectures may be programmed to serve several applications. Thus, they provide a good solution for complex applications, where flexibility is needed not only to accommodate design errors, but also to upgrade the specifications.

The main challenge in designing programmable architectures is in finding an appropriate solution to serve as a good compromise between a GPP and a hardware-based implementation. The continuously increasing complexity of the application requirements puts additional pressure on the design process. Thus, new techniques have to be employed to tackle the complexity of both software and hardware specifications by providing abstraction levels, tool support and automation of the development process. Such techniques will shorten the time-to-market and reduce the development costs of new products.

Recently, a new paradigm has gained momentum in the software engineering field. The *model driven paradigm* promotes the use of graphical *models* to specify the software at various abstraction levels, and of *model transformations* to refine the system from abstract specifications to concrete ones.

In this thesis, we employ the concepts and tools of the model driven paradigm as a development framework for programmable architectures. Since programmable architectures are a consistent combination of hardware and software, issues specific to each of the them need to be addressed. As such, we devise a methodology that enables the development of the software and hardware specifications in isolation and, when a certain level of detail is reached, the specifications of the two are integrated (mapped). Throughout our methodology we adopt the *Unified Modeling Language* (UML) as the main modeling language for the application and for the architecture. Through this choice, we intend to take advantage of already available UML tool support, not only for graphically editing the specifications, but also for providing automation of mechanical, repeating design activities.

Two alternative UML-based approaches for the specification of applications targeted to programmable architectures are discussed. The main emphasis of these approaches is on identifying the application functionality that needs to be supported by dedicated resources of the target architecture. The application types that we address here are in the range of data processing applications like protocol and multimedia processing. An IPv6 router case study is used to exemplify these approaches.

At architecture level, we employ model driven concepts to define *domain specific languages* (DSLs) for programmable architectures. These DSLs are intended to provide graphical capabilities and increased abstraction levels (including *programming models*) for modeling the two specific programmable architectures that we address in this thesis. An IPv6 router and a multimedia processing application are used as case studies on the selected architectures, respectively.

Having the specifications of both the application and of the architecture in place, we also look at the process of mapping the application on a selected architecture. The process is again placed in a model driven context in order to enable tool support and automation. The output of the mapping process is a *configuration* of the architecture for the given application and the *program code* that will implement the application on the architecture.

Finally, we show that by taking advantage of the concepts and tools of the model driven paradigm we can rapidly generate different perspectives (simulation, estimation, synthesis) of a system in an automated manner. The generated artifacts are used as input for the simulation, estimation, design space exploration and synthesis processes, respectively, of the system under design.

“All models are wrong, but some of them are useful”
(George Box)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An *embedded system* is a special-purpose computing system that is imbricated (i.e., embedded) in a physical device that it controls. In the last decade, such systems have become an overwhelming presence in all application areas, ranging from home appliances, office and hospital equipment to mobile phones, automobiles and airplanes.

Addressing the conflicting requirements of embedded systems has become, in recent years, an important challenge for designers. On the one hand, there is a need for shorter time-to-market, less costly development cycles for the new products, a goal that has been traditionally achieved by using *general purpose processors* (GPPs). Although they can be easily programmed to accomplish specific tasks, GPPs provide low performance as well as high power consumption and large silicon area. On the other hand, the increasing demands for functionality and performance of the current applications require the use of dedicated hardware circuits to boost the performance of the system. Consequently, extremely complicated *application specific integrated circuits* (ASICs) started to be developed. An ASIC is an integrated circuit customized for a specific problem. Being a hardware-based solution, it provides high performance, and low power consumption and blueprint area. The drawback is that their design process is expensive, and once the final product is obtained it cannot be modified anymore. As such, ASICs are limited by the difficulty in designing, debugging and upgrading them.

1.1 Programmable Architectures

As a compromise solution, *application specific instruction set processors* (ASIPs) have emerged as a flexible, high performance and cost effective alternative for embedded applications, tailored to process application-specific tasks in an optimized manner. ASIPs are biased to meet specific performance requirements by using dedicated *processing elements* implemented in hardware, in order to support the tasks demanding high performance. A *controller* is used to drive the activity of these processing elements. Using the program code running on this controller, ASIPs may be programmed to serve several applications. Thus, they provide a good solution for complex applications, where flexibility is needed not only to accommodate design errors, but also to upgrade the specifications [31].

There are several terms used when referring to ASIPs: *programmable embedded systems*, *programmable platforms*, *programmable architectures* or *heterogenous architectures*. Throughout this study we will use the terms *ASIP* and *programmable architecture* interchangeably. In the context of this thesis, the term *architecture* refers to an abstract representation of a hardware device. By being *programmable* the functionality of this device may be modified to serve application specific purposes.

Being an optimized solution for a given application, or for a set of applications, ASIPs provide improved performance as compared to the general purpose processors. According to some researchers [72], this increase can be of 2 to 100 times. Nevertheless, since ASIPs are not fully hardware-based solutions they can be several times slower as compared to a corresponding ASIC solution. Even with this drawback, the payoff is far greater in terms of flexibility and upgradability. Programmable architectures bring important benefits like rapid time-to-market, flexibility of design and, consequently, an increased product life-time and upgradability.

The main challenge in designing ASIPs is finding an appropriate architecture to serve as a good compromise between a GPP and an ASIC. Ideally, the architecture should be optimized for a family of applications, such that more than one application can be served by the same ASIP. Additional aspects that enable the successful development of ASIPs are discussed in [53].

There are basically two categories of approaches used for embedded system design: *top-down* and *meet-in-the-middle*. In the *top-down* approach [49, 86, 118], a high-level system specification is gradually refined towards implementation. When enough detail is gathered, the refined specification is partitioned into hardware and software, and co-designed. Such approaches are also known as *hw/sw co-design*. The top-down approaches are typically based on a *model of computation* (MOC), in which notions of formal semantics for communication and concurrency are defined. The methods in this category focus on the properties of the application and on the simulation of the system as a whole, but generally the implementation obtained is less efficient for families of applications and for applications using several distinct algorithms. An overview of top-down approaches may be found in [139].

In the *meet-in-the-middle* approaches, the application and the architecture are developed independently. A top-down design flow is used to specify the application. The architecture is developed following a *bottom-up* flow, in which the functionality it provides is identified starting from its hardware layer. When both specifications are complete, the application is mapped onto the architecture. Such an approach enables the reuse of software and hardware, reducing development times and design effort. The price to pay is the significant effort in designing complex libraries, since any new hardware component has to be designed from scratch.

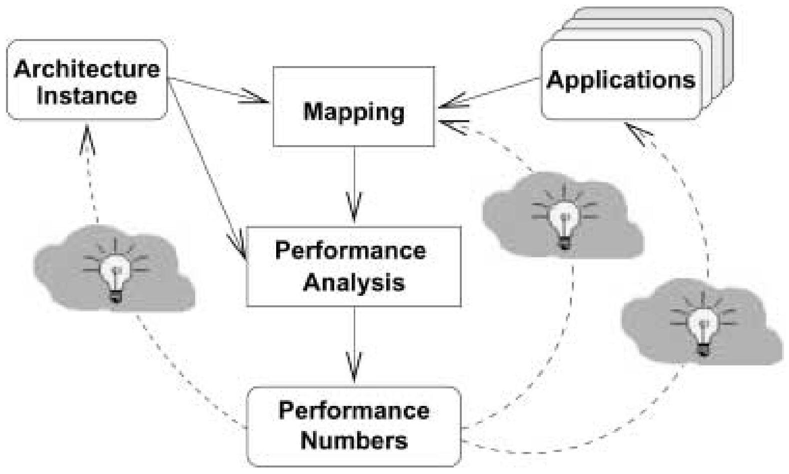


Figure 1.1: The Y-chart approach [74]

In this thesis, we adopt the second category to develop programmable architectures. Our choice is justified by two reasons: the architectures that we address promote the reuse of *intellectual property* (IP) components at different levels of abstraction, and such architectures are intended to be used as an implementation platform for several applications of the same family.

The Y-chart approach. The *Y-chart* approach [73, 74] is a generic framework for designing programmable architectures, in which the architecture is tuned to provide the performance required by a set of applications. Being based on a meet-in-the-middle flow, the Y-chart approach promotes the idea of a clear distinction between the application and the architecture, each being developed independently. The implementation of the application onto an instance of the architecture (i.e., a *configuration*) is done through a *mapping* process. The general view of the approach is shown in Figure 1.1. The application running on a given architecture instance is obtained through the mapping process and the performance of the resulting implementation is evaluated in the *Performance Analysis* step.

All three main artifacts of the approach (i.e., application, architecture and mapping) are considered to be equally important in achieving an optimal configuration of the architecture. Therefore, although the *performance numbers* aim mainly at suggesting improvements of the proposed configuration(s), the other two artifacts may also be targeted. For instance, some of the algorithms may be optimized at application-level, or the mapping process modified based on certain heuristics. The mapping process is performed iteratively until satisfactory performance numbers are obtained. The process of trying out different architectural configurations to find "the optimal" one is also known as *design space exploration*.

1.2 The Model Driven Paradigm

Due to the increasing complexity of computing systems and to the diversification of tasks that they need to perform, novel system specification techniques became necessary. Raising the level of abstraction at which the systems are programmed is one solution. Abstracting computer systems has been a recurring issue over the years. If computers were mainly programmed using machine language in the 1960s, high-level programming languages like C, Fortran and Pascal were developed and used in the 1970s. Still, this was not enough due to the increasing complexity of the specifications, and consequently, a revolutionary paradigm appeared in the next decade. The paradigm was based on a simple observation: *"Everything is an object"*. As a result, the *object-oriented* view was adopted on a large scale and became widely used in the software industry. Nevertheless, the abstraction level provided by the object-orientation reached its limits too; it is not able to cope with the high complexity of the software any more. Consequently, a new paradigm shift took place: *"Everything is a model"*. The idea was not completely new. Since the mid 1970s, fields like domain engineering and database systems became aware of the necessity of using models as first-class citizens during system specification.

The model driven paradigm caught ground rapidly in the software engineering field and, in consequence, the shift from textual object-oriented languages like C++ and Java towards *modeling languages* was quite natural. This new category of languages provide higher level of abstraction and benefit from a visual modeling environment, for system specification. One such language is the *Unified Modeling Language* (UML) [94]. UML provides a collection of graphical notations (i.e., diagrams) to be used for specifying various aspects of a software system. Although relying on an object-oriented basis, UML is intended to be a general-purpose modeling language. It can be applied to a wide range of application domains, primarily to those that are object-oriented or component-based in nature. One important feature of UML is that it allows for the definition of *domain specific languages* (DSL), or in the UML terminology *profiles*, using its extensibility mechanism (i.e., *stereotypes*, *tagged values* and *constraints*). Although at the time of publishing this thesis version 2.0 of UML has been officially adopted, the research discussed in this thesis has been performed with respect to version 1.4 of UML. However, with some adaptations, the research results discussed in this study may be extended and applied also to UML 2.0.

The *Model Driven Architecture* (MDA) [93], as promoted by the Object Management Group (OMG), has been, since the very beginning, the driving force behind the new mentioned paradigm shift. The underlying principle of MDA is moving the focus of design from implementation concerns to solution modeling. MDA proposes the use of *models* and *model transformations* as the main artifacts of evolving the system specification from requirements to implementation. UML has been proposed as the default modeling language of MDA. Additional UML-independent modeling languages (i.e., *metamodels*) can be defined and used within MDA, as we will discuss in Chapter 2.

In a recent panel discussion [104], it was argued that MDA had reached a certain level of maturity for software development. However, there are still many issues waiting to be addressed regarding the development of the *system*, as a combination of hardware and software. In this thesis, we intend to take a small step further, towards investigating some of these issues in the context of programmable architectures.

1.3 Contributions of this Thesis

Embedded systems are a consistent combination of software and hardware, in which both artifacts are equally important during the development process. Typically, different specification techniques and tools are used for each of them, creating a discrepancy in the way the system specification is perceived as a whole. We consider that common specification techniques should be applied both for application and architecture, yet adapted to the specific nature of each of them, in order to capture their particular details. Similar proposals have been made by other researchers [35, 42, 111].

The current thesis presents a model driven methodology for the development of embedded systems and, in particular, of programmable architectures. Throughout this work, UML is used as modeling language for the systems under consideration.

From a high-level perspective, the contribution of this thesis is three-fold:

- we define a set of techniques to address some of the development issues of embedded applications and programmable architectures, tailored to address specific problem domains like protocol and multimedia processing;
- we use the principles and tools of the model driven paradigm to support the previously defined methodology;
- we provide a starting point for the defined methodologies to be adapted and reused for other application domains.

In the following, we discuss the problems addressed in this thesis, and for each problem statement, we briefly list the solutions provided in this study.

1.3.1 Model Driven Application Specification

Problem. The applications targeted to programmable architectures have been traditionally specified either in machine language, or in a high-level programming language (typically C). In the latter case, the resulting application specification is mapped onto the architecture using specialized tools. Currently, this approach cannot cope with the increasing complexity of specifications anymore, thus more elaborated methods for the application specification are needed. Not only the use of higher *levels of abstraction* is required, but also a systematic *application analysis*.

On the one hand, such techniques allow the designer to focus on the relevant details of the specification at different development stages. On the other hand, they reduce the risk of missing functionality at later phases of the development process, which would delay the development cycle considerable. The optimized hardware is one of the key aspects of ASIPs. The hardware resources of the architecture are the ones providing an increased performance, as compared to a software-based approach. Hence, one of the most important issues in ASIP design is that, from the application specification, one can *identify complex and frequently used processing tasks* to be implemented using dedicated hardware.

Solution. In Chapters 3 and 4, we examine applications in the protocol processing and multimedia processing domains. Such applications are typically data-driven applications, where the main focus is on the order of the computations that affect this data, rather than on the state of the system.

We adopt UML as the modeling language for the application specification. Two systematic methods for analyzing the application are proposed.

In Chapter 3, a UML-based application specification process is presented. The goal of this process is to allow the identification of the system functionality starting from the written requirements. An existing method for functional decomposition of the specification [48] is adopted and complemented to serve our goals. The main contribution of the chapter is that we suggest the combination of collaboration diagrams and activity diagrams for performing the behavioral analysis of specific types of applications. The approach benefits from the hierarchical decomposition of activity diagrams, to identify distinct pieces of functionality of the application. In addition, systematic steps for transforming different models of the system are proposed. Our approach is intended for specific application domains (e.g., data-driven applications), where we are more interested in the sequence of processing tasks that the system performs, rather than in its state at given moment.

In Chapter 4, the combination of UML diagrams and *data-flow diagrams* (DFDs) for application specification is proposed. This research has been motivated by the observation that, in certain situations, UML is not able to adequately represent all the perspectives of a system. Several contributions are presented in this chapter, as follows:

- based on previous work in combining UML and DFD [45], we propose a process for application specification, in which the perspectives provided by DFD and UML are interplayed at different abstraction levels;
- following the same guidelines found in [45], we introduce a systematic approach for transforming DFDs into UML models and vice-versa. Although the proposed approach may look artificial in some situations, the main objective is to provide support for automation;
- by taking advantage of a UML-based tool and its scripting facilities, we also implement model transformations to support the previously mentioned combi-

nations. During this process, we show how both UML and DFD models can be interrogated and transformed, using model scripts.

1.3.2 Model Driven Architecture Specification

Similarly to the software specifications, the hardware specifications are also confronted with increasing complexity. In order to tackle this complexity and to shorten the development cycle of new products, abstractions of the hardware architecture, programming models and appropriate design frameworks are required [131].

We investigate these research goals on two concrete programmable architectures, namely TACO [132] and MICAS [109].

Hardware Abstraction

Problem. Managing the complexity of hardware specifications requires the development of abstract views of hardware implementations. As for software, *abstractions* have to be defined at several levels, starting from logical circuits and continuing with the components of the architecture, each capturing only details relevant to a given step of the development. Similarly to software specifications, the hardware specification techniques gradually increased their level of abstraction over the years. If initially hardware was designed in terms of transistors and logic gates, nowadays, *hardware description languages* (HDLs) (e.g., VHDL, Verilog) and even object-oriented system-level specification languages (e.g., SystemC) have become very popular in industrial environments.

Solution. In this thesis, we examine the use of UML as a *domain specific language* (DSL) for programmable architectures. Such a language takes one step forward in the abstraction hierarchy, placing itself on top of the existing hardware specification languages like SystemC, VHDL, Verilog, etc.; in addition, it provides a visual representation of architecture components. Moreover, we rely on UML-based tools to provide tool support for the defined DSLs. The approach allows us to edit models of the hardware architecture and to provide automation for generating different artifacts from these models.

We suggest the use of UML as a *hardware modeling language*, in order to benefit not only from the graphical UML notations and tools, but also from the model driven principles for abstraction, tool support and automation.

In Chapter 5, we use UML to define a visual DSL for modeling the TACO [132] programmable architecture. The DSL uses UML-based notations, such that it may be used in any UML tool without the need of tool customization. We also show that by taking advantage of appropriate tool support, we can rapidly create configurations of the architecture in a graphical environment, and enforce their architectural consistency. In addition, a component library has been implemented to provide support for automation and reuse, when using the TACO DSL. Several automated transformations are proposed for transforming the TACO models into simulation and synthesis models, respectively, or for rapidly estimating the physical characteristics of the created configurations.

In Chapter 6, we define a DSL for modeling a programmable multimedia architecture, namely MICAS [109]. The contributions of this chapter are as follows:

- the proposed DSL models the specifications of the MICAS hardware architecture at several levels of abstraction. A *conceptual level* is used to focus on the functionality of the components, a *detailed level* to refine the communication mechanisms between components, and an *implementation level* specifies the system in terms of "off-the-shelf" IP blocks;
- several transformations are defined to evolve the hardware specification from one abstraction level to another. Some of these transformations are manual (i.e., based on designer's intervention), others are automatable. In addition, support for generating the *simulation model* of the architecture is provided;
- to enable automation and reuse, several component libraries are proposed, which encode not only components of the hardware architecture, but also design decisions in architecting the system. The elements of the library are expressed using concepts of the MICAS DSL, and benefit from the corresponding tool support;
- we propose a series of customizations of the UML tool that we use to assist the design process.

The hardware architecture of MICAS is simulated using the SystemC [100] language, an extension of C++ for hardware specification. Thus, the hardware perspective of MICAS is transformed into SystemC specifications. In addition, different configuration-related information, which is not directly representable in SystemC, has to be taken into account during simulation. Hence, a C++-based DSL for MICAS is defined, to enable the integration of the generated information within the MICAS simulation environment.

Programming Models

Problem. In recent years, several programmable processors were developed, especially in the area of protocol processing [58]. Soon after, the difficulty in programming them, due to their complex architecture and the variable instruction set, became an obstacle to using them in practice [84]. Consequently, the use of a *programming model* of the architecture has been suggested [71, 74] to provide not only an abstraction of the hardware details, but also a functional view of the architecture. Such a model facilitates programming the architecture by allowing the designer to focus on the functionality that the architecture provides, rather than on the hardware implementation of this functionality.

Solution. In the above context, we investigate the use of UML to specify programming models for the two graphical programmable architectures addressed in this thesis, that is TACO and MICAS.

In Chapter 5, we define a *programming model for the TACO architecture*. For this purpose, we introduce a number of *programming primitives* and map them onto the concepts of the TACO platform. In addition, we express these primitives using UML concepts and include them in the TACO DSL. As we show in the same chapter, the approach enables us to store the programming primitives of TACO in the same library, together with the hardware components of TACO. Moreover, the proposed programming model enables the designer to program the TACO architecture using a higher-level programming language than the one provided by the machine assembly language of TACO.

In Chapter 6, we propose *a programming model for the MICAS architecture*. Similarly to TACO, the MICAS programming model is defined for abstracting the details of the hardware components, and it is embedded within the MICAS DSL. The programming model is defined at two levels of abstraction, each of them focusing on a specific view of the architecture. Furthermore, the programming model provides a graphical interface that is integrated with the MICAS DSL and consequently, it benefits from a similar tool support.

In order to integrate the MICAS programming model within the MICAS simulation environment, the MICAS C++–based DSL is enhanced to represent also the programming model of the architecture. The approach enables us to obtain the co-simulation model from the MICAS DSL models automatically, in a "push-button" manner.

Design Frameworks

Problem. An optimal tuning of the architecture, with respect to the application requirements can be realized by employing collections of customized tools needed to configure, compile, analyze, optimize, simulate, explore and synthesize architectural configurations. Programmable architectures are usually accompanied by *custom design frameworks* that enable the designer to squeeze the optimal performance out of a given architecture. Such a design framework should provide several features. Out of them, the *Architectural estimation* is one of the key ones. Embedded systems in general, and hardware in particular, imply high cost of designing and manufacturing. Therefore, it is important to be able to estimate the characteristics of the final product as early as possible in the development process. System-level estimation is of particular importance to embedded systems, since, by their definition, they are systems that must comply with tight constraints in terms of size, energy consumption and cost. Based on the estimation results, the architectural design space exploration is performed. During this process, the designer evaluates several architectural configurations and selects the one(s) complying best with the requirements of the application. *Simulation* represents an equally important technique in developing programmable architectures. To prevent and detect inherent errors in the specifications of both the application and the architecture, the simulation has to be performed at different levels of abstraction, with respect to both functional and non-functional requirements of the system. Finally, once the configuration of a given programmable architecture is chosen to implement an application, the *hardware synthesis* enables the specification of the configuration in a hardware description language and also its synthesis, using specific synthesis tools. Furthermore, an important feature of such a design framework is its capability of *rapid generation of different models* (e.g., simulation, synthesis, estimation, etc.) of the system, starting from its DSL models.

Solution. In our research, we examine the possibility that the principles of the UML and of the model driven paradigm are used to implement custom design frameworks for the two programmable architectures under investigation.

In the first part of Chapter 7, the simulation framework of the MICAS architecture is described. There are two main contributions related to this topic:

- we propose an approach that integrates the programming model of the architecture within the MICAS simulation model;
- using the object-oriented mechanisms of SystemC and the C++ DSL of MICAS, we show how to define a fully reusable SystemC component library, which allows the customization of module instances at run-time.

Both contributions have enabled the automated generation of the simulation model of a given MICAS configuration starting from the models of the MICAS DSL.

Targeting the TACO architecture, Chapter 5 suggests the use of the tool support corresponding to the TACO DSL as a design framework. To accomplish this task several steps are taken:

- we propose that the already existent simulation, estimation and synthesis models of TACO (presented in detail in [132]) are included into the TACO DSL. The approach brings two benefits: a) it enriches the expressiveness of the DSL with estimation information that may be used in a UML-based tool, to rapidly estimate the physical characteristics of different configurations; b) it allows one to generate the simulation and synthesis code of a given configuration from the models of the TACO DSL;
- taking advantage of the enhanced TACO DSL and of the inclusion of the new information in the TACO component library, we propose a set of automated transformations that generate both the estimates of the created architectural configurations, and the simulation and synthesis code of each configuration.

The second part of Chapter 7 should be seen as a validation of the TACO design framework proposed in Chapter 5. Thus, we discuss a case study in performing design space exploration of the TACO architecture. The study is intended to show that possessing tools to rapidly create configuration, simulation and estimation models of the TACO architecture, allows us to perform system-level exploration within a relatively short time-frame.

As a general contribution on the model driven architecture specification topic, we attempt to show that graphical modeling languages like UML are suitable to model various abstraction layers of programmable architectures. As such, modeling languages can be employed not only for the specification of different perspectives of the hardware, but also for defining and representing programming models. In addition, we show that by taking advantage of the existing model driven and UML-based tooling environments, one can provide customized design frameworks to assist the work of the designer, not only through model editing capabilities, but also through transformational support.

1.3.3 Model Driven Support for the Mapping Process

Problem. Mapping the application on a given architecture is a well-known issue of ASIP design, due to the conceptual gap between the application and the architecture. The programming model of the architecture was suggested by many researchers [74, 118], as a means to narrow this gap by providing not only a higher level abstraction of the architecture, but also by using concepts similar to the ones of the application specification.

Solution. In this study, we examine the benefits brought by such a programming model, in case of mapping the application specification onto the TACO programmable architecture.

Chapter 5 contributes to this research topic as follows:

- we show that having both the application specification and the programming model of the architectures expressed using similar concepts (as suggested in [74]) facilitates the mapping process. Furthermore, we show that employing the same modeling language (i.e., UML) for both the application specification and for the programming model of the architecture, enables us to use the same UML tool to support the mapping process. The approach is beneficial not only in editing both models, but also in automating the steps of the mapping process through model transformations.
- we define a systematic approach in which the application specification may be mapped onto the specification of the TACO architecture. The mapping process is decomposed into several steps, such that as many of them become automatable.

1.4 Related Work

Based on the topics addressed in this thesis, we can divide the work related to this study into roughly three categories: UML-based methodologies, combination of UML and DFD, and programming models for programmable architectures. For each topic, we have selected for discussion several existing related studies that we consider relevant in our context, without claiming that the list is exhaustive.

1.4.1 UML-based Methodologies

There are many general approaches that address the specification of embedded systems in the context of UML and of the model driven paradigm. These approaches typically propose general-purpose methodologies and profiles for embedded systems, which could also be adopted for the specification of concrete programmable architectures.

These approaches may be classified following several criteria. For instance, some of the approaches focus on the entire development process [29, 43, 83, 114], while others address only specific parts [33, 76, 97, 98]. A different categorization may be obtained if we consider the artifact of the development process that they mainly address: application specification [33, 43, 114], architecture specification [28, 43, 97, 106], or the mapping process [76]. Nevertheless, the approach that most of them adopt is customizing the UML elements to represent different perspectives of the system and to allow the connection of the suggested methodologies with external tools.

Embedded UML [83] is a proposal for a methodology and a profile for specification, design, and verification of embedded real-time systems. The authors propose a synthesis of attractive concepts of existing approaches in the real-time, co-design, platform-based and functional design. The use of *reactive objects* is suggested as a better match for embedded systems as compared to active objects. Reactive objects react to stimuli from the environment and eventually provide a response. The behavior of these objects is specified in terms of state diagrams or code. In addition, extensions of UML are considered for functional encapsulation and composition, communication specification, and performance evaluation during the mapping process. The methodology could be seen more as a conceptual framework for modeling embedded systems, rather than a concrete approach.

The methodology proposed by Chen et al. [29] addresses the network processing application domain. The requirements of the system are captured in a use case diagram with annotated performance requirements. The structure and the behavior of the system are identified from use cases, and the performance requirements are propagated. Interaction diagrams are used to model the interaction of the objects, and the internal behavior of the later is expressed either in terms of state machines or activity diagrams. The methodology follows a functional decomposition of the system specification, until the latter can be expressed in terms of concepts defined by the *UML Platform* profile [28]. The profile defines several abstraction levels of the architecture. The *architecture layer* models the physical hardware of the platform. The *application programming interface* (API) layer models the logical services provided by the architecture. The *application specific programming* layer represents a collection of domain-specific services provided by the platform to implement the application. Concrete UML diagrams are chosen to model each layer, and stereotypes are defined to customize the elements of each diagram. A number of *quality of service* (QoS) parameters like throughput, area, power, size, are defined for the elements of each abstraction layer using UML tagged values. The stereotypes of the UML Platform profile are defined in concordance with the concepts of the Metropolis design framework [16], enabling the translation of UML-based models into the language of the Metropolis framework, where communication refinement, partitioning, simulation and exploration of the architecture are performed. The approach in [29] is similar to the one that we propose in Chapter 3, since it follows a functional decomposition of the system specification. At the moment, our approach does not take into consideration performance requirements. In addition, we propose a systematic method in which the functionality of the application is identified, whereas the above mentioned methodology is more generic. At architecture level, the methodology in [28] resembles our approach in Chapters 5 and 6 in two aspects: it provides several layers of abstraction of hardware (including a programming interface), and the physical properties of the architecture components are modeled using UML tagged values similar to our approach in TACO. However, in our approach the mapping of the application onto architecture is currently performed with respect to the functional requirements of the application without the assistance of specialized exploration tools.

Another design methodology is proposed by De Jong [33]. The methodology starts also with use case diagrams from which the specification of the system expressed in terms of objects and message sequence charts is obtained. The internal behavior of the objects is expressed in SDL [62], to provide an executable specification. The missing aspect of this methodology is the connection with the specification of the architecture. Conceptually, the approach is similar to ours, except that we aim at mapping the resulting specification onto the concepts of selected programmable architecture.

The *UML-RT* [114] defines both a methodology and a profile that are based on the Real-Time Object Oriented Modeling (ROOM) [115]. A *capsule* concept is used for specifying the structure of the system. A capsule is basically an active object, that is, an object that has its own thread of control and may initiate control activities with other objects. The internal behavior of a capsule is represented using statecharts and follows a run-to-completion pattern. The communication between capsules uses a message-passing protocol. The UML-RT profile is mainly intended to capture behavior for simulation and synthesis purposes, and is limited with respect to modeling the architecture and performance of the system. One similarity between the UML-RT and the approach we promote in Chapter 3 stands in the fact that the *capsule diagrams* of UML-RT are similar to the collaboration diagrams we use. The difference is encountered in the way the functionality of a capsule and respectively, of an object is decomposed. In UML-RT a complex capsule may be hierarchically decomposed into several simple capsules whose state machines are combined to implement the behavior of the complex capsule. This approach enforces that the communication between the simple capsules and the external environment is done through the ports of the complex capsule. In our approach, objects may not be nested, instead we allow for an object to be split into several simpler objects, and for their state machine to be refactored. Last, but not least, UML-RT provides an event-driven perspective of the system, whereas we would like to focus on a data-driven perspective that is typically more useful for the data-driven application domains that we address.

The *HASoC* design methodology [43] is an extension of the UML-RT profile. HASoC starts with the use case diagram of the system and gradually identifies the objects (i.e., capsules) in the specification. The resulting specification is then partitioned into hardware and software. In turn, the architecture (i.e., the platform) is modeled in terms of hardware components and hw/sw interfaces that are gradually integrated. Two extensions to the UML-RT are added. The communication between capsules is done using data-streams and the target architecture may be configured based on the application requirements. To address the latter extension, additional notations have been defined to depict if a capsule is to be implemented in software or in hardware. HASoC employs the capsule concept to also model the hardware architecture. Two abstraction levels are used: *software-hardware interface model* and a *hardware architecture model*. However, no prescription is given on how the mapping process is performed. This methodology is closer to our approach presented in Chapter 4, since it models a data-flow view of the specification. In addition, it resembles the abstraction layers we employ in both TACO and MICAS architecture specifications.

The OMG's *Real-time UML* profile or, by its official name, the *UML Profile for Schedulability, Performance and Time* [98], defines notations for building models of real-time systems based on Quality of Service (QoS) parameters. External tools can be used to perform analysis of the models and the results may be reintegrated in these models. However, the profile definition is not accompanied by a suggested methodology, and it is generally considered verbose and difficult to use.

In the *HUT profile* [76], a number of stereotypes are proposed for classifying the concepts of both the application and the architecture. The elements of the profile are intended to be applied onto an existing high-level specification to "mark" its elements before the mapping process. Similarly, the components of the architecture are marked with predefined stereotypes. The mapping process is performed manually by pointing out the dependency between the components of the application and architecture. The profile defines specific tagged values for specifying non-functional properties like priority, memory requirements, of both the application and the architecture. The profile is used only for the mapping stage, and it seems that there is only one level of hierarchy during the mapping.

Recently, OMG has issued a draft of the *UML profile for SoC* [97]. The profile identifies a number of generic architectural concepts that are needed in defining SoCs, like module, controller, port, connector. Additionally, the notion of protocol (to describe behavior) is specified. The profile is intended for system-level architectural specification, at an relatively abstract level for being applied to various kinds of systems. In addition, the elements it defines are mappable one-to-one to the elements of the SystemC language, to ensure a smooth generation of the corresponding SystemC code. Yet, no indication is given on how the architecture is obtained from the application.

Targeting the hardware architecture, a UML profile for SystemC was proposed in [106]. The profile customizes structural and behavioral diagrams of UML to model the functionality expressed by processes and channels in a SystemC specification, from which SystemC code is generated.

Last but not least, although the OCTOPUS methodology [14] has not been defined in the context of UML and MDA, it is, to the best of our knowledge, among the first object-oriented methodologies for embedded software systems capable to evolve the specification from initial requirements to implementation. The methodology has as its main goal the development of the application, whereas a *hardware-wrapper subsystem* is used to identify the communication and the interface between the application and the hardware. Yet, the approach specifies the architecture only at the level of hardware-software interfaces, while the selection of its components is rather ad hoc.

To the best of our knowledge, there are no reports on building custom DSLs for concrete programmable architectures (although several metamodeling tools are available [1]), in either industry or academia, using metamodeling techniques. One possible reason could be the large effort (e.g., time, people, research) required to build such a DSL and also poor support for metamodeling provided by the UML tools. Another reason may also be the fact that usually, when the research is performed in large companies, the publication of the results is restricted by confidentiality rules. Finally, the adoption of modeling techniques, a software-based approach, may not be in agreement with the preferences of hardware designers.

All the presented approaches define a number of general stereotypes that should be applied to model a wide range of embedded systems. Although these stereotypes could be used for modeling a specific programmable architecture (e.g., TACO), the mapping of the profile stereotypes to the architecture components would not necessarily capture the essence of the architecture. For instance, the UML Platform profile [30] and the HUT profile [76] do not make a clear distinction between controllers and dedicated processing elements, which is useful in designing programmable architectures. The UML profile for SoC [97] and the UML SystemC profile [106] are biased towards SystemC code generation, and for this reason they do not provide a sufficient level of abstraction of the architecture.

1.4.2 Combination of UML and DFD

Many authors have already studied the combination of DFDs with object-oriented methods and many approaches have been presented in the literature. In the following, we discuss some of them, which we consider relevant to our work.

The adoption of reverse generated DFDs (i.e., DFDs obtained after interpreting the source code) is proposed for reverse engineering purposes, as the basis for obtaining the objects of a system [50]. The approach is partially automated, requiring, in specific situations, the assistance of a human expert with knowledge of the domain. Again in a reverse engineering context, it is suggested the combined usage of DFDs and *Entity-Relationship Diagrams* (ERDs) to describe the system being modernized [66]. Both these approaches target reverse engineering and are customized for their specific application domain.

Alabiso also proposes the transformation of DFDs into objects [6]. To accomplish the transformation, he proposes the following activities: (1) interpret data, data processes, data stores and external entities in terms of object-oriented concepts; (2) interpret the DFD hierarchy in terms of object decomposition; (3) interpret the meaning of control processes for the object-oriented model; (4) use data decomposition to guide the definition of the object decomposition. This work follows a structural approach in the object-oriented domain, whereas we focus on the functionality of the system.

Another interesting framework is the Functional and Object-Oriented Methodology (FOOM) [119], which is specifically tailored for information systems. The main idea behind FOOM is to use the functional approach, at the analysis phase, for defining user requirements, and the object-oriented approach, at the design phase, for specifying the structure and behavior of the system. In FOOM, the specification of user requirements is accomplished in functional terms by OO-DFDs (a DFD with data stores replaced by classes). In data terms, the same specification is modeled by an initial object-oriented schema, or an Entity-Relationship Diagram (ERD), which is transformed into an initial object-oriented schema. In the design phase, the artifacts of the analysis are used, and detailed object-oriented and behavior schemas are created. These schemas are the input to the implementation phase, where an object-oriented programming language is adopted to create a solution for the system. The approach addresses information systems and, in addition, places data and behavior on the same level, whereas we put behavior first.

In [18], the functionality associated with each use case is described by an E-DFD (an extended version of the traditional DFD) or an activity diagram, with the objective of automatically identifying the objects/classes of a distributed real-time system. E-DFDs are mapped onto a graph and, then a tool automatically partitions the graph, which allows the identification of an object set that constitute the “best” architecture from the design and test points of view. This approach could be considered closer to ours, as it refines the system functionality using DFDs and it proposes a systematic (and automated) approach to transform DFDs into objects.

A collection of transformations between the DFD and UML diagrams has been proposed in [126]. The framework aims at translating legacy systems, modeled with structured methods, into an object-oriented representation. Three levels of abstractions are proposed to transform the specification: 1) a DFD is partially transformed into a use case diagram; 2) the same DFD is transformed into sequence diagrams and state diagrams, in which the data flows become parameters of messages and signals; 3) the ERD is transformed into a class diagram. One weak point of the approach is that a sequence diagram is built for each use case separately and it is not clear how the state diagrams are integrated in the approach. Although the approach is not automated, it provides a set of guidelines for transforming the DFDs into UML diagrams in a systematic manner.

1.4.3 Programming Models for ASIPs

Programmable architectures are customized for serving a specific application domain in an optimal manner. Similarly, their programming models are designed to exploit at maximum the characteristics of a given programmable architecture and thus, they are heavily customized for each architecture in particular. As such, a general classification is difficult to achieve. To the best of our knowledge, there is currently no UML-based programming language for programmable architectures. However, we discuss in the following several programming languages just to give an overview of their general characteristics.

NP-Click [118] is one of the most popular programming models in the academic environment. NP-Click targets network processors, and is currently implemented on the Intel IXP1200 network processor. The programming model provides a graphical interface and uses components communicating through ports, while packets are transferred between components using function calls. In addition, threading and resource sharing are also modeled in NP-Click.

VERA [69] is another academic programming model for routers customized for the Intel IXP1200 network processor. Two main features are worth the attention. The first is the fact that VERA uses three levels of abstraction: one specifying the functions of the application, one for specifying the architecture, and one for specifying mappings between the two, in terms of programming primitives. The second feature is the extensibility of the model, that is, additional functionality can be included.

On a more general level, CAIRN [87] is a programming model that addresses generic programmable architectures (ASIPs), rather than being customized for a specific architecture, like the previous two models already discussed. CAIRN also supports three levels of abstraction (i.e., application, architecture and mapping) and has the main goal of capturing the concurrency of both the application and the architecture. At architecture level, *TIPI* [85], an *architecture description language* (ADL), is used to model the concurrency of different data paths in the architecture. At application level, the application is specified using a model of computation. The mapping process is performed manually, but once the marking is done, model transformations are used to generate the code. The approach relies on the TIPI framework to generate simulation, estimation and synthesis models of the system.

Focusing on the industrial side, we discuss two programming models: Intel MicroACE and Teja C. MicroACE [60] is fully customized for the IXP1200 and integrated in its development environment. The application running on the controller is expressed as a flow of data processed by different *active computing el-*

ements (ACEs). An ACE is a piece of functionality of the application, which is implemented by the microengines of the processor using *microcode* (i.e., blocks of code). Based on this approach libraries of ACEs may be defined to support the application.

The Teja C [124] is again developed especially for the Intel IXP family processors, yet it also runs on the IBM Power NP family, due to their similar structure. The programming model is defined as an extension of ANSI C, and specifies a set of data types and primitives for the three levels of the Y-chart [74] approach. For instance, at application level, threads, memory spaces and signaling primitives are described. At architecture level, physical components like processor, bus, memory, and their associated properties are specified.

From the enumerated approaches, we consider that conceptually, Teja C is closer to the programming model of TACO, while the MicroACE is similar, to some extent, to the programming model of MICAS.

1.5 Outline

The remainder of this thesis is organized as follows. In Chapter 2 we propose a design methodology for programmable architectures and place it in the context of the model driven paradigm. For this purpose we introduce the main tools of the model driven paradigm and we discuss how they can be employed to support the methodology. The proposed methodology consists of several phases which are detailed in the following chapters of this thesis.

As such, Chapters 3 and 4 propose two alternative solutions for the *application specification* phase. Chapter 3 introduces an approach for the specification of embedded applications intended for programmable architectures. The approach enables the systematic discovery of the functionality of the application, starting from requirements. We use collaboration and activity diagrams as the main tools of the analysis process. Well-defined models are specified for each step, and systematic transformations between steps are proposed to automate the process. Excerpts from an IPv6 routing case study are used to exemplify the approach.

In Chapter 4, we propose to complement UML with data-flow diagrams for the analysis of embedded systems. We adopt existing suggestions of theoretical combinations of data-flow and UML diagrams, and propose a systematic approach in which the transition between the models of the two paradigms may be realized. In addition, we group the identified solutions into an analysis process in which the conceptual model used to represent the system is changed several times. The process is supported by model driven tools and automation is provided using model transformations. The same IPv6 routing application is used as a case study.

Chapters 5 and 6 look at the *architecture specification* phase from a model driven perspective. DSLs for two specific programmable architectures, TACO and MICAS, are defined using the UML extension mechanisms. We show how different abstraction levels of the two architectures are modeled, including the way a programming model is defined and integrated in each DSL. We show how the two DSLs are intergraded in MDA tools, and what tool support could be provided to assist the system-level design process. Furthermore, we discuss how libraries could be defined at different levels of realization, and used to automate the generation of different system-level deliverables. In general terms, the proposed approaches may be seen as a solution to provide design frameworks for other programmable architectures.

In Chapter 5, we also discuss the process of *mapping* the application specification on the architecture specification, in the context of the TACO architecture. We show how the mapping between the application specification and the architecture specification is performed by taking advantage of the TACO programming model, and we define a systematic approach to support the process. Next, a library for encoding mappings (i.e., design decisions) is defined, in order to speed up future mapping processes.

Having the application implemented on a given architecture, Chapter 7 looks at the *simulation* and *design space exploration* of programmable architectures. The chapter is composed of two parts. In the first part, we discuss the system-level simulation of the MICAS programmable architecture from the perspective of different design decisions that allow us to automatically generate the simulation model. In addition, we discuss an approach to combine the MICAS programming model with the simulation framework at simulation time, and how the application "code" and the simulation model of the architecture are co-simulated. In the second part of the chapter, we present a case study in exploring the design space of the TACO programmable architecture. The case study lets us show that by employing system-level design tools for estimating different configurations, we can rapidly and reliably identify suitable configurations with respect to the non-functional requirements of the application.

Conclusions and directions for future work are discussed in Chapter 8.

1.6 List of Publications

The list of publications related to the work discussed in this thesis is given below. Some of their content cannot be found in the current thesis, since it was not the main contribution of the author. In addition, the research presented in this study extends and, hopefully, improves the content of some of the listed publications.

[P.1] J. Lilius and D. Truscan. **UML-driven TTA-based protocol processor design.** In Proceedings of the *Forum on Design and specification Languages (FDL'2002)*, Marseille, France, September 2002.

[P.2] J. Lilius, D. Truscan and S. Virtanen. **Fast evaluation of protocol processor architectures for IPv6 routing.** In Proceedings of the *Design, Automation* *and Test in Europe 2003 (DATE'03)*, volume Designer's Forum. IEEE Computer Society, Munich, Germany, March 2003.

[P.3] D. Truscan, J. M. Fernandes and J. Lilius. **Tool support for DFD-UML model-based transformations.** In Proceedings of *11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS'04)*, pages 388-397, IEEE Computer Society, Brno, Czech Rep, May 24-28, 2004.

[P.4] D. Truscan. **A model driven approach to configuration of TTA-based protocol processing platforms.** In Proceedings of the *Forum on Design and specification Languages (FDL'2004)*, Lille, France, September 13-17, 2004.

[P.5] D. Truscan. **A UML profile for the TACO protocol processing platform.** In Proceedings of the *22nd Norchip Conference*, pages 225-228. IEEE Computer Society, Region 8, Oslo, Norway, November 8-9, 2004.

[P.6] M. Alanen, J. Lilius, I. Porres and D. Truscan. **On Modeling Techniques for Supporting Model Driven Development of Protocol Processing Applications.** In Eds: S. Beydeda and V. Gruhn, *Model Driven Software Development - Volume II of Research and Practice in Software Engineering*, Pages 305-329, Springer-Verlag, 2005.

[P.7] J. M. Fernandes, J. Lilius and D. Truscan. **Integration of DFDs into a UML-based Model-driven Engineering Approach.** In *Software and Systems Modeling (SOSYM)*, 5(4):403–428, Springer-Verlag, December 2006 DOI: 10.1007/s10270-006-0013-0.

[P.8] M. Alanen, J. Lilius, I. Porres, D. Truscan, I. Oliver and K. Sandströom. **Design Method Support for Domain Specific SoC Design.** In Proceedings of the *Joint MOMPES/MBD'06 workshop* within the 13th IEEE Int. Conf. on Engineering of Computer Based Systems (ECBS 2006), Potsdam, Germany, March 27-30, 2006.

Chapter 2

Model Driven Development of Programmable Architectures

In this chapter[1], we propose a customized version of the Y-chart approach for the development of the programmable architectures discussed in this thesis. In addition, we present the principles of the model driven paradigm and we propose their adoption in support of the suggested methodology.

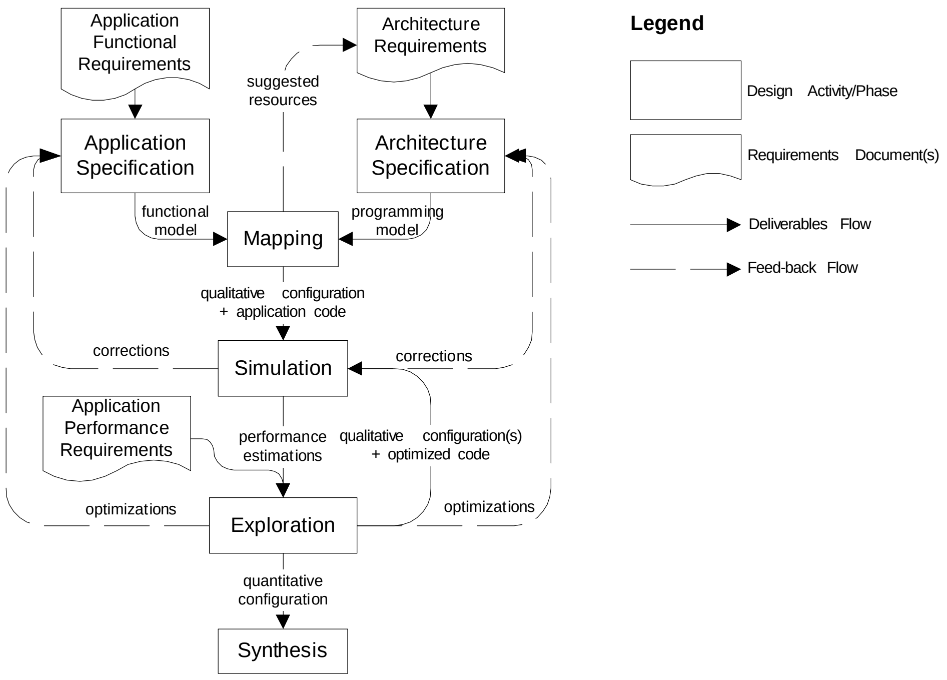
2.1 A Development Methodology for Programmable Architectures

Using the basic principles behind the Y-chart methodology [74], we propose an approach that addresses the characteristics of both the application domains that we look at, protocol and multimedia processing, and also of the programmable architectures, TACO and MICAS, that we use here.

Figure 2.1 presents the process of the generic methodology that we propose. The process differs from the one of Y-chart in the fact that only the *Application Functional Requirements* are taken into consideration during the application specification process. The impact of such an approach is that the mapping process is performed only with respect to the functionality of the application. Consequently, a *qualitative configuration* of the architecture is obtained. The *Non-functional Requirements* (e.g., performance and physical characteristics) of the application are taken into account only in the architecture exploration phase of the process, where a *quantitative configuration* of the architecture is obtained. We briefly describe the main phases of the process here, to provide a general overview of the approach, deferring the details of each phase to the following chapters.

In the *Application Specification* phase, the application is analyzed with respect to the functional requirements in a top-down manner. Several subphases are used,

[1]Ideas discussed in this chapter have been published in [P.6]



Application Functional Requirements
Architecture Requirements
Legend
suggested resources
Application Specification
Architecture Specification
Design Activity/Phase
Requirements Document(s)
functional model
Mapping
programming model
Deliverables Flow
Feed-back Flow
qualitative configuration + application code
corrections
Simulation
corrections
Application Performance Requirements
performance estimations
qualitative configuration(s) + optimized code
optimizations
Exploration
optimizations
quantitative configuration
Synthesis

Figure 2.1: Development process for programmable architectures

for instance requirements analysis, application analysis, etc. At each subphase, more details are gathered into the specification. The main goal of this phase is to identify the pieces of functionality of the application, which have to be supported by the architecture.

The *Architecture Specification* phase starts from the *Architecture Requirements*, which capture the functionality that the architecture has to provide. This phase may be seen as a combination of a top-down and a bottom-up approach, respectively. In the former, hardware resources are identified from requirements of the architecture; in the latter, the functionality supported by the architecture is extracted from its hardware resources. Several subphases may be defined aiming at specifying the system at several levels of abstraction.

The models resulting from the application and architecture specification processes provide a functional view of the application and of the architecture, respectively, which in turn form the input to the *Mapping* process. During this phase, the functionality required by the application is mapped to the functionality provided by the architecture. Two artifacts are obtained: a *qualitative configuration* of the architecture to support the functional requirements of the application, and the *application code* to run on this configuration. If some functionality of the application is not supported by the architecture, new resources of the architecture may be suggested.

In the *Simulation* phase, the functionality of the resulting system is validated, with respect to the requirements of the application. The validation is performed based on the input/output behavior of the system. In the situation that errors in the specification (of both the application and architecture) occur, corrections are suggested, and once they are attended, the mapping process is performed again.

The *Exploration* phase deals with tailoring the architecture towards an optimal implementation of the application, in terms of performance requirements and physical constraints. This phase implies performing estimations of the architecture from several perspectives. For instance, the system may be simulated again, and as a result, the designer collects performance estimates, with respect to the throughput of the application. Additionally, an estimation of the physical characteristics (e.g., occupied area, power use) of the configuration is performed. Based on this data, mainly optimizations of the architecture, but also of the application specification, are suggested. The exploration process is performed iteratively until a satisfactory configuration (i.e., the *quantitative configuration*) is obtained. For each quantitative configuration, the application code is optimized such that it takes into account the parallelism of the configuration.

Out of the exploration process, the designer selects one or many configurations suited to implement the application and targets them to be synthesized in hardware. However, we will not address the synthesis phase of the process in this thesis.

2.2 The Basic Principles and Tools of the Model Driven Paradigm

As mentioned in the introduction, software systems can be more easily and efficiently developed, if some abstraction principles are followed. Based on this paradigm, the *visual models* of the system have replaced the source code that was traditionally used for specifying software systems. Moreover, according to this paradigm, these models should be specified independently of the concepts of any implementation platform, thus moving the focus from implementation issues to problem-domain ones.

To support such a viewpoint, several modeling approaches have been developed in recent years. The *Model Driven Architecture* (MDA) [93], as promoted by the Object Management Group (OMG), has been, from the very beginning, the driving force behind the new paradigm. MDA proposes the use of *models* to specify the system, starting from requirements and advancing towards specific implementations. Several models are proposed within this scope. A *Computational Independent Model* (CIM) abstracts the concepts (and their relationships) of a given application domain; a *platform independent model* (PIM) is used to specify the software system independently of the characteristics of the implementation architecture, and last but not least, the implementation architecture is specified by a *platform model* (PM). By implementing the PIM on a given PM, a *platform specific* *model* (PSM) is obtained. One of the key ideas behind MDA is the use of *model transformations* as the main mechanism to move the system specification from one abstraction level to another. Four types of possible model transformations are proposed:

- the *PIM→PIM transformation* promotes refinements (e.g., adding more details) of a specification, in a platform independent manner;
- the *PIM→PSM transformation* projects a platform independent specification on a given "implementation" infrastructure;
- the *PSM→PSM transformation* covers refinements of the specification, in a platform-dependent manner;
- the *PSM→PIM transformation* changes existing specification models into more abstract ones by removing platform dependent details.

Beside being a promoter of the model driven paradigm, OMG [90] also proposes a number of technology standards to sustain the MDA approach:

- Object Constraint Model (OCL) [92] to enforce the consistency of models;
- the Meta-Object Facility (MOF) [95] to define new modeling languages;
- XML Metadata Interchange (XMI) [91] to support model interchange, etc.

Among the proposed standards, the Unified Modeling Language (UML) is suggested as the *de facto* language for "visualizing, specifying, and documenting software systems" [96].

Although MDA suggests the usage of models and model transformations to evolve the specification, there is no indication on how, when and under what conditions models are created and transformed during the entire development process. Initially proposed by Kent [70], the *Model Driven Engineering* (MDE) extends MDA by combining "process and analysis with architecture".

Other researchers realized that the MDA paradigm can be applied not only to software, but to other domains, too. The *Model Based Development* (MBD) [110, 111] is similar in spirit to MDE, with the difference that, now, the *system* seen as a combination of software and hardware is to be modeled. Beside pointing out that a number of activities need to be defined and performed at each step of the development cycle, MBD argues that the designer also needs a collection of domain specific languages to model the system at different levels of abstraction. The approach targets, in particular, embedded software, but also the embedded system as a whole. In addition, applying the same idea of abstraction to handle complexity, MBD suggests that the hardware platform should also be abstracted at several levels, following the principles of the platform-based design [131].

In our opinion, all these paradigms have the following aspect in common: they define an infrastructure and a set of concepts, which may be used for defining domain-specific methodologies (i.e., processes, methods and corresponding tool support). We see the above mentioned modeling paradigms as complementary, each of them adding new principles and ideas on top of the existing ones. Therefore, throughout this thesis, we try to adopt the best of each. In the rest of this work, we use the term *model driven paradigm* to cover the principles of all these extensions in a generic manner.

In the following, we briefly discuss the tools of the model driven paradigm that one may use to define and support a custom methodology for domain specific applications.

2.2.1 The Unified Modeling Language

The use of object-orientation provides an important abstraction mechanism if compared to traditional methods. Features like data encapsulation, inheritance and polymorphism greatly improved the efficiency of programming and gradually raised the popularity of the object-oriented programming languages.

Along with the large scale adoption of object-orientation, different methods for the analysis and design of software systems have been conceived. Due to the lack of consensus among these methodologies and as well as to their deficiencies, very few were actually adopted, whereas others were either combined or simply filtered out. Those few ones that survived started to present similarities and to slowly evolve towards common grounds. Following this trend and, at the same time, due to the need for a commonly used modeling language, the promoters of various methodologies decided to join forces in creating a unified notation. Consequently, the notations of the three of the major methods of that time, Booch methodology [22], the Object Modeling Technique (OMT) [108] and Object-Oriented Software Engineering [65], were conjoined into the *Unified Modeling Language* (UML).

UML was adopted by the Object Management Group (OMG) [90] in 1997 (three years before MDA). Several versions of UML have been issued, each one with additions and improvements as compared to the previous ones. Here we use UML version 1.4 [94], since it was the most recent version at the starting time of this research. Nevertheless, the approaches discussed in this thesis may also be applied to the latest UML version, namely version 2.0, possibly with some changes. Henceforward, the *UML* term is equated to UML version 1.4, unless otherwise specified.

UML provides nine diagrams to be used during the entire development process of software systems. The diagrams cover the four main perspectives of software modeling. The *functional perspective* is modeled using *Use Case diagrams*, which enable one to capture the requirements of a system by modeling the relationships among *actors* (external users of the system) and *use cases* (functionality of the system). The *Class diagram* provides a *static perspective* by modeling the structure of the system in terms of data types (i.e., *classes*), their properties (i.e., *attributes* and *operations*) and relationships (e.g., *containment*, *inheritance*, *associations*, etc.). In addition, *Object diagrams* may be used to show the structural relationships between class instances (i.e., *objects*). From a *dynamic perspective*, four diagram types are provided, and they are divided into two categories: *interaction diagrams* and *state diagrams*. Interaction diagrams are used to show the interaction between *objects*, either as a sequence in time (i.e., *Sequence diagrams*) or as communication (i.e., *Collaboration diagrams*). In the second category, the *State diagrams* provide a view over the behavior of objects, by modeling their *states* and the *transitions* between these states. The *Activity diagram* models the order in which the activities of a system are followed. Two diagrams are used to model the *physical perspective* of a system. The *Component diagram* provides a high-level view of the structure of the application code, like source code, binary, or executable components. A *Deployment diagram* is used to show a run-time perspective of the physical processing elements and of the software components using them.

UML has been proposed as a general-purpose language for software specification and, implicitly, as the default modeling language of MDA. Its nine diagrams are not intended to be used all at once, but the designer has to select the ones needed for a given problem. In case the basic diagrams are not sufficient, the UML language provides extensibility mechanisms which allow the customization of its elements by defining *profiles*.

There are many critiques regarding the use of UML (e.g., [8, 44]), nevertheless it seems that UML is being adopted by the industry at a steady pace. Statistics from 2002 show that the UML market has been growing 20 percent per year [121].

2.2.2 Models, Metamodels, and Profiles

The *model* is the main tool of the model driven paradigm that specifies the system at different levels of realization. A model is composed of a set of language elements that conform to a language definition that specifies the elements and their relationships.

A four-layered architecture (Table 2.1) is used to define new languages. At the bottom level (i.e., *level 0*), concrete *instances* of the domain concepts are present. *Level 1* defines the *language* or the *model* used to represent the concepts of the domain. *Level 2* provides a definition of the language used on level 1, or in a model driven terminology, a *metamodel*. Finally, on *level 3*, a *language for defining languages* is used to create new language definitions. Such a language is referred to as a *meta-metamodel*.

To define new modeling languages (i.e., metamodels), a new standard, the Meta Object Facility (MOF) [95], has been proposed. MOF may be seen as a language for language definitions, or in other words as a meta-metamodel. MOF is an object-oriented framework that uses only four elements to define new languages: classes, associations, data types and packages. *Classes* are used to model the meta-objects in the new language, while *Associations* model binary relationships between meta-objects. *Datatypes* define the data used in the new language, while *packages* provide model management facilities, enabling the split of the definition of the new language into several parts, in order to cut down on complexity. Basically, a metamodel is defined using the UML class diagram notation. As such, associated object-oriented features are supported: classes may have attributes, they

Level	**Description**	**MDA term**	**Example**
level 3	language for defining languages	meta-metamodel	a MOF Class (MetaClass)
level 2	language definition	metamodel	a UML Class
level 1	domain concepts/ language elements	model	class "Car"
level 0	domain/language instances	instance	object car = "BMW"

Table 2.1: The four-layer metamodeling architecture

may be associated using inheritance, aggregation, composition, etc. In addition to the abstract syntax provided by the metamodel definition, other domain specific constraints can be enforced using well-formedness rules expressed in the Object Constraint Language (OCL) [92].

A light-weight approach in defining new modeling languages is the *profile* mechanism provided by UML [94, pp. 2-191]. Using this mechanism the UML language elements can be adapted to represent concepts of specific application domains. There are three elements provided for building profiles:

- *Stereotypes* provide "a way of branding" a UML model element, such that the latter represents a given concept of the application domain;
- *Tagged definitions* are used to define additional properties for the stereotyped elements;
- *Constraints* enable for the specification of additional restrictions, beside the ones provided by the UML metamodel.

A coherent collection of stereotypes, tagged values and constraints forms a *UML profile*. In fact, since profiles customize the UML metamodel, they are metamodels themselves.

Using these extension mechanisms, any new or existing modeling language may be expressed in a model driven context. The UML metamodel itself has been developed following the four-layer architecture.

2.2.3 Model Transformations

The *model transformation* is the main tool of the model driven paradigm that evolves abstract specifications into more concrete ones. A model transformation translates a collection of source models that *conform to* a given set of language (i.e., metamodels) into a collection of target models that *conform to* the same or to a different set of language (i.e., metamodel).

One of the strong points of the model driven approaches, in general, is the ability to refine a platform-independent specification of the system into a platform-dependent specification in a continuous manner. This means that the same model that has been the target of a PIM→PSM transformation may become, in turn, the source model for a different PIM→PSM transformation (see Figure 2.2). In MDA terminology this means: any PSM may be considered as a PIM in the next level

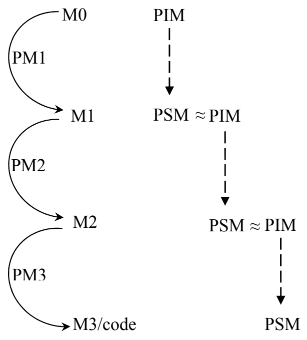


Figure 2.2: Applying model transformations. The same model can be platform-independent from one perspective and platform-specific from another perspective

of realization. This lets us regard as PM any new abstraction level used to project the specification during the development process. A special kind of model transformation is the *code generation*, which takes a source model (that conforms to a metamodel) into the code of a programming language.

Ideally, model transformations should encapsulate activities that are commonly repeated. In industrial settings, automating the model transformations is the primary focus, since it leads to a substantial increase in productivity and thus, in the company profits [125].

2.2.4 Processes

The available collection of artifacts (i.e., models and transformations) has to be organized in a consistent manner in order to obtain the expected deliverables from the process. MDA does not specify the way in which different artifacts of the system are obtained and transformed, neither does it indicate what the intermediate deliverables are and how they can be achieved [70]. Therefore, we adopt the concept of *process* [70] to depict the evolution of a system at different levels of abstraction.

A process may be hierarchically decomposed into (smaller) sub-processes (also called *phases* or *steps*). Additionally, the process must suggest the order in which the sub-processes are supposed to be performed. During each phase, a specific set of concepts, models, and metamodels are used to support the specification. The steps are mainly intended to provide a systematic approach on how a given phase is accomplished. In addition, tool support is assumed at each phase, tailored to the specific needs of that particular phase. This implies that a tool chain could be used to support the entire design process.

2.2.5 Tool Support

Tool support is one of the prerequisites of the model driven paradigm, although the MDA specification does not describe how this is to be achieved in practice. The importance of tools stands in the fact that the successful realization of model driven methodologies is given by the quality of the tools supporting them. In fact, the CASE (*Computer Added Software Engineering*) tool support has proved [26] to significantly increase the productivity of software projects. Tool support has to be provided in terms of editors for the modeling languages, transformation engines, constraint checking engines, and guidelines to assist the development process.

In recent years, a multitude of academic and industrial tools have been developed to support the model driven paradigm. Comprehensive lists of such tools and an overview of their capabilities may be found on several web sites like: `http://www.modelbased.net/mda_tools.html`, `http://www.omg.org/mda/`, or `http://planetmde.org/`, whereas a comparison of several MDA tools in terms of the features that they provide may be found in [123].

Benefiting from the concepts and tools provided by the model driven paradigm, we propose to use them as a framework for supporting the development of programmable architectures. More specifically, we regard the process proposed in Figure 2.1 as being supported by several solutions. A concrete solution in modeling the specification of the system is proposed in each phase, as it will be discussed in the next chapters. Moreover, several solutions for each phase may be available, and in this case, we leave to the designer the choice of selecting the ones that suit better a particular problem domain.

During each phase of the process, models (based on corresponding metamodels) are used to describe the specification of the system, at different levels of abstraction. In order to relate one model to another, transformations are defined; they provide not only a systematic approach in refining the specification, but also tool support and automation. Figure 2.3 presents the previously proposed process for programmable architectures, in terms of models and model transformations. As one may notice, the entire process can be seen as a sequence of PIM→PSM transformations performed at different levels of abstraction.

The input of this process is represented by the *Functional Requirements* and *Non-functional Requirements* of the application, and by the *Architecture Requirements* of the target programmable architecture. During the *Application Specification* phase (Figure 2.1) the *Functional Application Model* is obtained. Similarly, in the *Architecture Specification* phase the *Architecture Programming Model* is constructed. Using these two models, the application specification is "implemented" on the architectural specification during the *Mapping* phase. The preliminary result of the *Mapping* process is the *Platform-Specific Application Model*, representing the application specification expressed in terms of the programming model of the architecture. In a subsequent transformation, the *Platform-Specific Application*

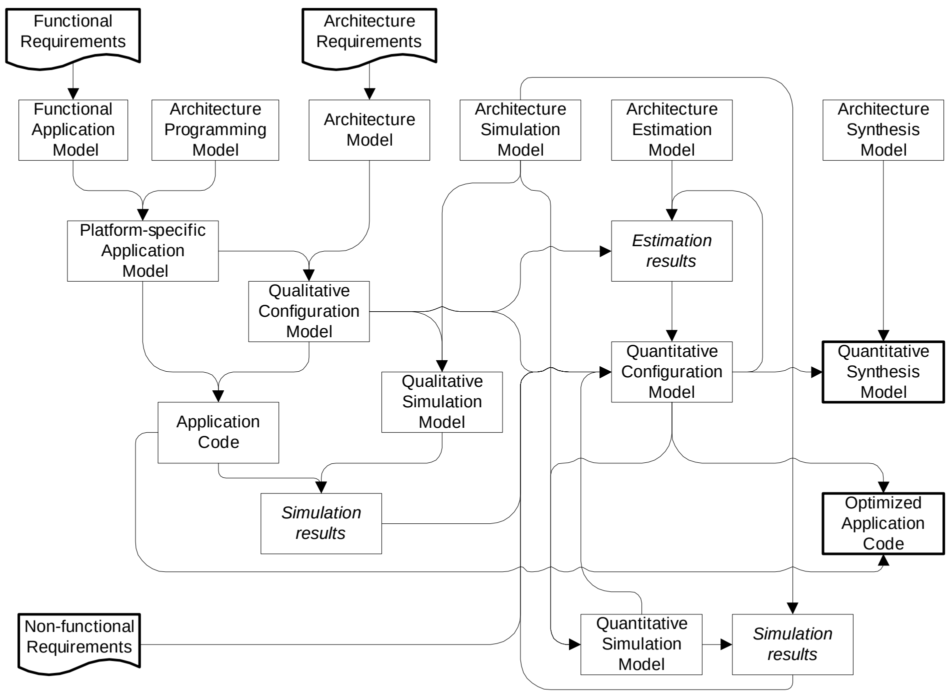

Functional Requirements
Architecture Requirements
Functional Application Model
Architecture Programming Model
Architecture Model
Architecture Simulation Model
Architecture Estimation Model
Architecture Synthesis Model
Platform-specific Application Model
Estimation results
Qualitative Configuration Model
Quantitative Configuration Model
Quantitative Synthesis Model
Qualitative Simulation Model
Application Code
Simulation results
Optimized Application Code
Non-functional Requirements
Quantitative Simulation Model
Simulation results

Figure 2.3: Artifact-based view of the development process for programmable architectures

Model is mapped onto the *Architecture Model* in order to identify the hardware components of the architecture (i.e., *Qualitative Configuration Model*). Based on the selected hardware components and on their capabilities, the *Platform-Specific Application Model* is refined (transformed) into the *Application Code* that will be used to drive the qualitative configuration of the architecture.

Once the *Mapping* phase is completed the *Simulation* of the identified system is performed with respect to its functionality. The *Qualitative Simulation Model* is obtained via a PIM→PSM transformation from the *Qualitative Configuration Model* and from the *Architecture Simulation Model*. Preliminary performance data (*Simulation results*) are gathered from the simulation to serve in the *Exploration* phase, along with *Estimation results* and the *Non-functional Requirements*, for building the *Quantitative Configuration Model*. Based on the latter model, the *Application Code* is optimized to provide the required application performance.

Two main deliverables are produced as output of this process: the *Quantitative Synthesis Model* to implement the architectural configuration in hardware and the *Optimized Application Code* to run on the resulting quantitative configuration.

As mentioned in the introduction, we adopt the use of UML as the main modeling language throughout the entire process. The process of our methodology stretches across several platforms, both in the hardware and in the software domains. When the set of diagrams provided by UML is not sufficient or not expressive enough, we use its extension mechanisms to create domain specific languages.

2.3 Summary

In this chapter, we have introduced a methodology for programmable processors. The methodology can be seen as a derivation of the Y-chart approach. We have placed this methodology in the context of the model driven paradigm to show that it may benefit from the model driven principles and tools.

In the following, we will discuss in more detail the solutions that we propose or use for each phase of the methodology. As a precursory remark, we consider that several solutions could be suggested for each phase, and that they should be combined by the designer in a consistent process, based on the particularities of the problem to be addressed.

Thus, our methodology is intended to be a general framework (i.e., collection of concepts, methods and tools) that the designer can customize for a specific application domain, based on his/her needs. We are not advocating a “one-size-fits-all” solution, instead we consider that each application domain should use a customized solution that optimally exploits the particularities of its concepts and features.

Chapter 3

UML for Application Specification

In this chapter[1] we discuss a UML-based methodology for specifying embedded applications targeting programmable architectures. The approach discussed in this chapter corresponds to the *Application Specification* phase in Figure 2.1. In contrast to most of the current methodologies for programmable architectures, in which ad hoc approaches are used for creating the application specification, we propose a solution in which the specification is systematically constructed from the initial application requirements, until it becomes "mappable" onto the implementation architecture. UML plays a central-role in this methodology, which is based on the observation that UML provides a set of diagrams for modeling various aspects of a system, without explicitly defining a process for combining these diagrams consistently. Therefore, we propose a methodology in which we select several UML diagrams and combine them for specifying applications in the application domains that we address, without aiming at providing a general approach valid for all domains. Furthermore, we do not aim for a completely new methodology, but rather to combine existing methods and tools that help us achieve our goals. Simplified examples of an *IPv6 router* are used to document the approach.

Figure 3.1 presents the main steps of the methodology. We start by capturing the application requirements using the use cases technique, then we identify objects in the system following a functional decomposition. Afterwards, we perform behavioral analysis to identify relationships among objects and the functionality that each object provides. Next, we suggest the use of collaboration diagrams in combination with activity diagrams as the main tool of the behavioral analysis process. Finally, the class diagram of the system is obtained from the identified objects.

[1]This chapter is based on and extends the [P.1] publication.

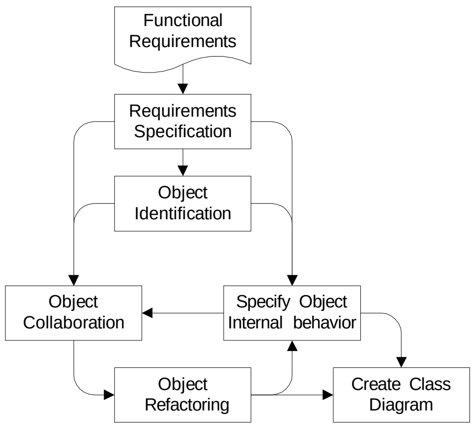


Figure 3.1: Main steps of the analysis process

3.1 Requirements Specification

The application requirements are usually provided to the system designer, during the negotiation with the customer, in a written form and using natural language. It is the designer's job to process these requirements and extract those aspects relevant to the specification of the system. For the IPv6 router case study that we use, the main sources of information are based on the *Internet Protocol version 6* (IPv6) [34] and the *Routing Information Protocol next generation* (RIPng) [82] specifications, as well as on related standards and books. A more detailed description of the two protocols and the motivation behind choosing RIPng as the routing protocol are given in [134]. In the following, we discuss only those aspects relevant to the current thesis.

Briefly, the goal of our case study is the design of a 10 Gbps IPv6 router that routes datagrams over Ethernet networks, using the RIPng protocol. A router is a network device that deals with switching data from one network to another in order to reach its final destination. Two main functionalities have to be supported by the device: forwarding and routing. *Forwarding* is the process of determining on what interface of the router a datagram has to be send on its way to destination. *Routing* is the process of building and maintaining a table (i.e., the routing table) that contains information about the topology of the network. The router builds up the routing table by listening to specific datagrams broadcasted by the adjacent routers, in order to find out information regarding its networking environment. At regular intervals, the routing table information is broadcasted to the adjacent routers to inform about changes in the "known" topology. In brief, an IPv6 router should be able to receive IPv6 datagrams from the connected networks, to check their validity for the right addressing and fields, to interrogate the routing table for the interface(s) that they should be forwarded on, and to send the datagrams to the appropriate interface(s).

Capturing the Requirements into Use Cases. The Use Case diagram is among the main artefacts used for capturing requirements in UML. In many research communities it is seen as an important tool for initiating object-oriented projects [23]. Even if our target application is not necessarily object-oriented in its nature, we consider use cases as a useful technique for requirements elicitation between the analyst and the domain expert. In fact, use cases are quite independent of object-oriented modeling and they may be applied to any system specification [64].

There are basically three types of elements in a use case diagram. An *actor* is an external user of the system, while an *Use Case* is a piece of functionality (e.g., a service) provided by the system. *Relationships* indicate not only which actor and use case instances communicate (i.e., *associations*), but also how use cases relate to each other (i.e., *include*, *extend* and *generalization*).

In our case study, two external actors that interact with the router have been identified from the IPv6 router requirements. The *Node* is a common network node that requests the router to forward datagrams and eventually receives an ICMPv6 error message in case of failure. The *Router* depicts neighboring routers that exchange topological information with our router. In the current example, we assume that the datagrams exchanged between the router and the environment are of three types, namely *Forwarding*, *Routing* and *Error* type, respectively and that their classification is done outside the system (i.e., the datagram is already correctly classified when entering the IPv6 router). Next, we have extracted six use cases, as shown in Figure 3.2, from the written specification of the IPv6 router. Several possibilities of describing use cases are proposed by the UML standard itself: textual, state machines and even methods and operations associated to use cases. There seems to be no definite guidelines for identifying and specifying use cases, different researchers proposing different solutions [14, 24]. For the moment, we follow a textual-based description, yet a more elaborated technique, like a template-based one [14], may be applied. In the following, we briefly describe the identified use cases:

1. *Forward Datagram* - The router receives a datagram from a node on one of its interfaces and decides on what interface(s) the datagram should be sent, in order to reach its final destination. Upon receiving, the datagram is checked for validity and correct addressing, and if not correct, the ICMPv6 sub-protocol is invoked to send an Error Message back to the originator of the datagram. Then, the Routing Table is interrogated, to decide the next interface. If no route is found, the ICMPv6 sub-protocol is invoked to send an Error Message back to the originator of the datagram.
2. *Send Error* - If, during the processing of a datagram, an error is encountered in its fields, or a forwarding route for the datagram is not found, an ICMPv6 Error Message is sent back to the originator.
3. *Treat Request* - The router receives Request datagrams (from the adjacent routers on the network) that ask for topological information from the Routing Table.

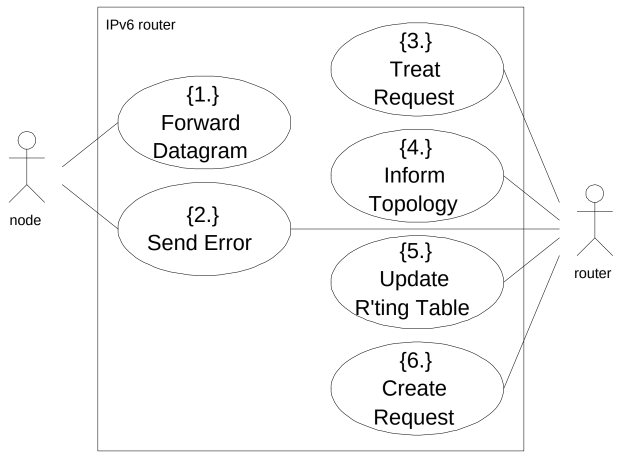


Figure 3.2: Use Case diagram for the IPv6 router

Upon receiving, the datagrams are checked for validity and correctness of the fields. If a datagram is not correct, the ICMPv6 sub-protocol is required to send an Error message back to the originator.

4. *Inform Topology* - The information stored in the Routing Table is sent to adjacent routers to inform about changes in the topological information. This information is sent either periodically, as multicast datagrams to all adjacent routers, or as a result of a special request from a single router.
5. *Update Routing Table* - The router receives Response datagrams from adjacent routers in order to update its Routing Table information. Upon receiving, the datagrams are checked for validity and correctness of the fields. If a datagram is not correct, the ICMPv6 sub-protocol is invoked to send an Error message back to the originator.
6. *Create Request* - Whenever the Routing Table of the router is empty, a Request message is sent (as a multicast datagram) on each interface, to all adjacent routers, to request topological information.

3.2 Object Analysis

The object analysis process is performed in order to get a better understanding of the application functional requirements. During this step, the emphasis is put on identifying the objects in the system and how they collaborate to achieve the overall functionality of the application.

The traditional UML approach, based on its object-oriented fundament, suggests that first classes of a system are identified, and then instances (i.e., objects) are created to provide a run-time view of the system. Here, the problem that we face is similar to the question *"Who was the first on Earth, the egg or the hen?"*.

Some researchers consider that the classes in the system should be discovered first and that, later on, objects are obtained as instances of these classes [14, 39]. This is also in agreement with the idea of object-orientation. In contrast, other authors [48, 107] consider that, for certain application domains, the objects in the system should be firstly identified and then classified.

At the time we have started this work, with the exception of the methods based on the *noun identification technique* [4], most of the existing methods for object identification seemed to be applied in a quite ad-hoc manner. That is, the process is very much based on the experience, intuition and inspiration of the analyst. Moreover, applying different methods to the same specification often results in a different set of objects. Many researchers admit the existence of a conceptual gap between use cases and class/object identification [59, 101]. A couple of techniques meant to alleviate this problem have been proposed in literature [15, 39, 63, 107, 138]. Other approaches consider that the objects should be extracted from use case diagrams [48, 107], or even following an automated object discovery process [18].

It is also our opinion that, in particular for the embedded system domain, identifying the objects first and then classifying them is preferable. It is the objects in their whole that constitute a system, and not the classes that only categorize them [47]. Furthermore, we need to analyze what are and how different instances of the application domain interact together [77], rather than to focus on a structural perspective of the system. In fact, the object diagram seems to be considered a second-class citizen, many object-oriented design methodologies focusing on the class diagram and very few on the object one [120].

The 4-Step Rule Set Method. After having surveyed several methods, we decide to follow the *4-Step Rule Set* (4SRS) method [47, 48, 81], because of its systematic approach in identifying objects. The 4SRS method has been intentionally developed to provide a systematic approach for bridging the gap between the requirements analysis and object identification phases. 4SRS is an extension of the Jacobson's approach [63], and proposes to split use cases into three objects: interface, data, and control, respectively. *Interface objects* are used at the border of a system in order to intermediate the communication with the external environment (i.e., actors), *control objects* implement the algorithmic and the control behavior of the system, while *data objects* take care of storing data and providing access to it.

The method is composed of four steps. In *step 1*, each use case is transformed into the three types of objects mentioned above. *Step 2* decides which of the three objects are necessary to implement the functionality of the use case. For instance, an *interface* or a *data* object obtained from a use case describing primarily an algorithmic computation might not be providing any information, so it may be removed. In *step 3*, objects are aggregated and combined into packages based on their common characteristics. Finally, *step 4* is used to identify the associations

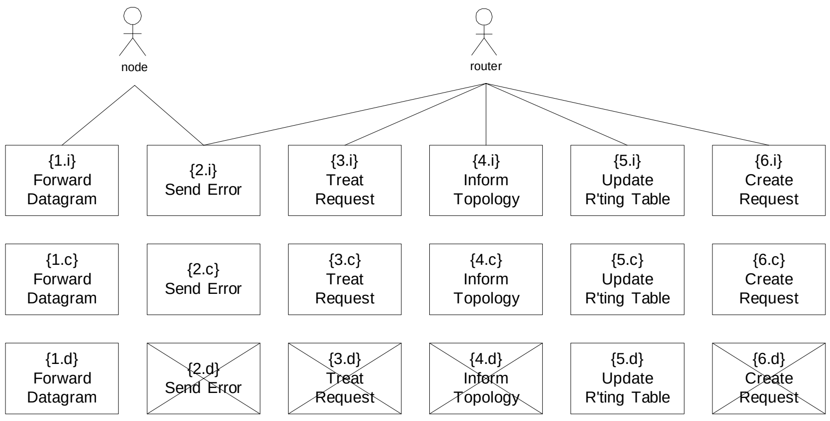


Figure 3.3: Initial Object diagram for the IPv6 router

between objects. At each step, numeric tags are used to trace the objects belonging to the original use cases.

As a result of applying step 1 to the use case diagram in Figure 3.2, we obtain three new objects for each use case. The objects have the same tag number as the original use case, and an additional tag name is used to indicate the object type. For instance, the {*1.i*}, {*1.c*} and {*1.d*} objects (see Figure 3.3) represent respectively the interface, control and data-objects obtained from splitting the {*1.*} *Forward Datagram* use case. In addition, an actor is created in the object diagram for each actor in the use case diagram.

By applying step 2, we take the decision upon which objects are kept or disposed, based on the original specification of the use case. Just for convenience, some of the objects may also be renamed. Following this approach, only the following data-objects have survived: {1.d} (i.e., *forward datagram*) and {5.d} (i.e., *routing table*). Although we could also have data-objects for use cases {2}, {3}, {4}, {6}, we have decided that only the major functionalities of the system (i.e., {1.} forwarding and {5.} routing) require a data object. This does not mean that information is not produced in objects {2.c}, {3.c}, {4.c} and {6.c}, but that the major role of those objects is to perform behavior.

In step 3, the remaining objects should be organized into packages or aggregations, based on their common characteristics, while in step 4 the associations between the identified packages/objects should be identified. These two steps are based on a rather high-level view of the system and on the intuition of the designer. We consider that in order to be able to identify "related" objects, as well as the associations between them, a deeper look into their functionality and into their collaborations is required. Keeping this goal in mind, we propose an approach that allows us to perform a more thorough analysis of object interactions and of their internal behavior.

3.3 Analyzing Object Behavior

Behavioral analysis is performed by applying use case scenarios on the set of objects identified in the previous steps. The process is focused not only on the identification of how objects interact at run-time, but also on how their internal behavior supports this interaction. Two UML diagrams are used to support the approach: *collaboration diagrams* for representing object interaction, and *activity diagrams* for specifying the internal behavior of the objects. At high-level, the approach may be seen as composed of three basic steps: a) the interaction between objects is depicted in terms of collaboration diagrams; b) the internal behavior of each object is specified; c) the resulting set of objects is refactored by grouping/splitting objects and their behavior. These steps are applied iteratively until no more refactorings are necessary.

a. Identifying Object Collaborations

In order to identify the interaction between the initial set of objects, use case scenarios are applied. We model this interaction by collaboration diagrams. A *collaboration diagram* specifies the object communication by showing the information passed among objects. This helps in determining what functionality should be provided by each object to its neighbors. UML allows the description of the messages sent among objects and their ordering in time. Messages have a name and may carry a list of parameters.

The identification of object collaborations is composed of three substeps:

a.1. apply scenarios on the initial set of objects;
a.2. create a collaboration diagram for each scenario;
a.3. identify services (i.e., operations) provided by each object;

After the first two substeps have been executed, a collaboration diagram is obtained for each scenario of the system. It is important to note that a scenario may involve more than one use case and therefore, a use case may take part in several scenarios. At object level, this translates into the fact that an object may also take part in several scenarios. Figure 3.4 shows an example of applying the "$\{$*1*$\}$. *Forward destination unreachable*" scenario on the previously identified set of objects. The interaction between the objects participating in the scenario is expressed in terms of service requests, which are tagged based on the tag of the scenario (i.e., *1.*). Six objects collaborate to achieve the scenario. Object $\{$*1.c*$\}$ is in charge of interrogating (*1.1 getNext()*) the input interfaces of the router (object $\{$*1.i*$\}$) for incoming datagrams, and of storing (*1.2 store()*) them into the $\{$*1.d*$\}$ object. Based on the information extracted from the datagram, the routing table is interrogated (*1.3 RTLookUp()*) in order to determine the next interface of the router on which the datagram has to be forwarded. In case such an interface is not found, an ICMPv6 datagram is created (*1.4 error()*) from the original datagram (*1.5 retr()*) and sent

(*1.5 send()*) to the interface from which it originated. In UML terminology, services provided by classes (and therefore by objects) are referred to as *operations*. Based on the service requests between objects, the operations of each object are identified in substep a.3. Applying all the scenarios of the specification, the operations provided by objects are gradually identified.

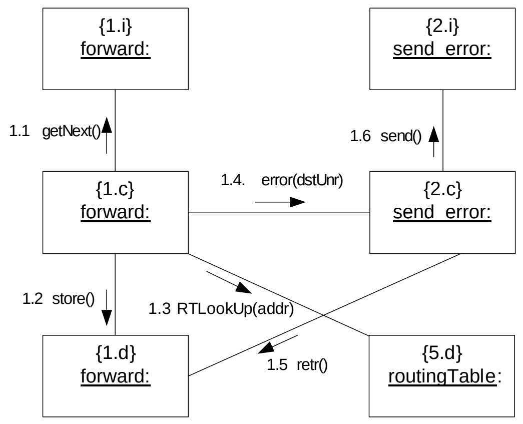


Figure 3.4: Collaboration diagram for the $\{$*1.*$\}$ *Forward destination unreachable* scenario

b. Specifying Object Behavior by Activity Diagrams

A UML *Activity Graph* [25, 39, 94] is a kind of a state machine that focuses on the sequence and the conditions under which certain actions are executed. An activity graph is composed of a set of nodes, called *states*, which represent atomic computations and directed vertices depicting *transitions* from one state to another. Each state is triggered by a transition and executes until completion. A state may be either a *SubActivityState* (a composite state that contains another activity graph) or an *ActionState* (an atomic state that cannot be further decomposed). *Pseudostates* (like *initial*, *join*, *fork* or *junction*) are also defined to model complex transitions. In addition, a *FinalState* is used to model states that cannot have any outgoing transitions, while *Decision* states are provided to specify branching in the execution flow. Conditional execution may be obtained through *guarded transitions* (transitions that are executed when an associated guard condition is met). In addition, transitions may trigger events. Two graphical notations are used to explicitly represent the *send signal* and *receive signal* events, which model the communication with the external environment (an example will be shown in Figure 3.7).

Although the UML standard suggests the use of activity graphs mainly for workflow specification, our motivation for choosing them to model the internal behavior of the objects is three-fold:

1. *subActivity states* may be refined (decomposed) into other activity graphs, enabling the designer to work on several levels of abstraction and also to refine the initial activity graphs into more detailed specification;
2. activity graphs enable the natural modeling of the sequencing of processing steps that are applied on data, in data-driven systems (such as protocol processing applications);
3. UML does not prescribe a specific language for specifying the activities, allowing by this flexibility in describing the states, at the expense of a loose specification. Thus, one may use natural language or programming languages constructs to describe the action states at different levels of abstraction.

Examples of using activity diagrams for behavioral specification are limited. In [11], the application analysis is performed and the design constraints captured in a use case representation. Then, by using activity diagrams and OCL, the design is transformed into a formal model that can be partitioned into hardware and software, according to a fixed set of hardware resources and following object-oriented techniques, respectively. Björklund [20] proposes the translation of UML statecharts and activity graphs into Rialto, a language for representing multiple models of computation. Ali and Tanaka [7] propose the use of activity diagrams to model control objects; Java code is generated from the specification. The use of activity diagram, as a versatile and easy to use tool for behavioral specifications, is also suggested by Barros and Gomes [17]: the authors propose the transformation of activity diagrams into class diagrams with executable code. Also, Chen et al. [28] use the activity diagrams as discussed in Section 1.4.

In our approach, we use activity diagrams to specify object behavior and also to refine it by taking advantage of the hierarchical decomposition of the activity graphs. This hierarchical decomposition gives us the opportunity to navigate at different levels of abstraction of the specification and to identify common behavior that has not been obvious during the object identification process. This step is also composed of several substeps:

- b.1. create an initial activity graph for each object/operation;
- b.2. decompose the initial activity graph into subactivity states, based on the specification of the applied scenario; use numeric tags to trace the relationship between collaborations and operations;
- b.3. identify the states that are operations of the neighboring objects and those that are activities of the local object; update the message ordering in the collaboration diagram, if necessary;
- b.4. decompose the activities hierarchically, based on the application requirements; propagate tags hierarchically;

The example in Figure 3.5-(a) shows the internal behavior of object $\{1.c\}$ *Forward*. After applying substeps 'b.1' and 'b.2', five subactivity states have been identified. Applying substep 'b.3' we observe that, excepting the *Analyze* state,

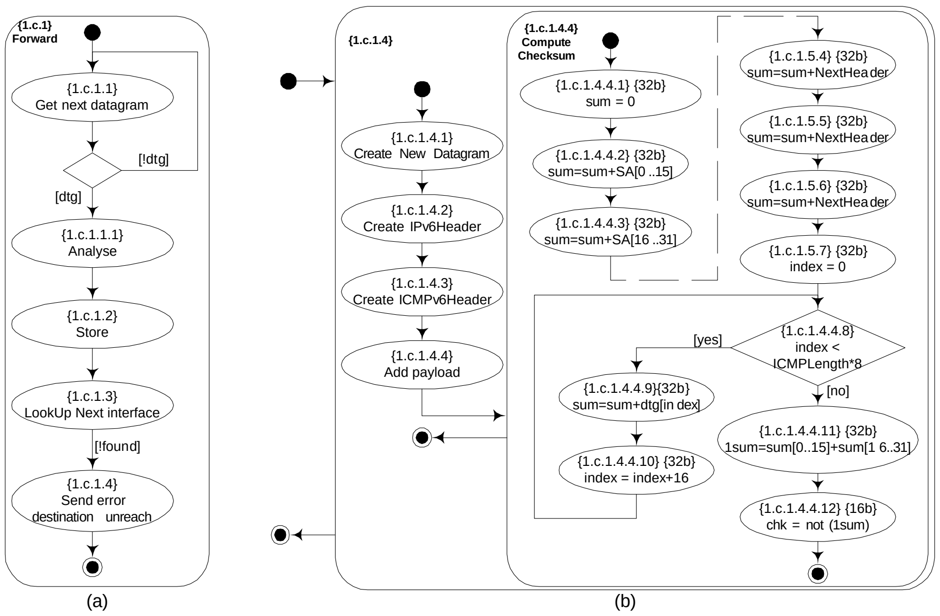


Figure 3.5: (a) Specifying internal behavior based on the applied scenario;
(b) Hierarchical decomposition of Activity Graphs. The dashed arrow represents states that have not been included in the figure due to space reasons

which is an internal computation of the *Forward* object, the rest of the states may be implemented as operations of the neighboring objects. The tags added to each state are derived from the tag of the object (e.g., {*1.c*}) and from the sequence number of the external operation that implements the state (for instance, {*1.2*} for *Store*), which also encodes the number of the scenario in question. As one may notice, the tag of the *Analyze* state has an additional level of hierarchy. This approach was used to mark that the {*1.c.1.1.1*} *Analyze* state precedes the {*1.c.1.2*} *Store* state, but it is executed after the {*1.c.1.1*} *Get next datagram* state. Finally, substep 'b.4' deals only with refining the activity graph of each operation based on the application specification. Figure 3.5-(b) presents an example where the {*1.c.1.4*} *Send error destination unreachable* state is decomposed hierarchically into more detailed activity graphs.

We point out that, during the behavioral analysis step, two kinds of objects may be identified: objects that provide services to other objects, and objects that execute their own state machine (i.e., activity graph) continuously (e.g., {*1.c*} *forward*) to coordinate the control flow of the entire scenario. The objects in the latter category have been investigated by many researchers and different names have been suggested: *permanent* [7], *continuous* [40], or *active* [113]. It is important to note that active objects may themselves provide operations to other objects.

c. Object Refactoring

The functional decomposition of the system into three categories produces rather complex control objects, which is a known problem of such approaches. However, analyzing their behavior one may further decompose the objects into less complex ones by identifying overlapping functionality. Based on this functionality, one can further split or group the objects. During this process, the activity graph of each object/operation may also be refactored. The step is based on several substeps:

- c.1. refine the communication of each collaboration diagram into an asynchronous message passing communication;
- c.2. refine activity graphs that use operations of other objects to support asynchronous communication;
- c.3. remove duplicate operations (i.e., same functionality, different naming) resulting from applying different scenarios to the same object; update the collaboration diagrams of each scenario based on the new naming;
- c.4. identify complex control behavior in objects and extract it into separate objects; (a). extract complex control behavior from data and interface objects; (b). identify common behavior in different objects and group/split objects accordingly.

At previous steps, object interaction has been specified using service requests. This interaction may be classified into two categories. The first category includes requests that invoke an operation of another object, after which the control is passed to that object and, when the operation completes its execution, the control is returned to the initial object. The second category includes requests that pass the control to another object, while the activity graph of the emitting object continues the execution without waiting for a reply. We refine these types of interactions using an asynchronous message passing mechanism. An asynchronous message passes the control to the target object, while the source object continues the execution of its activity graph. Following this approach, the interaction between objects is transformed from synchronous to asynchronous during substep 'c.1'. In practice, this means that the messages in the first category are transformed into a forward message to invoke an operation and a backward message to announce its completion, respectively. For the second case, the initial message only translates into a forward message that passes the control to the target object. In substep 'c.2', the activity graphs of the objects involved are modified to support asynchronous communication. Send signal and receive signal events are used to represent the sending and the receiving of a message, respectively. An example where an object (i.e., *Forward*) passes the flow of control to another object via a message (i.e., *DestUnr*) and continues its execution is given in Figure 3.6. In comparison, Figure 3.7 shows an example where the same object sends a message (i.e., *addr*) to another object and waits for the message (i.e., *int*) announcing the completion of the second object's operation.

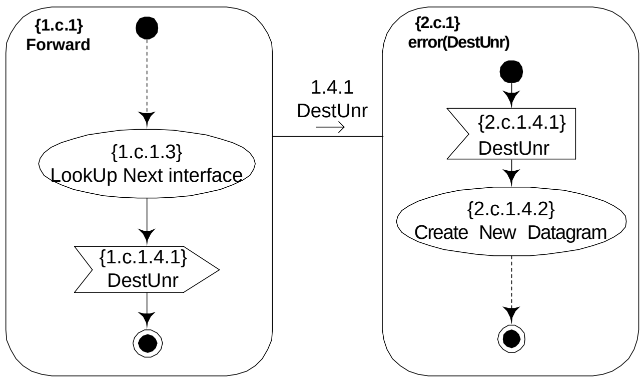


Figure 3.6: Example of Asynchronous Communication refinement

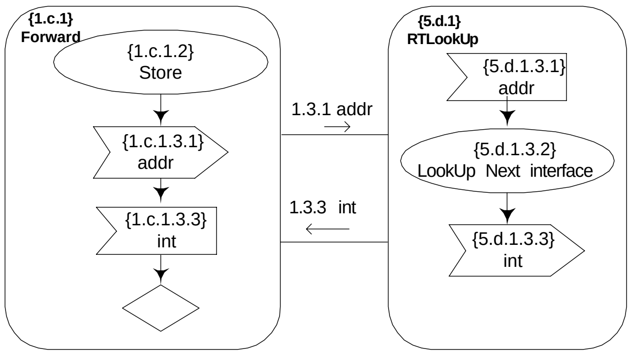


Figure 3.7: Example of Synchronous Communication refinement

In substep 'c.3', all the scenarios applied are overlapped such that each object contains the operations identified by each scenario in part. We analyze the operations of each object from the point of view of their functionality and, if operations with similar functionality are present, we keep one and eliminate the others. Consequently, the collaboration diagrams, in which the eliminated operations have been used, are updated to use the preserved operation. An example is the one of the *send_error* object that, being part of several scenarios, contains two operations for checksum computation, each originated by a different scenario. We mention here, that the identification of overlapping functionalities of the objects should also be performed in early stages of the behavioral analysis, namely after applying substeps 'a.3' and 'b.3', when the effort of refactoring the collaboration diagrams is smaller. We have inserted it here to be used as a final guideline before proceeding to the next steps of the process.

Finally, substep 'c.4' is the most complex step of the refactoring process. The purpose of this step is to identify and isolate common functionality in objects. Based on this, one may suggest the splitting of complex objects into more manageable parts, or grouping the objects providing similar functionality into a single object. The approach targets mainly control objects, since they are the ones implementing the behavior of the system. One has to notice, though, that during this refactoring step, the activity diagram of the objects involved is also refactored. One example may be the checksum computation provided by two different objects. In such case, the analyst may decide to move this functionality from the initial objects into a new object that provides such functionality.

Another situation where substep 'c.4' may be useful is in the identification of pieces of functionality that have not been detected in the previous steps. This substep applies mainly to interface and data objects. For instance, in Figure 3.4, a message *RTLookUp(addr)* is used to query the data stored in the $\{5.d\}$ *routingTable* object. By decomposing the activity graph of *RTLookUP()* operation, we have identified a complex algorithmic computation needed for querying the data of the routing table. Such computation should not be situated inside data objects, since they are considered bared of control. Therefore, one may decide to move the routing table interrogation to a new object and to refine the communication between the new object and the RT object. This also implies that the *RTLookUp* operation provided by the $\{5.d\}$ *routingTable* object is transformed into a *getRTEntry()* operation. Figure 3.8 presents the activity graph of the identified functionality that is "pulled out" of the $\{5.d\}$ *routingTable* object. The decision of whether the new object will be integrated, in next steps, with other control objects (e.g., *forward*) or it will remain as a stand-alone control object is left to the designer.

Creating the Class Diagram. The identification of the objects in the system, along with their operations and behavior, puts the basis for focusing, in the following step of the analysis phase, on classifying the identified objects into classes and creating a class diagram of the system. The class diagram provides a system template from which one may instantiate configurations of the system at design-time. Nevertheless, a class diagram may not always be necessary. In the embedded systems field, many components of the system often have only one instance (i.e., a controller of an elevator), thus classifying them will not bring any additional information.

In case a class diagram of the system is needed to give a better understanding of the specification, it may be easily obtained from the updated collaboration diagrams created for each scenario. By overlapping all the scenarios and taking into account the refactoring process, corresponding classes are derived from the objects. Associations between classes are simply obtained by analyzing the collaborations between different pairs of objects. For instance, if two objects communicate to each other, an association is created between their corresponding classes. More advanced object-oriented features like inheritance, composition, etc. may be used if they are deemed necessary. For the moment, the identification of such features is not taken into account.

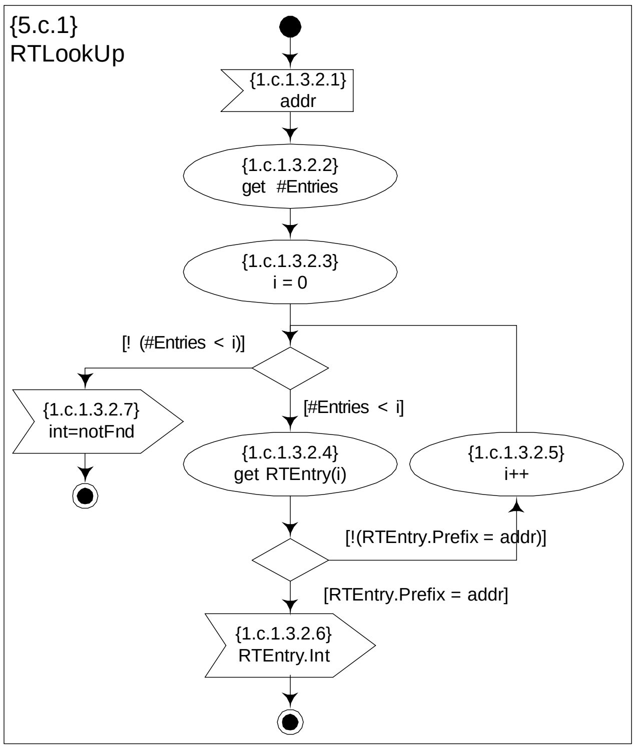


Figure 3.8: Object decomposition based on internal behavior

It is also worth mentioning that, after the completion of the behavioral analysis process, steps 3 and 4 of the 4SRS method may be applied before obtaining the class diagram of the system. For the moment, we consider this approach as subject for future work.

3.4 Summary

We have presented a UML-based methodology for specifying embedded applications. The proposed approach is focused on identifying the functionality of the application starting from its requirements. An existing method for object identification, 4SRS, has been employed to follow a functional decomposition of the system. The method has been complemented with a behavioral analysis process that enabled us to analyze in detail the functionality provided by each identified object. The deliverables of this methodology, that is the objects of the system and the specification of their internal behavior, correspond to the *Functional Application Model* discussed in Figure 2.3.

UML has been used as a means to support the analysis process. The requirements of the application have been captured using use case diagrams, while the components of the system have been represented by object diagrams. In addition, we have proposed the use of activity diagrams as a tool for analyzing and specifying the behavior of the objects. Several benefits from using activity diagrams could be observed. On the one hand, they have been an easy to use and understand notation, allowing us to focus on the computational steps of the application, rather than on its state at a given moment in time. Based on the case study, we have observed that activity graphs are well-suited for specifying data-driven systems, where the interaction with the external environment is limited. Moreover, the use of activity diagrams has provided benefits in terms of hierarchical decomposition and relatively easy refactoring, thus allowing one to suggest refactorings of objects/operations. Due to the hierarchical representation of activities, we have been able to expand or collapse the activities on different levels of detail, and to reuse the same activity graph to implement several activity states.

There are some obvious drawbacks regarding the use of activity graphs. The most noticeable one is that, for large specifications, the diagrams scale up badly and it becomes difficult to edit and manage. The refactoring process has helped us in reducing duplicate sequences of operations to some extent, yet the gain was not enough. Moreover, if the UML tool facilitates the navigation from one abstraction level to another, inside activity graphs, such complexity will be diminished even more.

Only basic features of the UML tools employed have been exploited for supporting the current approach. Further explorations of a good tool support are considered, nevertheless. Future work will be also concerned with investigating an approach in which the attributes of the objects are identified. Currently, the attributes of each object are derived from the parameters of the operations in a rather ad-hoc manner, and we consider that a more systematic approach has to be investigated.

Chapter 4

Combining UML and DFD for Application Specification

In the previous chapter, we have discussed a UML-based methodology for the specification of embedded applications. The main analysis tool of the approach has been represented by the collaboration diagram, which has been used to structurally decompose the system into less complex parts (i.e., objects). The approach was biased towards the identification of the functionality provided by each object via behavioral analysis. During the specification process, we have noticed that there has been a functional perspective of the system that was not adequately represented by the UML diagrams used. It follows that one needs to focus more on the data-flows in the system and on how different system components affect these data-flows. This necessity is largely due to the data-driven characteristic of the application domain that we have been addressing (i.e., protocol processing). Therefore, in this chapter [1], we propose an approach that complements UML with a functional view, to enable the inclusion of a data-flow perspective in the application specification process.

Since its apparition, object-orientation has been constantly gaining popularity over other specification techniques, becoming one of the main tools used for software development. Beside object-oriented methods, other languages have been proposed and used for similar purposes. One such language is provided by the *structured analysis and design* methods [56, 136]. Although quite popular in industrial environments in the 1970s, and even today in certain application areas [55], the structured methods have been largely overshadowed by the object-oriented methods, especially after the introduction of UML. Currently, designers prefer the use of either one or the other in an exclusive manner.

Nevertheless, we consider that both object-oriented and structured analysis methods represent important tools in embedded systems design, each of them (with its own strengths and weaknesses) providing important techniques for specifying

[1]This chapter is based on publications [P.3] and [P.7]

the system under consideration. However, one conceptual model is limited to represent only a specific view of a system, filtering out important details of the specification [37, 81]. Sometimes, several views of the system under development are required to capture all, or at least most of its features and details. Ideally, the designer should benefit from an approach in which several design methods and conceptual models are combined to specify the system at different abstraction levels [9].

The main goal of this chapter is to propose an approach in which the functional and the object-oriented views are inter-played to represent the various modeling perspectives of embedded systems. More specifically, we investigate a way of combining the main modeling tool of the traditional structured methods (i.e., the data-flow diagram) with UML-based models. The rationale behind the approach is that both views are important for modeling purposes, in embedded systems environments, and thus, the designer can benefit from their combined use for developing complex systems.

The chapter is organized as follows. We start with a brief presentation of the structured methods and of the data-flow diagrams. Next, we provide an argumentation in favor of integrating the structured and object-oriented methods for the specification of embedded systems. For this purpose, we adopt three combinations proposed by Fernandes [45], and we briefly present them. After that, we propose a methodology for specifying embedded applications that combine the three suggestions in a consistent manner. We focus our attention on defining a systematic approach in which the transition between the UML and DFD models is performed. Then, we present the tool support provided for the methodology, where the transformations between steps are implemented using model scripts.

4.1 Structured Analysis

Structured Analysis (SA) is a method for the analysis of system specifications, introduced by Tom DeMarco in late 1970s [36]. Three perspectives of the system are taken into consideration: *functional* (i.e., data-flow diagrams), *data* (i.e., entity-relationship diagrams), and *dynamic* (i.e., state transitions diagrams).

The main tool of SA is the *data-flow diagram* (DFD). DFDs use four symbols to represent a system at different levels of detail: *data-flows* (movement of data in the system), *data-stores* (repositories for data that is not moving), *processes* (transformation of incoming data-flows into outgoing data-flows), and *external entities* (sources or destinations outside the boundary of the system). DFDs provide an activity-based behavioral view of the system, also designated as dynamic or functional, useful for describing transformational systems, such as digital signal-processing systems, multimedia systems, or telecommunication devices. All the elements of a DFD come in two flavors: data and control, respectively. *Control processes* process (and produce) events, while data-processes (also referred to as *data transformations*) process and produce data. Consequently, we can have *data-*

flows and *data-stores* to handle and store data, and *control flows* and *control stores* to handle and store events, respectively.

A DFD specifies the system in a hierarchical manner. At the highest level, a context diagram (containing exactly one data-process) is used to specify the interfaces between the system and its external environment, as well as external entities interacting with the system. The interaction is depicted in terms of flows to and from the system. In the lower layers, the top-level data-process is refined into more complex DFDs. The processes on the lowest level (i.e., the leaves) are specified using state machines for control processes, while for data-processes no strict language is specified, typically pseudocode being used.

4.2 Combining DFD and UML

Reading through the UML specification, we have found the following statement:

> "A frequently asked question has been: Why doesn't UML support data-flow diagrams? Simply put, data-flow and other diagram types that were not included in the UML do not fit as cleanly into a consistent object-oriented paradigm. Activity diagrams and collaboration diagrams accomplish much of what people want from DFDs [...]" [94, p. 1-3]

In our opinion, this statement could be interpreted in two ways: firstly, from a conceptual point of view, object-oriented approaches do not require the use of a data-flow model; secondly, if such data-flow model is really necessary, activity and collaboration diagrams may be used to provide graphical notations for it. Indeed, for traditional object-oriented methodologies, DFDs may seem inadequate. However, although UML is regarded as a general-purpose modeling language (nevertheless, not perfect [44]) a data-flow paradigm may be necessary for certain application types. In fact, several authors pointed out the need of complementing the object-oriented view with a functional perspective [8, 9, 67, 137]. Of course, for applications whose characteristics require either a pure object-oriented or a pure functional approach, such a combination is pointless. Authors have shown that, when the right method is applied to the right application domain, substantial benefits are obtained [52, 75, 130]. In this work, we intend to provide a solution for those application domains that would benefit from the dual view provided by the two methods.

There are both similarities and differences between the two paradigms. For instance, both paradigms use a functional, a control and a data perspective of a system. In turn, they put the focus differently on which one of them represent the main artefact of the specification; furthermore, differences are also met in the order in which these models are created and used. Typically, object-oriented methods have the class diagram as their main modeling tool, whereas structured methods use the DFD as its main diagram.

For a data-flow approach to be really beneficial, it is not enough to select a UML diagram as a graphical notation, but one has to integrate it into a design methodology that specifies under what circumstances DFDs may be used, and what can they be used for. In addition, tool support for such an approach is vital, not only in providing editors for creating data-flow models, but also in assisting different transformational steps of the process.

Fernandes has proposed three possible combinations of integrating DFDs with UML [45]. In all three situations, both an object-oriented model and a functional model are used in combination, allowing one to capture the details promoted by each of the paradigms. We have decided to adopt these three suggestions as the basis of our methodology and therefore, we briefly discuss them, in the following, with the permission of their author.

DFDs to refine use case models. This combination is based on the observation that both the use case model and the DFD model provide a similar functional perspective of the system, useful to capture the requirements of the application. In addition, the DFD also models the interaction between different pieces of functionality from a dynamic perspective, whereas the relationships between use cases may only be modeled from a static perspective via ≪*include*≫, ≪*generalize*≫, and ≪*extend*≫ relationships. Therefore, the transformation of a use case model into a DFD model is proposed, in order to allow the designer to model the interaction (i.e., the data-flows) between use cases. To provide a systematic approach, Fernandes proposes the use of an intermediary step, in which the objects of the system are identified from use cases first, followed by a transformation of these objects into DFD artifacts. As a practical solution, the use of the 4SRS method [48] (that has already been discussed in Chapter 3) is suggested. The approach is favored by the fact that the 4SRS method uses a functional decomposition, of the system where control, interface and data objects are identified. With this categorization of objects, the control and interface objects may be transformed into data-processes, the data objects into data-stores, the actors into external entities, and the relationships between actors and use cases into data-flows.

DFDs to refine the behavior of a system component. The objects obtained as a result of applying the 4SRS method should be regarded as complex components or subsystems of a given system. Therefore, one has to be able to decompose them further, in order to tackle their complexity. Consequently of transforming objects into DFD elements, one may take advantage of the hierarchical decomposition of DFDs to analyze their internal behavior in more detail. The approach may be seen somewhat similar to the activity diagram-based behavioral analysis process discussed in the previous chapter, yet the emphasis is put on data-flows in the system and on the processing affecting the data, rather than on the control flow sequencing provided by the activity diagrams.

DFDs to be transformed into object/class diagrams. As a third option in combining DFDs and UML diagrams, DFDs can be transformed into an object-oriented model of the system [45]. The author argues that this approach might be beneficial in two situations: a) in re-engineering activities, when the program code written in some structured programming language is initially transformed into a DFD model, from which an object-oriented model is obtained; b) in situations where the specification of the system is expressed using a DFD model, and the designer wants to transform it into an object-oriented model that may be implemented following an object-oriented programming language.

4.3 A DFD-UML Methodology for Application Specification

In this section, we propose a methodology for specifying embedded applications, which combines the previously identified possibilities to integrate DFD and UML. The methodology that we present here, should be regarded as an alternative to the methodology proposed in Chapter 3. The main goal of the methodology is to provide a practical approach of merging DFDs with other UML models, with the main emphasis put on defining a systematic approach to transform the models of the system. We will show in the next section how we have automated the defined transformations in a UML tool.

The methodology (Figure 4.1) defines a set of models of the system, a process for integrating these models, and a number of model transformations for going from one model to another. The IPv6 router specification introduced in Section 3.1 is used to illustrate the approach. The main phases of the methodology are:

a. extract application requirements
b. create the use case diagram (UCD)
c. specify each use case in a textual manner
d. transform the UCD into an Initial Object diagram (IOD)
e. refactor IOD (group, split and discard objects based on their functionality)
f. transform the IOD into a DFD
g. identify data-flows and build a data dictionary
h. specify process behavior using activity diagrams
i. (1) transform the DFD into an Object diagram (OD) *or* (2) transform the DFD view into a Class diagram (CD)

As one may notice, the perspective from which the system is modeled is changed several times during the process. This approach enables the designer to focus on the specific details provided by each perspective. The presented process may be seen as being composed of two parts. Initially, a UML model (i.e., the use case diagram) is transformed into a DFD model, to provide a data-driven perspective of the system. The resulting DFD model is then transformed back into a UML model

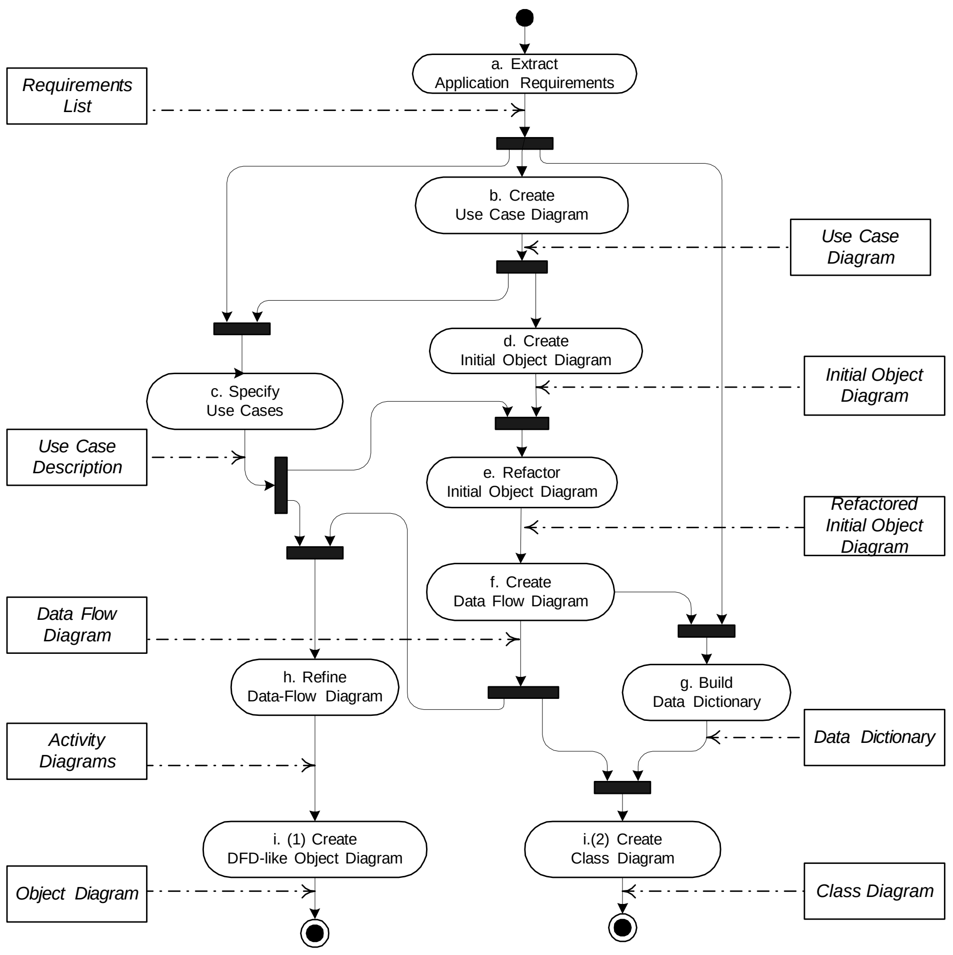

a. Extract Application Requirements
Requirements List
b. Create Use Case Diagram
Use Case Diagram
c. Specify Use Cases
d. Create Initial Object Diagram
Initial Object Diagram
Use Case Description
e. Refactor Initial Object Diagram
Refactored Initial Object Diagram
f. Create Data Flow Diagram
Data Flow Diagram
h. Refine Data-Flow Diagram
g. Build Data Dictionary
Data Dictionary
Activity Diagrams
i. (1) Create DFD-like Object Diagram
i.(2) Create Class Diagram
Object Diagram
Class Diagram

Figure 4.1: Integration of UML and DFD models into a model driven methodology

(i.e., object or class diagram) to obtain a structural view of the system. The following discussion focuses on how the system specification is transformed through the steps of the methodology, in a systematic manner, such that the transformations between the steps are easy to automate.

4.3.1 From UML to DFD

The first part of this methodology resembles the starting point of the methodology proposed in Chapter 3. Similarly, equipped with the application requirements (i.e., IPv6 router) we create a use case diagram. For simplicity, we use the diagram of Figure 3.2, and since the steps 'a' to 'e' of the approach have already been discussed in the previous chapter, we omit them here. Still, we mention that step 'd' corresponds to *step 1* of 4SRS, while step 'e' corresponds to *step 2* of 4SRS.

f. Create the Data-flow Diagram

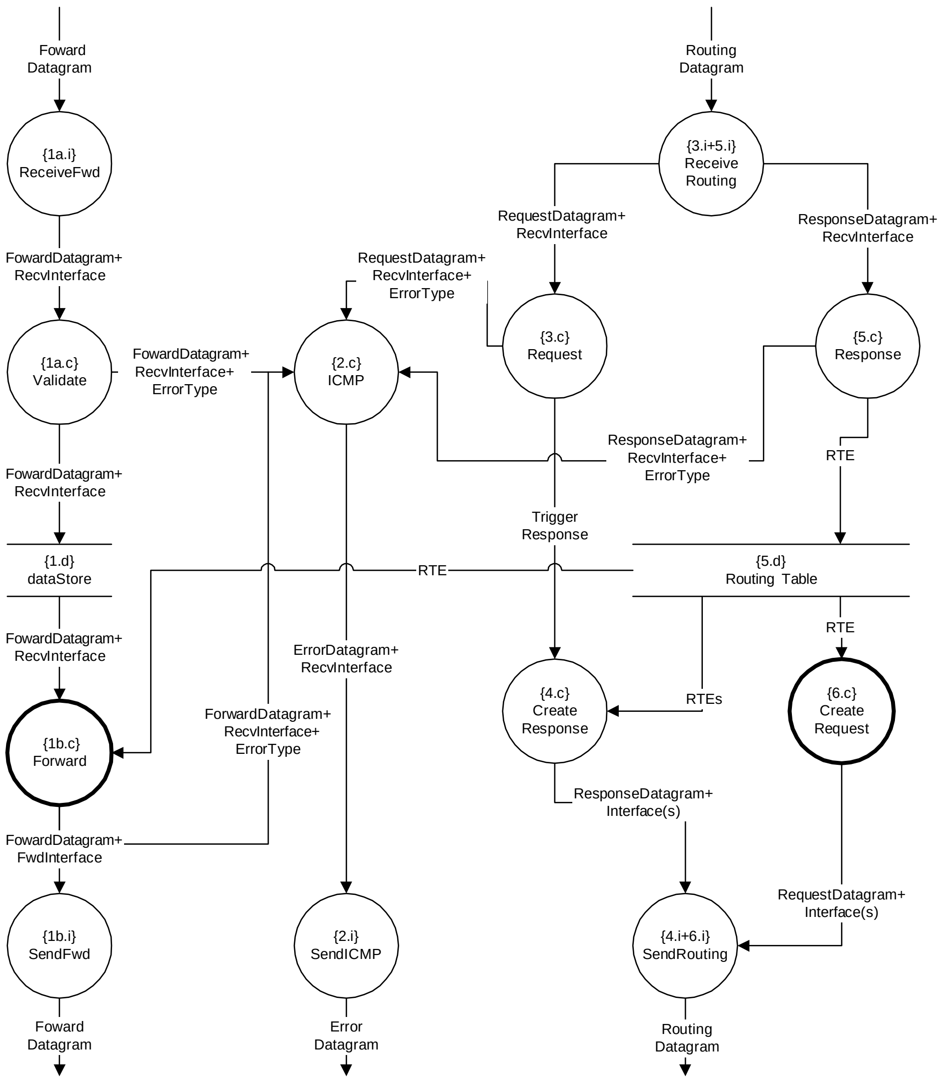
Data-flow diagrams are used, to identify, classify and refine data-flows involved in the system. In our approach, the data-flow model of the system is obtained by transforming the identified set of objects (see Figure 3.3) into DFD elements. The process is performed in several substeps:

- f.1. transform each *actor* in the IOD into an *external entity*;
- f.2. transform *interface* and *control objects* into *data-processes*;
- f.3. transform *data objects* into *data-stores*;
- f.4. transform *associations* between actors and objects into *data-flows*;
- f.5. transform *associations* between objects into *data-flows*;
- f.6. identify data-flow types by applying scenarios;
- f.7. refactor the data-flow diagram;
- f.8. identify and mark active processes in the DFD.

Steps 'f.1' to 'f.5' are straight forward, due to the one-to-one mapping between the elements of IOD and DFD. In the IOD, we have control and interface objects that support and process the communication with the external environment (i.e., *router* and *node*). This is close to the behavior of the data-processes provided by the DFD concepts, allowing us to transform all control and interface-objects into data-processes. Similarly, data objects in the IOD are transformed into data-store elements in DFDs. Actors in the IOD are transformed into external entities in DFD. Finally, the associations between objects, and between objects and actors, respectively, are simply transformed into data-flows. During the transformation, we also propagate the tag numbers of the initial objects to the DFD model.

Step 'f.6' focuses on identifying the data-flows in the system by applying use case scenario. The type of data that each flow carries is directly identified from the application requirements and associated documents (e.g., protocol specifications). Additionally, the direction of each data-flow is identified.

The resulting DFD may be refactored, in step 'f.7', by decomposing or grouping together DFD elements of the same category. For instance, processes that exchange the same information (i.e., they input or output identical data-flows) with the same external entity or data-process, and also in the same direction, are grouped together. In our example, both the data-processes originating from objects {5.i} and {3.i} (Figure 3.3) receive *routing datagrams*, either as *request datagram* or *response datagram*, from the *router*. Therefore, these data-processes may be conjoined into a single process {3.i+5.i} (Figure 4.2). A data-process originating from an interface object that communicates bidirectionally with an actor may be split into two separate data-processes (e.g., {*1a.i*} and {*1b.i*}), one dealing with the incoming and the other with the outgoing traffic. A similar approach could have been applied for obtaining the {*2.d*}, {*3.d*}, {*4.d*} and {*6.d*} data-stores, if their elimination had not been performed during *step 2* of the 4SRS method. In fact, it seems



Foward Datagram
Routing Datagram
{1a.i} ReceiveFwd
{3.i+5.i} Receive Routing
RequestDatagram+ RecvInterface
ResponseDatagram+ RecvInterface
FowardDatagram+ RecvInterface
RequestDatagram+ RecvInterface+ ErrorType
{3.c} Request
{5.c} Response
{1a.c} Validate
FowardDatagram+ RecvInterface+ ErrorType
{2.c} ICMP
ResponseDatagram+ RecvInterface+ ErrorType
RTE
FowardDatagram+ RecvInterface
Trigger Response
{1.d} dataStore
RTE
{5.d} Routing Table
FowardDatagram+ RecvInterface
ErrorDatagram+ RecvInterface
RTE
{4.c} Create Response
RTEs
{6.c} Create Request
ForwardDatagram+ RecvInterface+ ErrorType
{1b.c} Forward
ResponseDatagram+ Interface(s)
FowardDatagram+ FwdInterface
RequestDatagram+ Interface(s)
{1b.i} SendFwd
{2.i} SendICMP
{4.i+6.i} SendRouting
Foward Datagram
Error Datagram
Routing Datagram

Figure 4.2: Data-Flow diagram for the IPv6 router. Active processes are drawn in thicker lines

that *step 1* of the 4SRS method should be the only one used to intermediate the transformation of the use case diagram into a DFD, while it is more natural that all the object eliminations and refactorings are performed in the DFD. The refactoring process is currently performed manually. Nevertheless, tool support is envisioned in the form of a set of design guidelines and metrics that are automatically presented to the designer to suggest possible refactorings.

Most of the DFD approaches known in literature do not make a clear distinction among processes, with respect to their behavior. We may observe two types of behavior: processes that start their execution when one of their input data-flows becomes active, and processes that execute continuously, regardless of the status of their inputs. We call them *reactive processes* and *active processes*, respectively. A process is considered to be active if it has no input flows from other processes or its behavior is self-triggered (output flows are fired without an input flow triggering the process). One example of an active process would be a process that is periodically reading a data-store (e.g., processes $\{1b.c\}$ and $\{6.c\}$ in Figure 4.2). The input data-flows of active processes are triggered by the processes themselves and not by the data-stores, despite the fact that there is a data-flow from the data-stores to the mentioned processes. In contrast, the reactive processes are those whose behavior is triggered by an input data-flow, and which, in turn, trigger the behavior of other processes. Following this rationale, substep ’f.8’ of the approach classifies processes into active and reactive to help the designer in specifying, at the following steps, the internal behavior of each process. The diagram resulting after applying step ’f’ is shown in Figure 4.2.

g. Building the Data Dictionary

We classify the data-flows involved in the system by building a *data dictionary*. This is done by gathering the types of data transported by the flows. Additionally, a decomposition of the initial data types into more detailed ones may be obtained from the application requirements. Based on this decomposition, refinements of the data-processes may be suggested. At the moment, the identification of the data-flows is performed manually and it is based on the designer’s skills. Our future work may examine an approach towards building tool support for creating the data dictionary.

A complete data dictionary specification of the IPv6 router under study may be found in [46]. Below, we only present a small example, in which the datagrams transported through the system (the IPv6 router) are classified and decomposed.

```
Datagram = ForwardDatagram|RoutingDatagram|ErrorDatagram
    ForwardDatagram = IPv6Header+Payload
    RoutingDatagram = IPv6Header+UDPHeader+RIPMessage
    ErrorDatagram = IPv6Header+ICMPv6Message
```

h. Refining the Data-flow Diagram

The data-flow diagram obtained at step ’f’ may be further refined by decomposing the data transformations into other DFDs. Traditionally, the terminal nodes (the leaves) have their behavior specified using pseudocode. To provide a graphical modeling language instead, we propose the use of activity diagrams as the main tool for specifying the behavior of these processes. The use of activity diagrams for specifying the internal behavior of the objects has been discussed in Chapter 3, and since the process is similar, we skip it here. The only difference is that the signal activities are used to send or receive the tokens carried by the data-flows.

4.3.2 From DFD to UML

In the second part of the methodology, two alternative solutions are proposed. The first alternative transforms a DFD model into an object diagram focusing on the behavior of the system, whereas a more object-oriented approach is suggested as a second alternative, to focus on the data classification in the system.

i. From DFD to Object Diagram

Several rules support the transformation of a DFD into an object diagram (OD). Each *data process* in the DFD is transformed into an *object* in OD, and the *data-flows* among these data-processes are transformed into *links* between objects. In addition, the data-flows become internal attributes and are encapsulated into the objects. Corresponding methods are added to access these attributes. For instance, the initial *ForwardDatagram* data-flow, between the *ReceiveFwd* and the *Validate* processes (Figure 4.2), may be transformed into an attribute of the $\{1a.c\}$ *Validate* object (Figure 4.3); the object is only accessed by the *sendFwdDatagram()* method, while its value is dispatched to the adjacent objects through the *writeFwdDatagram()* and *sendFwdError()* methods, respectively.

Moreover, objects originating from processes placed at the border of the system receive *set()* methods to communicate with the environment, whereas all the other processes have *send()* methods that use the original input data-flows as parameters. A *run()* method is added to the objects obtained from active processes, in order to specify their state machine, whereas *write()* and/or *read()* methods are added to objects originated from data-stores, to provide access to object data. Once the transformation is completed, the names of different elements may be changed to be more meaningful to the designer.

The final object model (Figure 4.3) is very similar to the DFD one, but now we have objects that have an internal behavior and provide services (implemented by methods) to the adjacent objects. The newly created objects are also classified as being active or reactive, based on the processes from which they have been generated. We recall the difference between these objects: the execution of the reactive objects is triggered only when one of their methods is invoked, while the active objects have a state machine (implemented by a *run()* method) that executes continuously.

To summarize, the following steps are performed:

- i.1. *external entities* in DFD are transformed into *actors* in OD;
- i.2. *data-processes* in DFD are transformed into *objects* in OD;
- i.3. *data-flows* between processes are transformed into links between objects;
- i.4. *data-processes* originating from *active objects* receive a *run()* method;
- i.5. objects originating from border processes that deal with input communication receive an attribute corresponding to the input flow and a *set()* method to access that attribute;

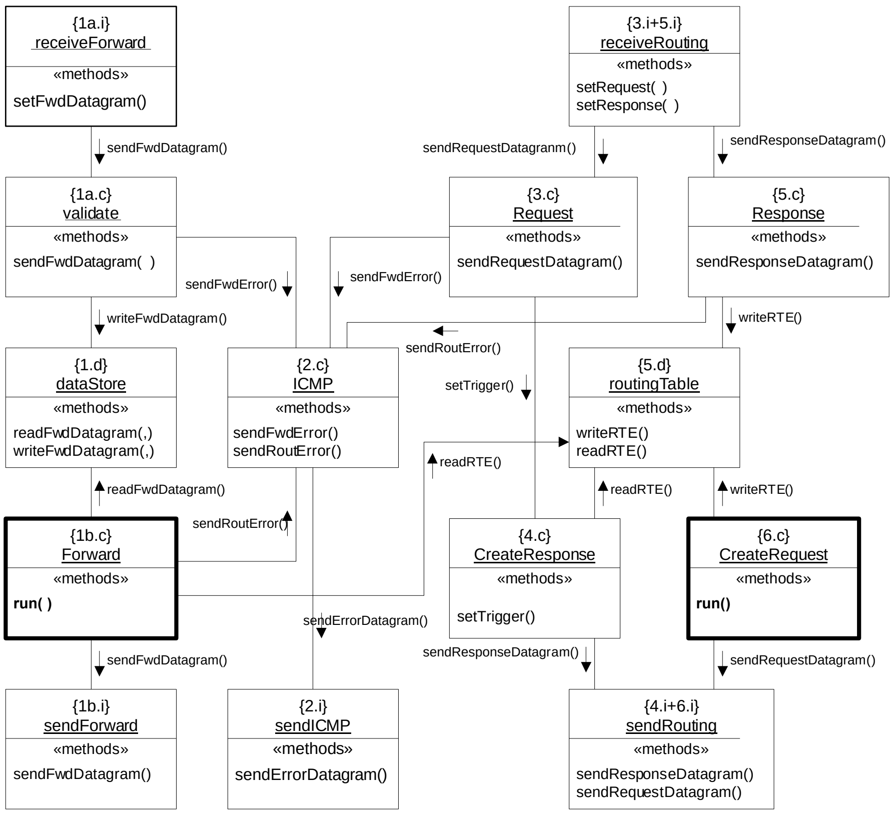


Figure 4.3: The Object diagram of the IPv6 router[3]. Objects drawn in thicker lines depict active objects

i.6. objects originating from non-border processes receive a *send()* method of the incoming data-flow and the corresponding attribute;

i.7. data-stores are transformed into objects that contain *read()* and *write()* methods to provide access to their data.

The object diagram in Figure 4.3 provides a low level of abstraction and data encapsulation. However, it is suited for prototyping purposes and for functional testing of the specification, as it may be easily implemented in an object-oriented programming language. As a proof of concept, a Java prototype of this model has been manually implemented and it may be downloaded from `http://www.abo.fi/~dtruscan/ipv6index.html`.

[3]Due to space reasons, some details (i.e. attributes, actors, etc.) have been intentionally omitted.

j. From DFD to Class Diagram

The second alternative that we propose for obtaining an object-oriented model of the system, adopts a view in which the data involved in the system plays a central role. This approach is not far from the paradigm underlying structured methods, where data classification is the main task. Hence the transformation between the models is based on the classification and encapsulation of data into classes, along with the corresponding methods that operate on this data.

Briefly, the algorithm classifies all the data-flows and the data-stores inside a DFD, based on their type. For each identified type, a corresponding class is created in the class diagram. To identify class methods, three kinds of patterns are used: data-flows communicating with the external environment of the system

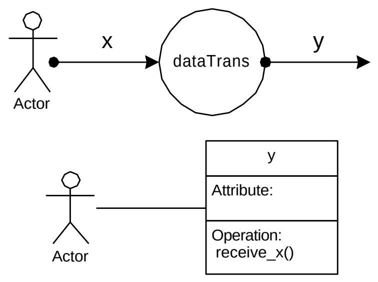


Figure 4.4: Border Process pattern

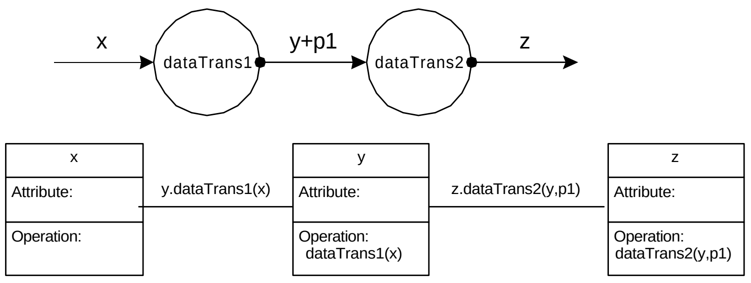


Figure 4.5: Interprocess Communication pattern

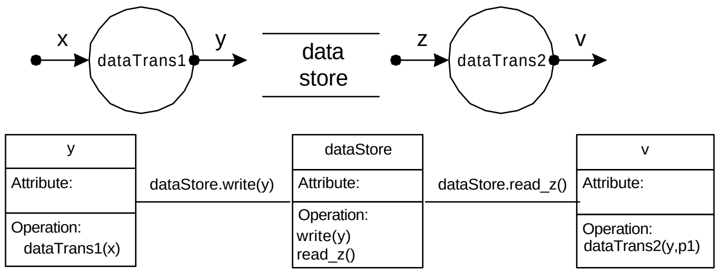


Figure 4.6: Data Store Communication pattern

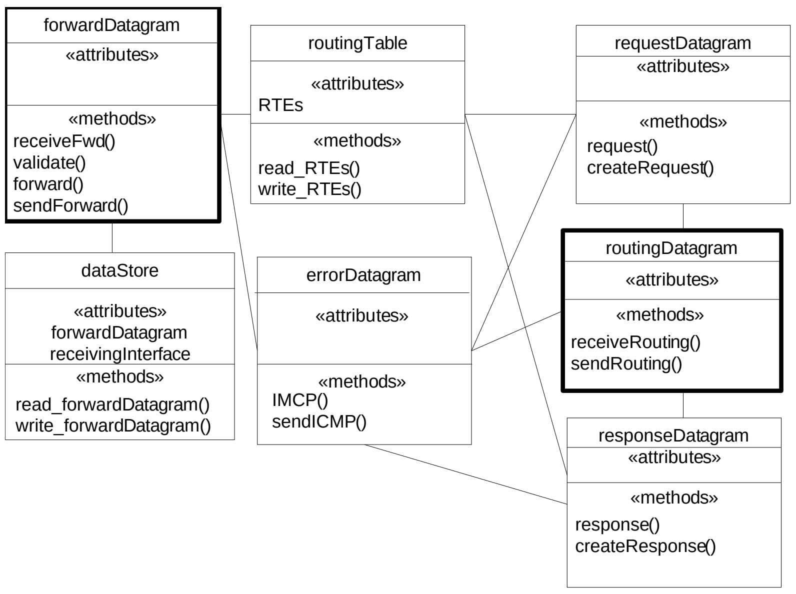


Figure 4.7: The class diagram of the IPv6 router

(Figure 4.4), data-flows between two processes (Figure 4.5), and data-flows that communicate with data-stores (Figure 4.6).

A number of processes, that is data transformations, operate over each data-flow type in a DFD. We transform these processes into logical methods of the class that responds to the data-flow that the processes operate on. For instance, *Forward-Datagram* data-flow (Figure 4.2) is processed by several data transformations (e.g., *ReceiveFwd*, *Validate*, *Forward*, *SendForward*, *ICMP*). Thus, a *forwardDatagram* class is added to the class diagram (Figure 4.7) and the DFD processes that affect this data are added as logical methods of this class. We consider that a process becomes a method of a class only if it has as output the data-flow type modeled by that class. For instance, the *ICMP* process in Figure 4.2 is not transformed into a method of *forwardDatagram* because it outputs an *ErrorDatagram*.

Data-stores receive a special treatment. Each data-store element is transformed into a separate class, unless such a class already exists. The newly created class provides *read()* and *write()* methods to access its data, based on the input and output flows of the initial data-store (see *dataStore* in Figure 4.6). In addition, the attributes of the classes originating from data-stores are identified, based on the data-flows that are input or output to the data-store.

To summarize, the transformation is based on the following steps:

j.1. transform *data-flow* and *data-store* types into *classes*;
j.2. apply the Inter-process Communication pattern;
j.3. apply the Border Process pattern;
j.4. apply the Data Store Communication pattern.

The model in Figure 4.7 has been manually constructed in Java. Due to the object-oriented principles implied by this model, the process was straightforward. The obtained code provides a simulation model of our specification that can be used as executable specification for checking the functionality of the router. The code is not the contribution of the author of this study, but it may be found in [2].

4.4 Tool Support

Appropriate tool support is required for benefiting from a model driven approach, fully. On the one hand, tools should provide means to (graphically) create, edit and manipulate model elements. On the other hand, scripting facilities for implementing automated manipulation and consistency verification of such models have to also be provided, in order to speed up the design process and cut down development times. We discuss, in this section, how several steps of the previously defined methodology are automated using model scripts.

4.4.1 The SMW Toolkit

To support our approach, we have used the Software Modelling Workbench (SMW) tool, available for download at [3]. The tool is built according to the OMG's MOF and UML standards, allowing to edit, store and manipulate elements of several metamodels. SMW uses the Python language [129] (an interpreted object-oriented programming language) to describe the elements of a metamodel, each element being represented as a Python class. Python scripts are used to query and manipulate models. Moreover, making use of the *lambda*-functions of the Python language, OCL-like idioms are supported.

SMW may be customized to support new user-defined metamodels. The consistency of the models is enforced via well-formedness rules coded in the metamodels using OCL-like constructs. Tool profiles may be created by defining custom editors for each new metamodel. The *UML14 profile* [102] is currently the default tool profile of SMW and it supports the UML 1.4 metamodel. In addition, an *SA/RT profile* provides support for a MOF-based metamodel for structured analysis [61], allowing one to graphically create, edit and manipulate data-flow models.

The SMW tool, along with its UML14 and SA/RT profiles, has been used to provide support for the methodology discussed in the previous section. We show in the following how the specified model transformations are implemented using the scripting facilities of SMW.

4.4.2 Implementing Model Transformations

The *model transformation* is seen by many authors as the fundamental tool of the model driven paradigm, which enables the evolution of the initial specifications into final ones [116]. A model transformation takes a source model expressed in a given language and transforms it into a target model expressed either in the same language or in a different one. During a transformation, two elementary operations are performed: *queries* and *elementary transformations*. Queries are used to gather information, (e.g., collection of elements that meet a given condition or certain metrics that provide design quality information) from a model. Elementary transformations operate over model elements by creating, modifying and deleting them. The main difference between queries and element transformations is that queries are free of side-effects; they only provide information on a given model, but do not modify the model. Usually, when performing model transformations, both operations are involved. That is, one has to select the source elements using queries and then to transform them into target elements by applying several elementary transformations.

In the following, we present two such transformations that support the design flow in Figure 4.1, namely steps 'd.' (transforming a use case diagram into an initial object diagram) and 'i.' (transforming a data-flow diagram into an object diagram). The rationale behind these transformations and the corresponding algorithms have been presented in the previous sections; here, we only intend to describe the practical aspects involved in implementing such transformations. The scripting facilities of SMW have been used, to implement the transformations in the tool, certainly benefiting from the Python language and the OCL-like idioms.

As a precursory note, we mention that in our approach, due to tool restrictions, one is not allowed to create an object diagram before having a corresponding class diagram in place. Therefore, for these transformations, a class diagram is first created and then an object diagram is obtained by instantiating its elements. However, the approach can be changed in case a less restrictive tooling environment is employed.

From Use Case Diagram To Initial Object Diagram

We discuss in here the implementation of the script supporting the first part of step 'd.' of the methodology, namely the transformation of a use case diagram into an initial object diagram. The script provides an example of a model transformation between models of the same metamodel (UML). As an initial step, the source model (use case diagram – *UseCases*) is loaded and a corresponding target model (i.e., *InitialObjectDiagram*) is created.

```
ucModel=io.loadModel("UseCases.smw")
oModel=Model(name="InitialObjectDiagram")
elementMap={}
```

The last line of the above code initializes a Python dictionary to keep track of how elements of the source model map to elements of the target model. Basically, a Python *dictionary* is a collection of elements indexed by a unique "key" element.

Once the source model is loaded, all model elements of type *Actor* are collected using an OCL-like query (lines 4-5). For each identified element, a new Actor is created (line 7) and added to the target model (line 8). The pair of actors is saved in the *elementMap* dictionary (line 9).

```
4 ucActors=ucModel.ownedElement.select(lambda x:
5   x.oclIsKindOf(Actor))
6 for act in ucActors:
7   p=Actor(name=act.name)
8   targetModel.ownedElement.append(p)
9   elementMap[act]=p
```

Similarly, we collect all the use case elements of the source model (lines 10-11) and, for each element found, three corresponding classes are created (lines 13, 19, 24) and added to the target model (lines 29-31). Each class bears the name of the use case that generated it, a stereotype specifying its category (lines 14, 20, 25), and a tag (lines 15-17, 21-23, 26-28) based on the tag name of the original use case. For instance, the use case $\{$*1.*$\}$ *Forward Datagram* (Figure 3.2) will generate a ≪*control*≫ class labeled $\{$*1.c*$\}$ *Forward Datagram* (Figure 3.3).

```
10 useCases=ucModel.ownedElement.select(lambda x:
11 x.oclIsKindOf(UseCase))
12 for el in useCases:
13   p1=Class(name=el.name)
14   p1.stereotype.append(Stereotype(name = "interface"))
15   p1.taggedValue.append(UML14.TaggedValue(
16       name=el.taggedValue[0].name+".i",
17       dataValue=None,modelElement=p1,type=td))
18   elementMap[el]=p1
19   p2=Class(name=el.name)
20   p2.stereotype.append(Stereotype(name = "control"))
21   p2.taggedValue.append(UML14.TaggedValue(
22       name=el.taggedValue[0].name+".c",
23       dataValue=None,modelElement=p1,type=td))
24   p3=Class(name=el.name)
25   p3.stereotype.append(Stereotype(name = "data"))
26   p3.taggedValue.append(UML14.TaggedValue(
27       name=el.taggedValue[0].name+".d",
28       dataValue=None,modelElement=p1,type=td))
29   targetModel.ownedElement.append(p1)
30   targetModel.ownedElement.append(p2)
31   targetModel.ownedElement.append(p3)
```

Optionally, in specific situations, the links (i.e., associations) between the objects (classes) of the initial object set have to be identified from the use case diagram. To provide an automated discovery process, we use the following rules: a) the communication between interface and data objects is only allowed through control objects; b) the communication between actors and the system is only performed through interface objects. This approach provides a preliminary step in identifying the communication between the objects in the initial set, and serves as a basis for future refinement of this communication.

Applying rule a), associations are added between ≪*interface*≫ and ≪*control*≫, and between ≪*control*≫ and ≪*data*≫ objects, respectively (lines 32-33).

```
32 a1=addAssoc(p1,p2,"")
33 a2=assAssoc(p2,p3,"")
```

The mechanism for adding an association between two classes is presented in function *addAssoc()* below. The function receives as arguments two class identifiers from the target model and the corresponding association in the source model, and returns a new association element between the two classes in the target model. Initially, a new UML *Association* element is created and added to the target model (lines 35-36). In the UML1.4 metamodel, the element that links an *Association* to a model element is *AssociationEnd*. The relationship between *Association*, *AssociationEnd* and *Class* elements is as follows: an *Association* has two *AssociationEnd* elements corresponding to its endings, and each *AssociationEnd* element is contained in the *.association* property of a given *Class*.

Thus, to add an association (i.e., *as1*) between two existing classes (i.e., *c1* and *c2*), two new *AssociationEnd* elements, *ase1* and *ase2*, are created (lines 37-40). Additionally, they are linked to the *p1* and *p2* classes through the *participant* property of the *Association* element and they are set to belong to *as1*. Consequently, the *ase1* and *ase2* elements are added (lines 41-42) to the *association* property of the connected classes *c1* and *c2*, respectively. Finally, the newly created association is stored in the *elementMap* dictionary (line 43).

```
34 def addAssoc(c1, c2, as):
35   as1=Association(name=as.aname)
36   targetModel.ownedElement.append(as1)
37   ase1=AssociationEnd(participant=c1,
38    association=as1,multiplicity=None,isNavigable=1)
39   ase2=AssociationEnd(participant=c2,
40    association=as1,multiplicity=None,isNavigable=1)
41   c1.association.append(ase1)
42   c2.association.append(ase2)
43   elementMap[as]=as1
44   return as1
```

For the second rule, the transformation script adds associations between actors and ≪*interface*≫ classes. These associations correspond to the associations between actors and use-cases in the source model. Thus, all these associations in the use case diagram are selected (lines 45-46) and, for each association, the elements that it connects (i.e., Actor-UseCase pairs) are identified (lines 47-53). To add associations between the corresponding pairs of elements in the target model, the *addAssoc* function is invoked, using the mapping information stored in the *elementMap* dictionary (line 54).

```
45 ucAssoc=ucModel.ownedElement.select(lambda x:
46   x.oclIsKindOf(Association))
47 ucAssoc.select(lambda as:
48   ucModel.ownedElement.select(lambda x:
49       x.oclIsKindOf(Actor) and
50       as.connection[0] in x.association and
51       ucModel.ownedElement.select(lambda y:
```

```
52          y.oclIsKindOf(UseCase) and
53          as.connection[1] in y.association and
54          addAssoc(elementMap[x], elementMap[y],as))))
```

The reader may notice our use of OCL-like constructs in an imperative manner. This is allowed by the way the OCL constructs are implemented in Python, using *lambda*-functions, which improves the flexibility and the level of abstraction of the scripts considerably. Alternatively, nested loops could have been used, at the expense of a larger code size and lower execution speed.

From Data-flow Diagram to Object Diagram

In the second example, we present parts of the model transformation script supporting the step 'i.' in Figure 4.1 (i.e., transforming a DFD into a object diagram). The script provides an example of a model transformation between a source model, expressed in a given language/metamodel (i.e., DFD), and a target model expressed in a different language/metamodel (i.e., UML). The algorithm and rationale behind this transformation have been presented in Section 4.3.2. Similarly to the previous example, the script loads a source model and creates a new target UML model (lines 1-2).

```
1 dfdModel=io.loadModel("dfdInput.smw")
2 targetModel=UML14.Model(name=dfdModel.name)
```

Then, the top-level data transformation of the DFD model (i.e., *topDfd*) is identified (lines 3-4), and model (element) information from its lower-level DFD is collected by applying a number of OCL-like queries (lines 5-13). The low-level DFD, which is a refinement of the top-level one (i.e., *topDFD*), contains the data-flow diagram on which we focus our example (Figure 4.2).

```
 3 topDfd=dfdModel.ownedElement.select(lambda x:
 4    x.oclIsKindOf(DataTransformation))[0]
 5 ee=dfdModel.ownedElement.select(lambda x:
 6   x.oclIsKindOf(ExternalEntity))
 7 dt=topDfd.ownedElement.select(lambda x:
 8   x.oclIsKindOf(DataTransformation))
 9 df=topDfd.ownedElement.select(lambda x:
10   x.oclIsKindOf(DataFlow) and
11   not x.oclIsKindOf(DataStore))
12 ds=topDfd.ownedElement.select(lambda x:
13   x.oclIsKindOf(DataStore))
```

Next, the script transforms each *External Entity* in the DFD into a UML *Actor* in the UML model, ensuring that an actor is not added to the class diagram more than once (lines 14-18). As in the previous example, the *elementMap* dictionary is used to store (line 19) pairs of source-target elements of the two models.

```
14 elementMap={}
15 for e in ee:
```

```
16   if e not in elementMap:
17       act=Actor(name=e.name)
18       targetModel.ownedElement.append(act)
19       elementMap[e]=act
```

To obtain the class diagram of the system, the following steps are performed. Firstly, for each *DataTransformation* and *DataStore* element in the *dfdModel*, a new object (i.e., *Class*) element is added to the *targetModel* (lines 26-29). Adding a new class to the UML model is provided by function *addClass()*. We use again the *elementMap* dictionary to store the correspondence between the source and the target elements (line 24). Also, the use of OCL-like constructions in an imperative manner may be observed in line 29.

```
20 def addClass(initialElement, className):
21   if initialElement not in elementMap:
22       newClass=Class(name=className)
23       targetModel.ownedElement.append(newClass)
24       elementMap[initialElement]=newClass
25   return newClass
26 topDfd.ownedElement.select(lambda ts:
27   (ts.oclIsKindOf(DataTransformation) or
28   ts.oclIsKindOf(DataStore)) and
29   addClass(ts, ts.name))
```

In the following step, the script creates the associations between the newly added classes. As presented in Section 4.3.2, we have three different cases of associations between elements based on the source and target elements of the data-flows in the DFD model. The first case is that of an initial data-flow linking two data transformations. For each pair of source-target data transformations, their corresponding classes are identified in the *targetModel* and a *send()* association is added (lines 30-39), using the function *addAssoc()* presented above.

```
30 topDfd.ownedElement.select(lambda f:
31   f.oclIsKindOf(DataFlow) and
32   topDfd.ownedElement.select(lambda src:
33       src.oclIsKindOf(DataTransformation) and
34       f.connection[0] in src.association and
35       topDfd.ownedElement.select(lambda dst:
36           dst.oclIsKindOf(DataTransformation) and
37           f.connection[1] in dst.association and
38           addAssoc(elementMap[src],elementMap[dst],
39           "send"+string.split(f.name,'+')[0]))))
```

The second case addresses data-flows that either originate from or have as target a data-store element in the *dfdModel*. This type of data-flows will be transformed into *read()* or *write()* associations, depending on the direction of the initial data-flow. The third case of creating associations is based on the data-flows that communicate with external entities of the DFD model. This type of data-flows will generate *set()* associations. Since the code of both scripts is similar to the one presented in lines 30-39, we omit it here.

Finally, for those classes linked by associations, one of the classes receives methods and encapsulated attributes corresponding to the name of the associations. The *addAssocMeth* function below is used to add a new operation (method) and a corresponding attribute to a given class (lines 40-47). Following this approach, classes originating from data-stores receive, as attributes, the parameters of the data-flows, and corresponding methods to read and/or write those parameters (lines 48-51).

```
40 def addAssocMeth(clas, methName, prefix):
41   o=Operation(name=methName)
42   if methName not in clas.feature.name:
43       clas.feature.insert(o)
44       attr=Attribute()
45       attr.name="someName"
46       clas.feature.insert(attr)
47   return 1
48 ds.select(lambda t: df.select(lambda f:
49   f.connection[1] in t.association and
50   addAssocMeth(elementMap[t],
51   string.split(f.name,'+')[0],"write")))
```

In both examples, we have intentionally omitted the initialization part and other code that is not relevant to the presented examples. More detailed versions of the presented, as well as of other transformations of the methodology may be found in [127].

4.5 Summary

In this chapter, we have discussed a methodology that combines the tools of the structured and object-oriented paradigms, for the specification of embedded applications. The approach combines the main modeling languages of the two paradigms, namely the DFD and UML diagrams, to allow the designer to focus during the modeling process on different perspectives of the system. The main goal of this integration is to allow the analyst to focus on views not provided by UML, and to give one the possibility to follow an object-oriented approach, if desired. Thus, the DFDs should be regarded as a complement of UML, rather than a replacement.

Three conceptual combinations between UML diagrams and DFDs have been adopted from [45], as a starting point in defining a custom application specification methodology. The methodology starts with a functional decomposition of the system, followed by a data-flow perspective. When enough level of detail is reached, the system specification is transformed into either an object diagram or into a class diagram. The first approach seems to be more natural for component-based hardware systems, while the second one fits well in object-oriented software-intensive designs. We consider that the designer should follow the one that fits best with his affinities, experience and working culture. Both object-oriented models obtained from the DFD have also been used for developing prototypes in Java, in order to demonstrate their adequateness to describe the system. The prototypes have been built with the central idea of showing that the models do constitute a valid solution for the implementation of the system under consideration.

The methodology is specified in a model driven perspective, in which a number of models and model transformations are combined in a systematic process of system specification. The proposed transformations between UML and DFDs may look somewhat forced. This is due to the fact that one of the main goals in specifying these transformations has been to provide an approach easily automatable. The SMW tool has been used to provide support for the methodology, through graphical editors for both the UML and DFD models. Additionally, the scripting facilities of the tool have been used to automate several model transformations. These scripts have been implemented in Python, using high-level constructs similar to OCL. Taking advantage of the availability of both UML and also of the SA/RT metamodel implemented in the same tool (i.e., SMW), model transformations could be implemented not only between the models of the same metamodel, but also between models of different metamodels.

Future work may be directed towards the following topics. Firstly, applying the techniques proposed here to more complex examples would allow one to obtain more solid assessments on the usefulness of these techniques. Secondly, the transformation scripts between DFD and UML diagrams have been discussed mainly from the perspective of mapping elements of the DFDs onto elements of UML, but no focus has been put on how the behavior of the data-processes is refined into the behavior of the identified class methods. This aspect has to be investigated further. Thirdly, using DFDs for functional decomposition of data-driven applications seems to be more natural than using the complete 4SRS method. Analyzing in more detail if and how the 4SRS method can be enhanced with a data-flow perspective should be taken into account. Fourthly, the transformations from DFD to UML have been designed to be applied to flattened DFDs. An approach that takes into consideration the hierarchy of the DFD, during the transformation, has to be investigated. Finally, it is important to devise a more rigorous method for data classification inside DFDs, to bring out the benefits from object-oriented mechanisms like inheritance and polymorphism. For instance, we can easily see that the *routingDatagram* class in Figure 4.7 looks similar to a parent class of the *responseDatagram* and *requestDatagram*. For the moment, since we are addressing data-driven applications that target programmable architectures, we have intentionally avoided referring to specific object-oriented mechanisms such as inheritance and polymorphism. This is justified by the fact that they are in contrast with the static nature of the hardware components of the architecture.

Chapter 5

A UML Profile for the TACO Architecture

In this chapter[1], we discuss the use of the UML extensibility mechanisms for defining a DSL (i.e., a UML profile) for the TACO programmable architecture. The DSL is intended not only to benefit from the graphical UML notations, but also to support the design flow of the architecture by providing tool support and automation in existing UML tools. Beside modeling the hardware infrastructure of TACO, the DSL also provides a programming model of the architecture.

The chapter proceeds with an overview of the TACO protocol processing framework. Then, we present the TACO design flow from a model driven perspective, in which the models and model transformations supporting the flow are identified. After that we define a UML profile for the TACO architecture and we discuss how the latter is modeled in a UML tool by taking advantage of the profile. Next, we examine the mapping of application specifications onto the TACO architecture and we propose a systematic approach, in which the mapping process is performed.

5.1 The TACO Protocol Processing Framework

The protocol processing domain started to gain increasing attention in the last decade, stimulated by the technological advances in the telecommunication field. As a result, a new category of processing devices has been developed for supporting network communication. Network (aka protocol) processors are a special kind of programmable architectures tailored to deal efficiently with protocol processing tasks. Several solutions for protocol processing have been developed in recent years, among the most popular being Vitesse PRISM IQ2000, IBM PowerNP, Motorola C-Port DCP C-5, Xelerated Packet Devices, and Intel IXP 1200. Comprehensive overviews of both industrial and academic protocol processing ar-

[1]This chapter is based on publications [P.4] and [P.5].

chitectures have been given by several authors [57, 117], thus we omit discussing them here.

In this section, the TACO programmable architecture is discussed. TACO (**T**ools for **A**pplication-specific hardware/software **CO**-design) [132] is an integrated design framework for fast prototyping, simulation, estimation and synthesis of programmable protocol processors based on the *Transport Triggered Architecture* (TTA) concept [32]. A detailed description of the TACO framework may be found in Virtanen's Ph.D thesis [132], where the hardware-related aspects of the architecture are discussed in detail.

5.1.1 Hardware Considerations

A TACO processor (Figure 5.1) consists of a set of *functional units* (FUs) connected by an *interconnection network* (IN). The FUs may be of different types, each implementing its function(s) independently. There may be more than one FU of the same type in a TACO processor. In turn, the interconnection network is composed of one or many buses, which are controlled by an *interconnection network controller*.

An FU is basically composed of an interface to the interconnection network and an internal logic. The interface consists of several registers that are used for storing input and output values. Input registers are of two types: *operand* and *trigger*, respectively. *Operand registers* are used to store input values received from the buses. *Trigger registers* are a special kind of operand registers that, once written with data, trigger the computation of the FU. There may be zero or many operand registers, and exactly one trigger register in an FU. In addition, *output* registers are used for providing the result of the computation to the buses. Some FUs may also have a *result signal* that provides a boolean result of the computation to the interconnection network controller.

The FUs of TACO are connected to the buses via *sockets*. There are three types of sockets: *operand*, *trigger* and *result* sockets. *Operand sockets* connect one bus to an input register of an FU. *Trigger sockets* are a special kind of input sockets, which connect a given bus to the trigger register of an FU. There may be several trigger sockets connected to the same trigger register of an FU, each socket corresponding to a different FU operation. *Result sockets* connect FU result registers to the buses.

TACO processors may have several memory spaces, which may share the same physical memory block. A memory space is interfaced to the buses similar to any FU, and its data is accessed through the corresponding input/output registers.

Being a TTA-based architecture, TACO provides features like modularity, flexibility, scalability of performance and control of the processor cycle time, which are important concepts in the area of embedded systems design. The modularity of the architecture enables a good support in automating the design, each FU being designed separately from the other FUs and from the interconnection network.

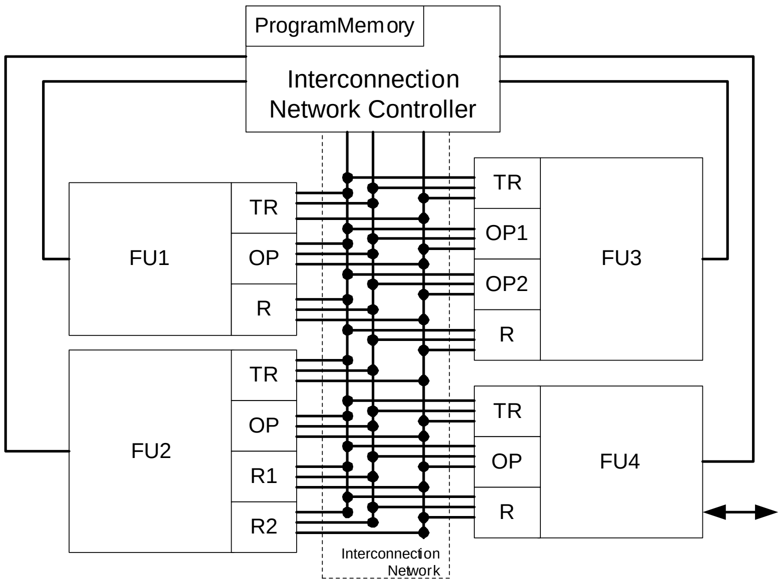


Figure 5.1: Generic architecture of the TACO processor

Each FU implements one or more pieces of functionality, and the final configuration is assembled by connecting different combinations of the functional units. The FUs are completely independent of each other and, at the same time, of the interconnection network, all of them being interfaced in a similar manner. The performance of the architecture can be scaled up by adding extra FUs, buses, or by increasing the capacity of the data transports and storage. The functionality of the architecture may be enhanced by adding new FU types to provide computational support for the application.

5.1.2 Software Considerations

In TACO, data transports are programmed and they trigger operations, in contrast to the traditional processors, where the operations of the processor trigger data transports. The architecture is programmed using only one type of instruction, the *move*, which specifies data transports over the buses. An operation of the processor occurs as a side-effect of the transports between functional units. Each transport has a *source*, a *destination* and *data* to carry from one FU register to another. The parallelism level in TACO can be exploited not only by increasing the number of FUs of the same type, but also by adding more buses. This allows the execution of several bus transports in the same processor cycle.

TACO has a variable-length instruction format based on the number of buses present in a given configuration. An *instruction* (Figure 5.2) is composed of several *subinstructions* (i.e., *moves*), which specify data transports on individual buses. In

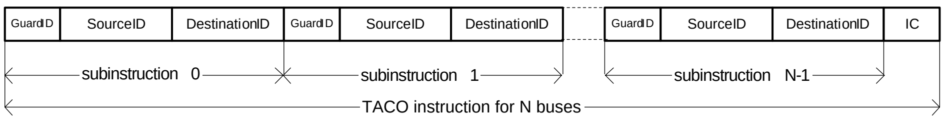


Figure 5.2: TACO instruction format

addition, an *IC* field is used to specify immediate integers on the buses. In turn, a *subinstruction* consists of three fields: *GuardID*, *SourceID* and *DestinationID*.

The *GuardID* field enables the conditional execution of the subinstruction. Upon evaluation of this field by the interconnection network controller, the subinstruction is either dispatched on its corresponding bus or ignored. GuardID values are configurable, based on combinations of result signal values received from FUs. For instance, 64 guard combinations can be defined, provided that a GuardID on 6 bits is used. The process of defining the GuardIDs for a given processor configuration is done at configuration-time based on the user experience and also on the application requirements. Combinations of the FU result signals may be obtained using the negation (*not*) and the conjunction (*and*) boolean operations. For instance, to implement the following hypothetical example:

```
if (x>2 or y<3 or z<>0) then
    do_something1();
else
    do_something_else();
```

on a TACO processor configuration with three comparator FUs, one could define a GuardID based on the result signals of each comparator FU. Let these result signals be *a* (true for x $>$ 2), *b* (true for y $<$ 3), and *c* (true for z $=$ 0), respectively. A guard that replaces the conditional statement can be written as *myGuardID=!a.!b.c*, where '!' stands for negation and '.' for conjunction. As such, a TACO-like implementation of the previous code would be as below:

```
compare(x>2);compare(y<3);compare(z=0);
!myGuardID:do_something1;
 myGuardID:do_something_else;
```

For brevity, the TACO subinstructions in the previous example have been replaced with a textual description of the operation performed (e.g., *compare*). We will discuss in the following sections, how the TACO operations are implemented, in practice, in terms of bus transports.

Finally, the *SourceID* and *DestinationID* fields are used to specify the source and target logical addresses (i.e., sockets) between which a transport takes place.

The interconnection network controller is the "brain" of the processor, being in charge of implementing data transports on the buses. The controller uses a *program memory* from which it fetches instructions, splits them into subinstructions and dispatches each subinstruction on the corresponding bus. It is important to mention that one does not have to change the instruction format when adding FUs, as long as the existing FUs are addressable by the length of source and destination addresses.

A *program counter* (PC) is maintained by the interconnection network controller. A built-in trigger socket is used to load the PC with a desired value (either absolute or relative), making possible to implement program jumps. More details on the actual implementation of the controller may be found elsewhere [132, p. 64–67].

Visibility of data transports at the architectural level is an important feature of TACO, allowing compilers to optimize and schedule these transports. Since all the operations of the processor are side-effects of the data transports on the buses, the processor cycle-time depends on the availability of the FU results.

5.1.3 Programming for TACO

From the programmer's point of view, programming TACO processors is a matter of moving data from output to input registers, identified using the address spaces of the corresponding sockets. An example of two instructions, each composed of three subinstructions, is given below:

```
g1:src1 > dst1; src2 > dst2;  +02 > dst3;
   src4 > dst4; src5 > dst5; src6 > dst6;
```

In this example, the first subinstruction is executed only if the guard 'g1' is evaluated to *true*, while the others are executed unconditionally. Operands *src1* through *src6* and *dst1* through *dst6* depict input and respectively output sockets of different FUs. The operand '+02' represents an *immediate value* (integer) that is written to the *dst3* socket.

Using registers for interfacing FUs allows one to apply TTA-specific optimization techniques [32], like moving operands from an output register to an input register without additional temporary storage (bypassing), using the same output register or general purpose register for multiple data transports (operand sharing), or removing of registers that are no longer in use, etc. All these techniques help in reducing code size and consequently, in reducing the number of bus transports. Some general compiler optimizations may also be performed on the TACO assembler code, like sinking, loop unrolling, etc. A compiler performs the necessary allocation and scheduling, along with transforming the assembler code into hexadecimal code, which is uploaded in the program memory of the processor.

5.1.4 The TACO Component Library

To provide prerequisites for automation and reuse, the TACO processor resources are organized in a library of components, namely the *TACO Component Library*. Each resource included in this library is specified from three perspectives. A *simulation model* provides SystemC executable specifications, enabling the simulation of the processor configurations, in order not only to check their functional correctness, but also to evaluate their performance. A *synthesis model* provides implementations of each processor resource in VHDL. The synthesis model of TACO targets synthesizable off-the-shelf ASIC components, thus enabling the designer to generate synthesizable processor configurations, at system-level. The simulation model is developed in concordance with the performance characteristics provided by the synthesis model, such that the simulation of the system provides cycle accurate results, with respect to the synthesis model. An *estimation model* allows one to obtain high-level estimations of different physical characteristics like *area*, *power consumption* and *gate delay* of the components. The estimates are based on a mathematical model built in Matlab. The estimation model for TACO has been designed and developed by M.Sc Tero Nurmi [89, 133].

Building the TACO Component library is an iterative process that relies both on the analysis of the processing needs of different protocols and also on the TACO resources identified in previous applications. The SystemC and VHDL modules corresponding to each TACO resource are designed independently. Each new FU is simulated, synthesized and validated before being added to the library. TACO configurations are created using top-level files, which specify the required modules and their interconnection, as we will discuss in the following sections. Based on the results obtained from the simulation and estimation models, optimizations may be suggested for improving the provided performance, with respect to a given application family.

Several libraries may be defined, to serve for different application domains. In [10], a TACO library for SDR (Software Defined Radio) wireless communications is defined. In this thesis, we use a library for IPv6 routing.

5.2 The TACO Design Flow

This section introduces a model driven approach for the TACO design flow (see Figure 5.3). Several models represent different perspectives of the system, at various abstraction levels. Transitions between models are specified and supported through model transformations.

Being a programmable architecture, the TACO processor is configured at design-time to support the application specification with respect to both the performance requirements and the physical constraints. Concretely, starting from a given application specification, one has not only to identify the components of the processor needed to support the application, but also to multiply these components in an optimal manner to provide the required throughput. Several steps are applied iteratively (Figure 5.3- (a)) in order to obtain the “final” processor configuration.

From the application mapping onto the TACO architecture, we identify the functionality that the TACO architecture has to provide for supporting the application. At the *Qualitative Configuration* step, the TACO processor is configured

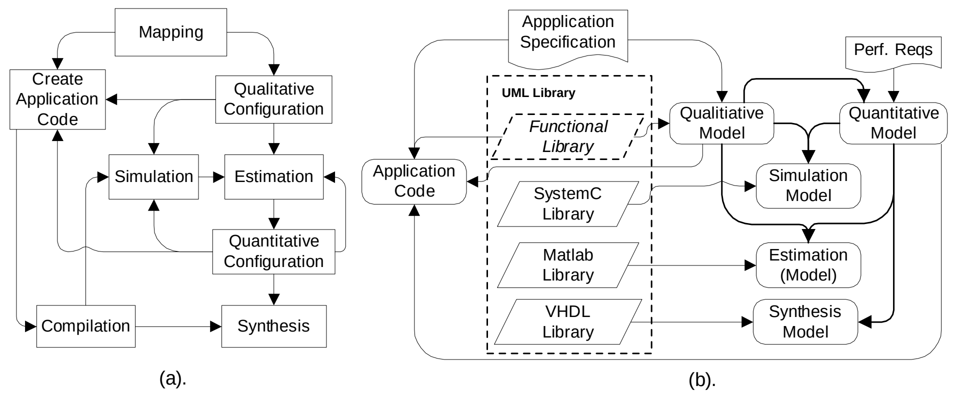


Figure 5.3: Design flow for TACO processors: (a) activity-based view and (b) model driven view

to support the application functional requirements. Only one FU of each type and one bus are included in the *qualitative configuration* of TACO. Consequently, the application program code is constructed, starting from the identified resources.

The qualitative configuration is then simulated and estimated. From the *Simulation* process, beside verifying the correct functionality of the application, one can gather application performance information, like number of cycles required to execute the application, bus utilization, and register pressure. Out of the *Estimation* process, we get estimates of the physical characteristics.

Various architectural configurations are explored, at the *Quantitative configuration* step, to tune the architecture towards maximum throughput, with respect to the physical constraints (i.e., area and power consumption). This is done by varying the number of FUs of each type, and the number of buses. The resulting configurations are again simulated, this time with respect to the application performance. Moreover, the estimates of their physical characteristics are collected. In the end, one or more *quantitative configurations*, which are able to provide the required application performance within given power and area constraints, are selected as input for the *Synthesis* process.

Figure 5.3-(b) presents the design flow discussed above, form a model driven perspective, where several models and model transformations are defined. The transformations are supported by the three libraries of the TACO framework. As we have already mentioned, each library models the same TACO component, from three different perspectives: simulation, estimation, and synthesis.

Beside these three libraries, we propose a new library to support the design process, namely the *Functional Library*. This library encodes the correspondence between the functionality and the hardware implementation of each TACO resource. We will discuss this approach in more detail, in the following sections.

Furthermore, in order to provide a “unified” component library, in which all the library information is available in a single model, we suggest the adoption of a

UML Library model. This library will not only conjoin, but will also abstract the information of the four different libraries, in order to facilitate the generation of the various TACO models.

5.3 A UML Profile for TACO

In this section, we propose a UML-based DSL for the TACO protocol processing framework. The purpose of the TACO DSL is two-fold: to define a graphical modeling language of the TACO architecture, and to experiment the advantage of using UML in providing rapid tool support in existing UML tools. The TACO DSL is specified by extending the UML language via its profile mechanism, which has been introduced in Chapter 2.

5.3.1 Mapping TACO Resources to UML elements

Several researchers have focused their efforts on classifying UML stereotypes [19, 41, 112], while others pointed out guidelines for creating and using stereotypes [12, 13, 19, 54, 78, 122]. In our opinion, this shows that there is not yet a common (*universal*) approach to follow, when defining new UML profiles.

Nevertheless, we consider that two aspects have to be taken into consideration when defining a profile. On the one hand, one has to select the appropriate UML elements to represent the concepts of the application domain, without altering the semantic definition of those elements. On the other hand, the set of elements and their abstract syntax must provide enough expressiveness for the user of the new model, such that one may easily apply the profile for modeling domain-specific problems.

As a starting point in defining a UML profile for TACO, we focus on its physical (hardware) structure. For this purpose, the UML class diagram, reflecting a structural perspective of a system, is a good candidate. This also allows one to create instances (i.e., configurations) of the TACO architecture, using object diagrams. In addition, we need to decide which elements of the class diagram can be extended to naturally represent the TACO concepts. We consider that a similar approach is not necessary in case of the object diagram, since object instances may be interpreted, with respect to the TACO architecture, based on their classifier.

From an ideological point of view, all the concepts of the TACO architecture (including buses and sockets) may be modeled as UML classes. However, the approach may seem unnatural, especially for a hardware designer, and the resulting models could become cluttered with too many graphical elements. Instead, we decided to classify the TACO artifacts based on their role in the architecture: *components*, *connections*, and *component properties*. This classification facilitates the mapping of the TACO concepts onto the elements of the class diagram, as follows: components are mapped onto classes, connections onto associations, and properties of the components onto class properties (i.e., attributes).

Components. Three stereotypes, extending the UML *Class* element, have been defined to model the TACO components: functional unit ($\ll$*FU*$\gg$), network controller ($\ll$*NetCtrl*$\gg$) and interconnection network ($\ll$*Network*$\gg$).

Connections. Three stereotypes, $\ll$*Socket*$\gg$, $\ll$*ResultBit*$\gg$, and $\ll$*NetLink*$\gg$, have been defined to model the connections of the TACO architecture. All these elements extend the UML *Association* element.

A $\ll$*Socket*$\gg$ models the connection between an $\ll$*FU*$\gg$ and a ($\ll$*Network*$\gg$) element, respectively. In practice, several sockets are used to connect a given FU to the TACO buses. However, we have taken two design decisions to simplify our design: to model all the sockets connecting the TACO FUs to buses as one single $\ll$*Socket*$\gg$ element, and furthermore, to enable for the $\ll$*Socket*$\gg$ to connect an FU to the interconnection network, rather than to each bus. These two decisions have been taken based on the following rationale: a) the number of sockets connecting the registers of an FU to the buses can be rather large, and thus, it may clutter the design; b) in our work, we use fully connected buses, that is, each socket is connected to all the buses in a configuration.

Finally, a $\ll$*ResultBit*$\gg$ stereotype models the result signal between an FU and the interconnection network controller, while a $\ll$*NetLink*$\gg$ stereotype is used for explicitly connecting the $\ll$*Network*$\gg$ and the $\ll$*NetCtrl*$\gg$ elements.

Component properties. We remind the reader that each FU has one trigger register and zero or many operand and result registers. Since FU registers may be seen as a structural property of each FU, we decide to model the TACO FU registers as attributes of the $\ll$*FU*$\gg$ class. Consequently, three stereotypes extending the UML *Attribute* element have been defined to model each register type: $\ll$*TR*$\gg$, $\ll$*OR*$\gg$, and $\ll$*RR*$\gg$.

The main advantage of having the connection between FUs and buses modeled as a single $\ll$*Socket*$\gg$ element provides the advantage of less cluttered models. However, the information abstracted away (e.g., number of sockets, their type and direction) has to be modeled somewhere else, in order to provide a sufficient level of detail. Although used as external connections to FUs, the sockets and their types (i.e., trigger, operand, and result) are tightly dependent on the implementation of a given FU. Hence, sockets may also be considered as internal properties of the FUs. Three new extensions of the *Attribute* element could have been created to model the logical sockets: $\ll$*TS*$\gg$, $\ll$*OS*$\gg$, and $\ll$*RS*$\gg$, respectively.

We have noticed that having too many attributes for a class complicates the model excessively. In the average case, the functional units of the TACO processor can have around four to five registers, and six to eight sockets connected to them. Having a total of ten-to-thirteen attributes (taking into account both registers and sockets) in a class would not provide an efficient abstraction of the FU's characteristics. However, we have observed that three rules can be postulated regarding the TACO architecture:

- there is one trigger register in a given FU;
- there is at least one trigger socket assigned to a trigger register;
- each operand/result register, if present, is connected to exactly one socket.

Applying these rules, we conclude that one can “subtype” the sockets based on the register type that they are connected to. For instance, an FU with the configuration as given below, would feature two trigger sockets (i.e., *socket1* and *socket2*), connected to exactly one trigger register, and two operand sockets (i.e., *socket2* and *socket3*) connected to two different operand registers:

```
<<TR>>:socket1
<<TR>>:socket2
<<OR>>:socket3
<<OR>>:socket4
```

The approach not only enables us to reduce the number of the ≪*FU*≫ class attributes, but also encode the functional[2] and the physical characteristics of an FU, in an integrated manner.

An ≪*RB*≫ stereotype has been created by extending the *Attribute* element, to encode, in the FU definition, the fact that an FU may have a result signal connected to the network controller. The only meaning of an ≪*RB*≫ element is to specify whether a given FU has such an interface to the external environment.

We intentionally left for the final discussion the modeling of the TACO bus. Although the bus element may be conceptually seen as a feature (e.g., an attribute) of the interconnection network, in our opinion it is more natural to model it as a component of the ≪*Network*≫. Such an approach allows for several instances of the ≪Bus≫ element to be contained in the same instance of the interconnection network, at instantiation time. Following the principles of object-orientation, this approach would be modeled using the *composition* relationship. Unfortunately, such a relationship between UML stereotypes is not allowed by the UML metamodel. In UML 1.4., only *generalization* relationships may be used between stereotype definitions. However, such relationships between stereotypes may be defined using OCL constructs as we will discuss later on in this chapter. As such, the ≪Bus≫ stereotype is defined to extend the *Class* element, while for specifying its containment in the ≪*Network*≫ element, we will rely on the well-formedness rules associated to the profile, as it will be discussed in the next section.

An example of a TACO processor configuration modeled using the TACO Profile is presented in Figure 5.4. The presented configuration is composed of a network controller (*Controller*), an interconnection network (*Interconnection-Network*), and an FU (*Foo*).

[2]sockets can be seen as functional features, as their combination provides the FU’s functionality

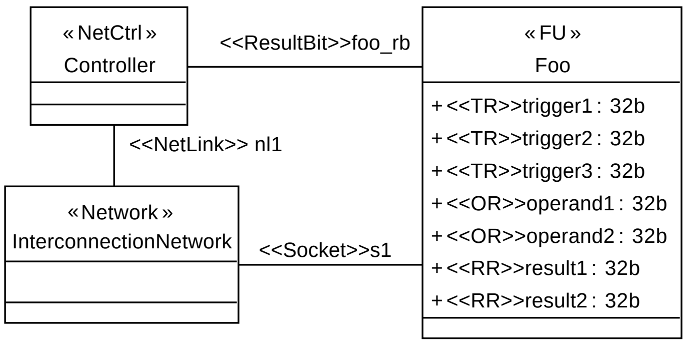


Figure 5.4: Generic TACO processor configuration using the UML notation

5.3.2 A Programming Model for the TACO Architecture

As we have discussed in the introduction of this thesis, researchers have pointed out the necessity of defining and employing a programming model for programmable architectures. Such a programming model abstracts the hardware details of the architecture, while providing enough detail to allow one to tune and program the architecture for specific tasks.

Therefore, we have defined a programming model for TACO for the following purposes: a) to provide an abstraction of the hardware architecture enabling the designer to focus on the functionality of the architecture, rather than on its physical details; b) to bridge the gap between the hardware architecture and the application specification during the mapping process. The TACO programming model is composed of several programming elements (i.e., *programming primitives*), split into two categories: functional primitives and control primitives.

Functional primitives. The *functional primitives* (FPs) are those programming constructs providing computational support for a given application. Their main characteristic is that their presence in a TACO configuration varies with the types of TACO resources (FUs) included in a configuration. We recall that the functionality of the TACO processors resides in their FUs. Each FU provides one or more processing functions, whereas buses are only used to support data transports between FUs. Each processing function is associated with a specific trigger socket. When data is written through this socket into the trigger register of an FU, the function is executed. In TACO terms, such a function is called *operation* or functional primitive. For executing an FP, several bus transports may be required. In the first stage, the operand registers of the FU are set up, whereas in the second stage, the trigger register is written and, consequently, the operation is executed. This implies that an operation is equivalent with a number of TACO bus transports and, even more, every time the operation is invoked, the exact same sequence of transports is used. Therefore, we can abstract the operations of the processor as macros containing TACO bus transports. The benefit from this approach is that every time an FP is used, it can be automatically translated into bus transports.

Control primitives are used to support the sequencing of functional primitives in a given application, and they are typically present in all TACO configurations. Three types of control primitives are defined in TACO:

- *unconditional jumps* are used to specify jumps at a specific address of the program code, being implemented by the interconnection network controller;
- *conditional jumps* implement jumps to a specified address of the program code, in case the GuardID of a given subinstruction is satisfied;
- *labels* provide references to a given address (line) of the program code.

Using these types of operations more complex programming structures like loops, cases, subroutines, etc., can be defined.

The set of functional and control primitives provided by a given processor configuration form the *application programming interface* (API) of that configuration. Adding or removing FUs modifies accordingly the API of the configuration. Furthermore, by reflecting the functionality of a given configuration in terms of processing capabilities, the programming interface allows one to deduce the right configuration to support the application. This correspondence is encoded and stored in the *Functional Library* (introduced in Figure 5.3). By using such a library, the API provided by different FUs can be reused not only for programming a given configuration, but also for facilitating the mapping process, as it will be discussed in Section 5.5.

It is worth mentioning that the TACO FPs could be defined at several levels of granularity. At the lowest level, each FP is provided by a single FU. At a higher level, the FPs can be more complex, each of them being implemented by a combination of FUs. The former approach provides a better mapping of the application specification onto the architecture, whereas the latter is preferred when, for applications in the same family, complex pieces of functionality of the architecture can be identified, optimized, and reused.

In order to integrate the functional primitives within the TACO Profile, we decided to use the *Operation* behavioral property of a UML class. “An operation is a service that can be requested from an object to affect behavior” [94, p. 2-45]. This definition recommends the *Class Operation* as a good candidate for modeling the TACO FPs. A UML *operation* is basically characterized by a *name*, a number of *parameters* and a *specification*. After mapping these elements on the TACO FPs, we obtain the following:

- the *name* property will depict the processor functionality (i.e., the FP) modeled by the *UML Operation*;
- the *parameters* will provide the list of input/output values that an FU receives as input and provides as output when the FP is invoked; these parameters map one-to-one on the sockets used by each FP;

- the *specification* will describe the FP in a textual language. Based on these facts, a new stereotype, namely ≪*FUop*≫ has been defined to extend the *UML Operation* element.

The UML standard defines the *Operation* as an abstract element that specifies a service provided by a given class, only. The corresponding implementation of that service is modeled in UML using the *Method* behavioral feature. The UML specification enforces that, a given method has the same signature as the operation it implements, and furthermore, that both are placed in the same class. Several implementations (i.e., methods) of the same class operations may be specified. The approach enables one to define more then one implementation for each TACO FP. For instance, we may have an FP implemented in terms of bus transports, but we could also define a "platform independent" implementation of the same FP (e.g., using the C programming language) to provide an executable specification of that FP.

In our approach, we use a corresponding suffix added to the method name, to denote the implementation language that it targets. This lets us differentiate between different implementations of the same operation. An example is presented below, where *addition_taco* and *addition_c* methods provide a TACO and, respectively, a C implementation of the ≪*FUop*≫ *addition* operation.

```
<<FUop>> addition(a:int, b:int, c:int);

addition_taco(a:int, b:int, c:int){
    a > TSC; //setup COUNTER with the value of 'a'
    b > TIC; //increment COUNTER with the value of 'b'
    RC > c;  //read the result into 'c'
    }
addition_c(a:int, b:int, c:int){
    c = a+b;
    }
```

FP specifications like the ones above will be defined and integrated with the UML models of the TACO architecture. Their purpose is to provide a functional perspective of a given TACO configuration and to facilitate the application mapping process, as it will be discussed later on in this chapter.
Finally, all the elements of the TACO Profile are grouped under the *TACOProfile* package, bearing the ≪*profile*≫ stereotype, as defined by the UML metamodel. Branding a package with the ≪*TACOProfile*≫ stereotype will depict a model containing elements specified by the TACO Profile definition.

Table 5.1 presents the list of the stereotypes defined by the TACO Profile and the corresponding UML elements that have been extended by each stereotype.

5.3.3 Encoding Additional Information in the Profile

Stereotypes provide a classification of the UML elements, based on their intended meaning with respect to an application domain. Sometimes, additional properties

Name	**Base Class**	**Description**
TACOProfile	Package	Model containing elements of the TACO Profile
Network	Class	The IN of a TACO processor
NetCtrl	Class	The interconnection network controller of TACO
FU	Class	Functional unit
Bus	Class	Component of the IN
Socket	Association	Generic socket connection between an FU and the controller
Result_Bit	Association	Result signal of an FU connected to the controller
NetLink	Association	Element modeling the connection between the controller and the network
TR	Attribute	Trigger register of an FU
OR	Attribute	Operand register of an FU
RR	Attribute	Result register of an FU
RB	Attribute	Feature of the FU depicting the presence of a *result_bit* signal
FUop	Operation	Piece of functionality provided by a FU (i.e., functional primitive)

Table 5.1: The stereotypes of the TACO Profile

are required to be added to these "classifications", such that each element belonging to a given "class" inherits the properties automatically.

TaggedValues are suggested by the UML specification to define additional properties of the stereotyped elements. A tagged value is "an arbitrary property attached to the model element based on an associated tag definition" [94, p. 2-78]. Therefore, each tagged value has a type given by a *TagDefinition*, which is declared as a property of a given stereotype.

We recall that the TACO framework contains a Matlab estimation model providing estimates of area, power consumption and gate delay for the TACO components. Having such estimates available at design-time, enables the TACO designer to make educated choices in configuring TACO, in order to comply with the performance requirements and physical constraints of the application. We have included these estimates in the TACO Profile definition as additional properties of its elements. We will briefly discuss each estimate type in the following.

In TACO, the estimation model provides worst-case estimates for the delay introduced by the gate levels of different components like FUs, sockets and buses. Of these, only the delay for FUs can be statically estimated, before configuration-time, based on the implementation characteristics of each FU. The gate delay estimates for the other TACO resources are calculated mathematically, at configuration-time, based on the distance between FUs, bus lengths, and internal execution delay for each FU (see [132, p.126]). Consequently, we have defined a tagged value, namely *gd*, to store the gate delay of each FU. A corresponding *gd_estimation* tag definition has been added to the ≪*FU*≫ stereotype.

The Matlab model also provides estimates of the power consumption for each FU and the interconnection network controller, which are the main energy consumers in a TACO processor. In addition, estimates for the area occupied by the FUs and the interconnection network controller are provided. Following an approach similar to the case of gate delay, corresponding *area_es* and *pc_es* tag definitions have been declared for the stereotypes that model the above mentioned elements, and two new tagged values *area*, and *pc*, are used as properties of the ≪*FU*≫ and ≪*NetCtrl*≫ classes, respectively. Due to the numerical nature of all three estimates, their tag definitions have been declared of type *UML14::DataTypes::Integer*.

Next, we have defined pointers to the simulation and synthesis models of each TACO component (including buses), in order to integrate the information stored in the SystemC and VHDL libraries, within the TACO Profile. This lets one select, at code-generation stage, the corresponding implementation files for each component of a given configuration. Consequently, two new tagged values are used: *sysc* and *vdhl*, respectively. A corresponding *implementation* tag definition of type *UML14::DataTypes::String* has been added to their stereotype definitions.

Figure 5.5 presents an example where tagged values are used to add estimation and implementation properties to the *MyFU* functional unit.

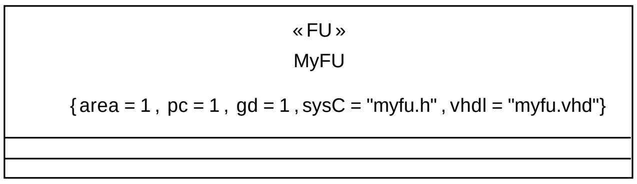

«FU»
MyFU
{area = 1 , pc = 1 , gd = 1 , sysC = "myfu.h" , vhdl = "myfu.vhd"}

Figure 5.5: Tagged values for specifying additional properties of TACO FUs

5.3.4 Enforcing Architectural Consistency through Well-formedness Rules

UML stereotype definitions provide a list of customized UML elements intended for representing the components of the TACO architecture. These definitions do not enforce the architectural restrictions of the architecture, except for the ones provided by the abstract syntax (i.e., relationships between elements) of UML. By selecting certain UML elements to extend their abstract syntax, the associated constraints behind these elements are automatically inherited. For instance, since the UML metamodel prevents two *Association* elements from being connected together, it means that connecting a ≪*Socket*≫ and a ≪*Result_Bit*≫ elements is not allowed either.

To enforce additional architectural restrictions of the TACO architecture, a number of well-formedness rules have to be defined as constraints associated to the TACO Profile. Constraints are used in UML to introduce new semantics to UML models, beyond the graphical capabilities of UML. They are attached to the UML elements and expressed either in a textual language or using a semi-formal language like OCL, the default constraint language of UML. The well-formedness rules of the UML metamodel can be found in [94]. These rules are attached to different elements of the metamodel.

Element	**Well-formedness rule**
Socket	A single *Socket* connects an *FU* to a *Network* element
Result_Bit	*Result_Bit* connects an *FU* to *NetCtrl*
FU	If a *RB* attribute is present in a *FU* then a *ResultSignal* is connected to the *FU* class An FU is connected exactly by one *Socket* to a *Network* There is exactly one *RB* in a *FU* There is at least a trigger socket in an *FU* There is the same number of *TR* sockets and operations in an *FU*
Network	There is only one *NetCtrl* in a TACO model An instance of *Network* class contains at least one instance of type *Bus*
NetCtrl	There is only one *NetCtrl* in a TACO model A *Network* has at least one *Bus*
NetLink	A *NetLink* connects a *NetCtrl* to a *Network*

Table 5.2: Architectural well-formedness rules of the TACO Profile

When creating UML profiles, the stereotypes are the ones that define new "types" of elements. Therefore, it is to the newly created stereotypes that architectural constraints of TACO have to be attached, such that their instances inherit these constraints. In case of TACO Profile, the well-formedness rules may be cosidered as falling into two categories: a) enforcing the architectural constraints of the TACO hardware architecture (Table 5.2); b) enforcing the consistency of additional properties (Table 5.3). While rules specified in the first category are mandatory, the ones of the second category may be regarded as optional.

Element	**Well-formedness rule**
FU	All tagged values have assigned values An *FU* contains the five tagged values associated to them by the profile definition Only TACO defined tagged values are allowed in a TACO Profile
FUop	An operation has at least an implementation method

Table 5.3: Additional well-formedness rules of the TACO Profile

Other well-formedness rules can be envisioned to assist the designer in his work. For instance, verifying the conformance of a tagged value to its tag definition type, or that the numerical tagged values are in a predefined range, can be considered as such rules. In addition, since the TACO Profile restricts the UML elements to a given set of elements, constraints to forbid the usage of other elements of the class diagram may also be introduced. For instance, in the current TACO Profile definition, class relationships like inheritance or composition should not be allowed.

Although UML uses and suggests the use of OCL as an executable language for specifying and enforcing well-formedness rules, the implementation and verification of such rules depends on capabilities of the UML tool used. Normally, constraints associated with profiles should be automatically enforced by the UML tools. Unfortunately, most of the UML tools at the time of performing this research provide poor support for implementing UML profiles and for verifying their properties [27]. Our approach of integrating and verifying the TACO well-formedness rules within a tooling environment will be discussed in the following section.

5.4 Modeling the TACO Architecture in UML

This section discusses the modeling process of the TACO architecture using the TACO Profile. Tool support and automation, assumed by the profile implementation in a UML tool, is presented for various steps of the TACO design flow.

For convenience, we selected the SMW tool introduced in the previous chapter, and soon after, its newer version, the Coral modeling framework [102], for implementing the TACO Profile. The choice is motivated by the discontinued support for SMW, as well as by the improved capabilities of Coral. As similar to SMW, Coral provides a scripting environment using the Python programming language. This allows one not only to query and transform models using scripts, but also to enforce their consistency using OCL-like constructs. Coral supports the definition of new stereotypes, tagged values and constraints for the UML 1.4 profiles. In addition, editors are provided for most of the UML diagrams.

5.4.1 Creating Processor Configurations

Following the traditional object-oriented design flow, and the TACO Profile definition discussed in the previous section, two UML diagrams are used for modeling TACO. The TACO class diagram (Figure 5.6) provides a template of the TACO architecture for modeling its components and their interconnections.

Configurations of the TACO architecture may be obtained using object diagrams. The object diagram has only two types of elements: *objects* and *links*. Objects represent instantiations of classes, whereas links are instances of class associations. The TACO object diagram is used to model both the qualitative and the quantitative configurations of the processor. Figure 5.7 presents a configuration of TACO derived from the class diagram in Figure 5.6. One may notice that two instances of the Shifter FU (i.e., *S1* and *S2*) are present in this configuration.

Each TACO object diagram also depicts the buses included in the interconnection network of the specified configuration. We recall that an instance of a ≪*Network*≫ class may contain one or more instances of a ≪*Bus*≫ class, as defined by the TACO Profile. Currently, the Coral editor does not display the contents of the *ownedInstances* slot of an *Instance* element, yet an interconnection network (i.e., *net*) consisting of three buses would look as in the example of Figure 5.8.

Several configurations of the TACO architecture may be created, using object diagrams. It is important to notice that the level of detail provided by the object diagram is low, focusing on the static relationship between different class instances. Nevertheless, the properties of each instance can be easily obtained by interrogating the properties of its classifier. The approach allows one to rapidly create TACO configurations in UML tools.

Architectural constraints are enforced at object diagram-level, only based on the underlying metamodel of the UML object diagram, although additional well-formedness rules may also be specified. For instance, a constraint could be added

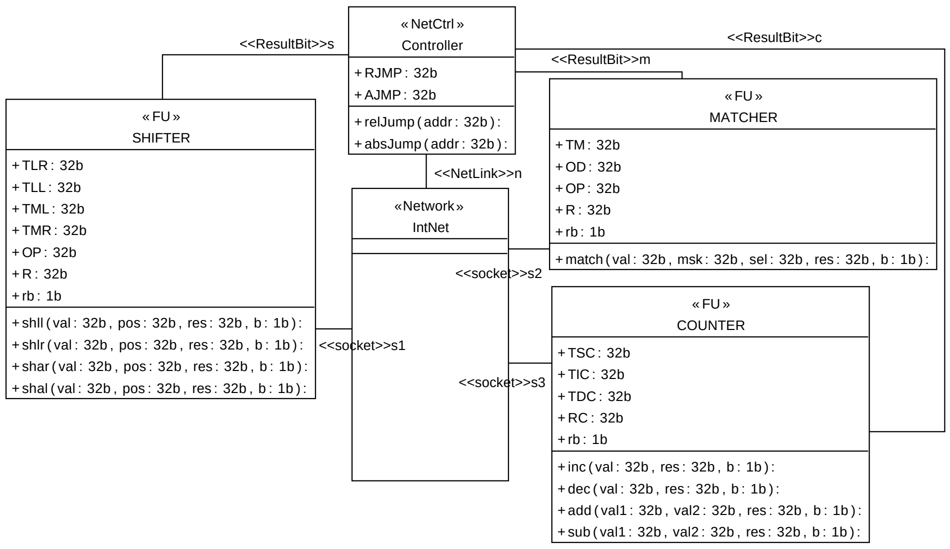

«NetCtrl»
Controller
+RJMP: 32b
+AJMP: 32b
+relJump(addr: 32b):
+absJump(addr: 32b):
<<ResultBit>>s
<<ResultBit>>c
<<ResultBit>>m
«FU»
SHIFTER
+TLR: 32b
+TLL: 32b
+TML: 32b
+TMR: 32b
+OP: 32b
+R: 32b
+rb: 1b
+shll(val: 32b, pos: 32b, res: 32b, b: 1b):
+shlr(val: 32b, pos: 32b, res: 32b, b: 1b):
+shar(val: 32b, pos: 32b, res: 32b, b: 1b):
+shal(val: 32b, pos: 32b, res: 32b, b: 1b):
<<NetLink>>n
«Network»
IntNet
<<socket>>s1
<<socket>>s2
<<socket>>s3
«FU»
MATCHER
+TM: 32b
+OD: 32b
+OP: 32b
+R: 32b
+rb: 1b
+match(val: 32b, msk: 32b, sel: 32b, res: 32b, b: 1b):
«FU»
COUNTER
+TSC: 32b
+TIC: 32b
+TDC: 32b
+RC: 32b
+rb: 1b
+inc(val: 32b, res: 32b, b: 1b):
+dec(val: 32b, res: 32b, b: 1b):
+add(val1: 32b, val2: 32b, res: 32b, b: 1b):
+sub(val1: 32b, val2: 32b, res: 32b, b: 1b):

Figure 5.6: Processor configuration (class diagram) using the TACO Profile

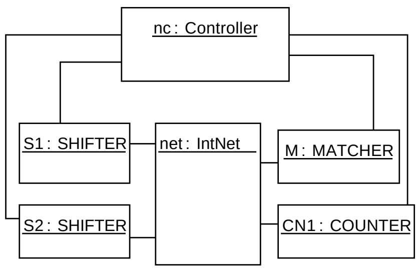


Figure 5.7: Object diagram representing a quantitative configuration of TACO

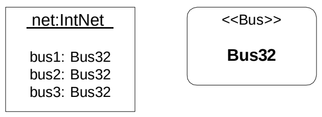


Figure 5.8: Adding buses to a TACO configuration

to the some of the stereotype definitions to impose that only one instance of a given class (e.g., ≪*NetCtrl*≫) is allowed in a configuration.

5.4.2 A UML-based Approach for the TACO Library

The TACO Profile definition and its implementation in Coral allows the designer to manually create configurations of the TACO processor from scratch. However, the approach can be time consuming and error prone. Following a library-based approach alleviates such problems by enabling support not only for reuse, but also for rapid creation of new processor configurations.

As already mentioned in Section 5.2, we have defined a TACO UML Library that encompasses information provided by the functional, simulation, estimation and synthesis libraries of the TACO framework, unified in a single model. Therefore, four types of information are included in the library:

- *structural* - internal structure of components (e.g., registers, result signals, etc.);
- *physical characteristics* - estimates of area, power use and gate delay;
- *simulation* and *synthesis* specifications - pointers to the SystemC and VHDL implementations;
- *functional* - functional primitives (i.e., the API) provided by each component, and their implementations in terms of TACO bus transports, as well as additional implementations (e.g., using the C language);

The TACO UML Library is modeled using the TACO Profile definition, in order to benefit from support in UML tools. The approach allows one to graphically create and maintain such a library, and moreover, to store it as a UML model. Consequently, a new stereotype ≪*TACOUMLLibrary*≫ has been added to the TACO Profile definition by extending the UML *Package* element. The purpose of this stereotype is to provide capability for differentiating a TACO design from a TACO library model. The TACO library encompasses all the components of the TACO architecture like FUs, network controllers, interconnection networks and buses, along with their related properties. In fact, the library model is similar to a TACO class model, except that no association-related stereotypes (e.g., *Socket*, *Result_Bit*) are included.

New TACO class diagrams can be created either from scratch, or by using the TACO UML library. In the latter case, the designer has to select the desired components from the library model and add them to the new model. The addition of new elements from the library can be done either manually or automatically. To support the former, Coral provides *copy/paste* operations of model elements from one model to another, assuming that they conform to the same underlying metamodel [103]. This feature assists the designer in creating new TACO models (i.e., class diagrams) by simply copying existing elements from the TACO library. To support an automated design, a dedicated script (similar to the Python scripts discussed in Chapter 4) may be implemented, for copying library elements from one model to another.

Beside graphically creating and editing TACO models, other tool support could be provided for assisting the design activity. As mentioned previously, only TACO components (i.e., stereotyped classes) are stored and reused from the library. Once new components are added to a new model, they also have to be manually interconnected by the designer. Nevertheless, we have observed that only three types of elements (i.e., *Socket*, *Result_Bit* and *NetLink*) have to be added to make the model complete, and furthermore, their addition can be automated. We have implemented a Python script to assist the process, based on the following steps:

- for each ≪*FU*≫ element, a ≪*Socket*≫ element is created and connected to the ≪*NetCtrl*≫ element;
- all ≪*FU*≫ elements, containing an ≪*RB*≫ attribute (depicting the necessity of a result signal), are connected to the network controller through a ≪*Result_Bit*≫ signal;
- a ≪*NetLink*≫ element is added between the interconnection network controller and the interconnection network.

Figure 5.9 provides a snapshot of the TACO UML Library for IPv6 routing. At the bottom of the screen, a property editor allows one to edit the properties of TACO elements. In this particular example, a property editor for editing the methods of the MATCHER FU is shown at the bottom of the screen. The left hand side pane, provides a list of library elements, while the main editor lets one to graphically create and edit the library components.

The TACO Library is currently built manually by the TACO hardware designer and complemented with estimation information extracted from the libraries of the TACO framework. The process of building the library might seem tedious, but the number of library elements is relatively small (e.g., fifteen for IPv6 routing) and new additions may occur rather seldom. Nevertheless, once we have the TACO library built, we are able to quickly create processor configurations from which can further generate different artifacts of the development process.

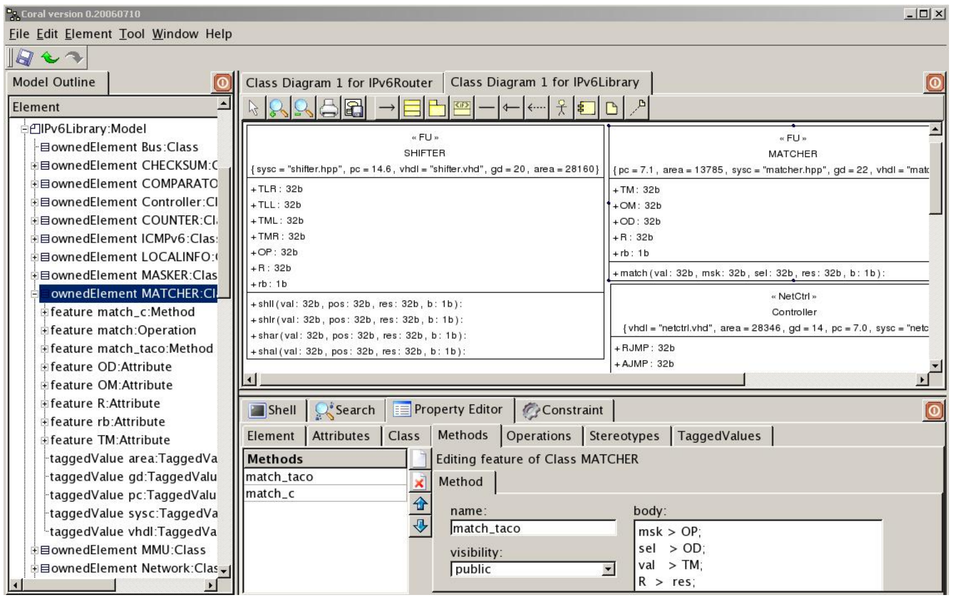
5.4.3 Tool Support for the Verification of Well-formedness Rules

We have used OCL-like constructs specified in Python, in order to specify the well-formedness rules associated to the TACO Profile. An example of such a constraint and its Python implementation are shown below:

```
def wfrFUTrigger(self):
 "There is at least one trigger socket in a FU""
 return (self.feature.select(lambda att:
   att.oclIsKindOf(UML14.Attribute)).stereotype.exists(lambda st:
      st.name=="TR") > 0)
```

One of the missing features of Coral, at the time when the TACO Profile was implemented, was the automatic verification of the constraints associated to a profile. To tackle this problem, we have specified the constraints of the TACO Profile in a separate file, which is executed in the scripting environment of the tool each time one wants to check the consistency of the models created. An excerpt of the code that is used to verify the constraints is given below:

```
 for el in self.ownedElement:
    if hasStereotype(el,"FU"):
        print " verifying element " + el.name
```



```
IPv6Library:Model
  ownedElement Bus:Class
  ownedElement CHECKSUM:C
  ownedElement COMPARATO
  ownedElement Controller:Cl
  ownedElement COUNTER:Cl
  ownedElement ICMPv6:Clas
  ownedElement LOCALINFO:
  ownedElement MASKER:Clas
  ownedElement MATCHER:Cl
    feature match_c:Method
    feature match:Operation
    feature match_taco:Method
    feature OD:Attribute
    feature OM:Attribute
    feature R:Attribute
    feature rb:Attribute
    feature TM:Attribute
    taggedValue area:TaggedVa
    taggedValue gd:TaggedValu
    taggedValue pc:TaggedValu
    taggedValue sysc:TaggedVa
    taggedValue vhdl:TaggedVa
  ownedElement MMU:Class
  ownedElement Network:Clas
```

```
« FU »
SHIFTER
{ sysc = "shifter.hpp", pc = 14.6, vhdl = "shifter.vhd", gd = 20, area = 28160 }
+TLR : 32b
+TLL : 32b
+TML : 32b
+TMR : 32b
+OP : 32b
+R : 32b
+rb : 1b
+shll (val : 32b, pos : 32b, res : 32b, b : 1b) :
+shlr (val : 32b, pos : 32b, res : 32b, b : 1b) :
+shar (val : 32b, pos : 32b, res : 32b, b : 1b) :
+shal (val : 32b, pos : 32b, res : 32b, b : 1b) :

« FU »
MATCHER
{ pc = 7.1, area = 13785, sysc = "matcher.hpp", gd = 22, vhdl = "matc
+TM : 32b
+OM : 32b
+OD : 32b
+R : 32b
+rb : 1b
+match (val : 32b, msk : 32b, sel : 32b, res : 32b, b : 1b) :

« NetCtrl »
Controller
{ vhdl = "netctrl.vhd", area = 28346, gd = 14, pc = 7.0, sysc = "netc
+RJMP : 32b
+AJMP : 32b
```

Methods: match_taco, match_c

Editing feature of Class MATCHER

name: match_taco

visibility: public

body:
```
msk > OP;
sel  > OD;
val  > TM;
R  >  res;
```

Figure 5.9: Caption of the TACO library for IPv6 in Coral[3]

```
        if not wfrFUTrigger(el):
            print "      FAILED!!! "
    if hasStereotype(el,"Result_Bit"):
        .......
```

The output of the script displays in the scripting editor of Coral the list of elements verified, the constraints verified for each element, and those constraints that failed.

5.4.4 System-level Estimation of TACO Configurations

Having all the estimation-related information gathered in the TACO UML models has enabled us to use the scripting facilities of Coral, for estimating the physical characteristics of the created TACO configurations. Physical estimates, like occupied area and power consumption, are obtained by querying the object model of a given configuration, from where the corresponding tagged values of each object's classifier are read and summed up. For instance, the following script computes the area of all FUs in a TACO configuration:

```
area = 0;
for ob in self.ownedElement:
  if ob.oclIsKindOf(UML14.Object):
    if ob.classifier.stereotype.exists(lambda st: st.name=="FU"):
      area=area + int(str(ob.classifier.taggedValue.select(lambda tv:
```

[3]The author wishes to thank PhD student Tero Nurmi for providing the physical estimates included in the TACO UML library

```
6        tv.name=="area")[0].dataValue))
7 print "Total area", area
```

The script selects (line 4) all instances of the ≪*FU*≫ elements, and adds the value of their *area* tagged value to the total area of the configuration. If the area occupied by the interconnection network controller or by other components is to be included in the calculation, only line 4 needs to be modified to include the corresponding stereotypes. A similar script is used to compute the power consumption of TACO configurations. As one may notice, the complexity and size of the script is quite low, being pretty simple to implement it.

Finally, based on the gate delays included in the profile, we are able to automatically estimate the clock cycle of the processor. This estimation is only preformed with respect to the gate delays introduced by the FUs, as follows: the FU with the largest gate delay will provide (a rough estimate of) the processor clock cycle for a given configuration. Due to space reasons, we do not include the script here.

5.4.5 Code Generation

In our approach, the SystemC and VHDL models of the processor are obtained from the UML models of TACO (see Figure 5.3). Although the SystemC and VHDL models are co-developed in a cycle-accurate manner, the simulation and synthesis code of a given configuration is generated independently.

Two code generation transformations have been defined to support the automated generation of the SystemC and VHDL models. Since all the processor components have corresponding implementations files in SystemC and VHDL, only the top level configuration files of the two models have to be generated. The generated files will contain information, like module instances, included in a configuration and their interconnection at port level. The transformation scripts are rather large in size, therefore we will not include them here. We briefly present the algorithm of the transformation using pseudocode:

```
for each object in the object diagram
   if object.classifier is <<NetCtrl>> or <<FU>>
       add module instance from the SysC tag
   if object.classifier is <<Network>>
       instantiate a bus for each <<Bus>> instance
for each <<Result_Bit>> link
   declare a sc_signal<bool> signal
   connect signal to <<FU>>
   connect signal to <<NetCtrl>>
for each <<Socket>> link
   compute register types
   for each FU register
       for each bus
           connect register to bus
for each <<FU>>
   assign triggers
```

The partial result of applying this algorithm to the TACO processor configuration (i.e., object diagram) of Figure 5.6, is presented below:

```
 #include "matcher.cpp"
 #include "counter.cpp"
 #include "shifter.cpp"
....
 #include "netctrl.hpp"
 #include "bus.hpp"

int sc_main(int argc, char* argv[]){
  sc_clock clk("clock",20);

  //instantiate SystemC modules
  NetControl nc("NetCtrl1");
  nc.clk(clk);
  Matcher m1("M", clk);
  Shifter s1("S1", clk);
  Shifter s2("S2", clk);
  Counter cn1("CN1", clk);

  //instantiate buses
  Bus* bus0 = new Bus("Bus0");
  Bus* bus1 = new Bus("Bus1");
....
```

```
  //declare and connect result signals
  sc_signal<bool> sigM1Guard;
  m.resultBit(sigM1Guard);
  NetControl::insertGuard(sigM1Guard);
....
  //connect M:MATCHER to bus0
  bus0->insertOperand(m1);
  bus0->insertData(m1);
  bus0->insertTrigger(m1);
  m1.assignTriggerIds();
  bus0->insertResult(m1);
....
  nc.initialize();
  sc_start(clk, n);
....
  }
....
  nc.initialize();
  sc_start(clk, n);
....
}//main
```

Following a similar approach, a script has been implemented to generate the VHDL model of the processor, in order to serve as input for hardware synthesis tools.

One remark can be made at this point. In the TACO SystemC model, the trigger sockets are dynamically created at run-time by the network controller. This is the reason why only one line of code (i.e., *m1.assignTriggerIds()*) is used, in the previous example, for connecting and generating the trigger sockets of the Matcher unit, whereas the rest of the sockets are automatically created when an FU is connected to the bus (see [132] for more details).

5.5 Mapping Applications onto the TACO Architecture

In this section, we discuss the process of mapping application specifications onto the TACO architectural specification. The process relies on the UML-based approach for application specification, which we have presented in Chapter 3, and on the use of the TACO UML profile. From the mapping process, a *configuration* of the architecture to support the application and the corresponding *program code* to drive the configuration are obtained. We exemplify the approach with the implementation of the IPv6 router application specification on TACO.

We recall that in our application specification process, only the functional requirements of the application are taken into account. Therefore, a *functional implementation* of the application, against the TACO architecture, is obtained from the mapping process. Later on, the functional implementation is tuned, during the design space exploration phase, to satisfy the performance requirements and physical constraints of the application.

In the mapping process, the artifacts of the application specification have to be expressed using concepts of the selected architecture. This approach poses a problem: the difference between their level of abstraction creates what researches refer to as *the implementation gap* [74, 118]. This gap is even wider when the target architecture is hardware-based. To narrow this gap, one solution proposed by the same researchers is the use of the programming model of the architecture. This approach enables, on the one hand, to raise the level of abstraction at which the architecture is specified and, on the other hand, to express the architecture in similar concepts with the ones used in the application specification. To benefit from such a programming model, it is important that a natural resemblance exists between the computation model of the application and the programming model of the architecture, in order to facilitate the mapping of their concepts. Kienhuis et al. [74] suggest three levels of *natural fitness* between the two models: granularity of operations, operational semantics and data types used. In the approach that we present in the following, only the first two levels of fitness are addressed, while the data type level is left for future investigations.

Throughout this chapter, the various models of the TACO architecture, including its programming model, have been discussed. In addition, a methodology for specifying protocol processing applications has been proposed in Chapter 3. The application model resulting from this methodology follows a control-oriented computation model, whose concepts fit naturally on the control-oriented programming model of TACO.

Seen from a model driven perspective, the TACO mapping process is similar to a PIM→PSM transformation, as specified by the MDA standard. In our case, the *platform independent model* (PIM) is represented by the application specification (e.g., IPv6 router), while the *platform specific model* (PSM) represents the implementation model of the application on a given architecture (i.e., TACO configuration and application program code). A *platform model* (PM) (i.e., the TACO architecture specification) is used to support the transformation.

In order to tackle the conceptual gap between the application and the architecture, and by benefiting from the abstraction layers of the TACO architecture, we have decomposed the mapping process into several smaller PIM→PSM transformations (Figure 5.10). At the highest level of abstraction, the mapping process is performed between the application specification model (i.e., PIM) and the programming model of the architecture, namely the *TACO functional primitives* (i.e., PM1.1). The application specification model is obtained from the specification process discussed in Chapter 3. Hence, the specification is represented in terms of activity graphs communicating via messages. The result of the transformation (i.e., PSM1.1) provides an "implementation" of the PIM, in which the subactivity states of the application model are expressed in terms of functional primitives of the architecture. From the model obtained (i.e., PSM1.1) the set of TACO resources needed to support the computation (i.e., PSM2) is identified by mapping the functional primitives onto the TACO hardware model (i.e., PM2).

In a different transformation, the control primitives of the activity graphs in the PIM1.1 model are transformed in control primitives of the TACO architecture

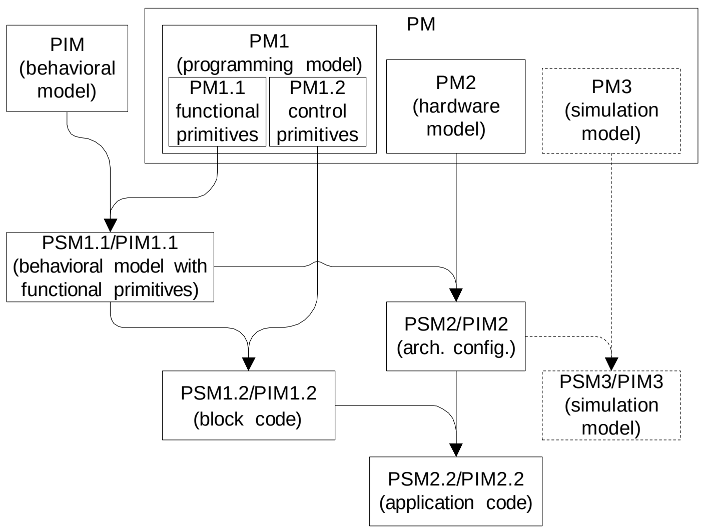

PM
PIM (behavioral model)
PM1 (programming model)
PM1.1 functional primitives
PM1.2 control primitives
PM2 (hardware model)
PM3 (simulation model)
PSM1.1/PIM1.1 (behavioral model with functional primitives)
PSM2/PIM2 (arch. config.)
PSM1.2/PIM1.2 (block code)
PSM3/PIM3 (simulation model)
PSM2.2/PIM2.2 (application code)

Figure 5.10: The PIM-PSM transformations supporting the mapping process

(i.e., PM1.2). The resulting artifact (i.e., PSM1.2) represents a model of the IPv6 router application, implemented on TACO. In this model, the entire application is expressed in terms of TACO programming primitives.

In a subsequent transformation (PIM1.2→PSM2.2), the programming primitives are expressed in terms of TACO bus transports, according to the resources in the TACO configuration under study.

Two observations can be made at this point: a) the transformation process can be further continued using other models of the architecture (e.g., to create a simulation model); b) the application code (PIM1.2) may suffer subsequent transformations (e.g., optimizations), before being executed on the TACO configuration.

In the following, we will discuss each transformation in more detail.

Expressing the Application Specification with TACO Programming Primitives

In the first transformation (i.e., PIM→PSM1.1) of the mapping process, the activity states are expressed in terms of functional primitives of the TACO architecture. The process is composed of two steps. In the first one, the subactivity states are hierarchically decomposed into less complex subactivity states until they match the granularity of the TACO FPs. This step is heavily based on the experience of the TACO domain expert and thus, performed manually. A certain level of tool support could be assumed in decomposing the subactivity states, though. In the second step, each "leaf" subactivity state is transformed into an action state, and implemented with one of the functional primitives provided by the programming model of TACO. We remind the reader that TACO FPs are available in the TACO UML Library (see Section 5.4.2 for details). We also remark that this step is similar

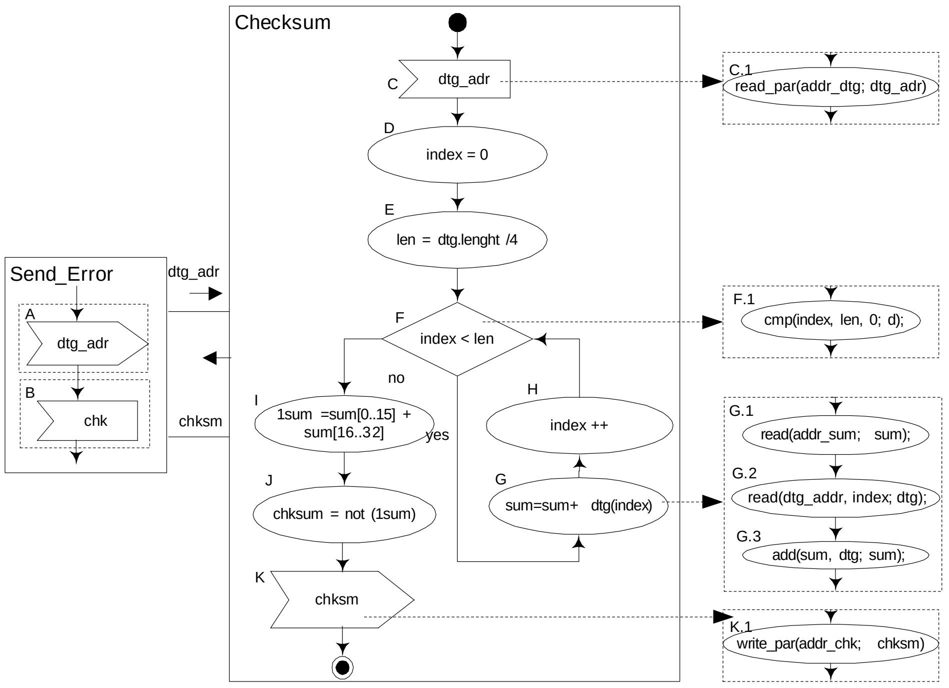


Figure 5.11: Decomposing activity states to match the complexity of TACO operations

to the "marking" process as suggested by MDA, and hence, it is not trivial to automate. Figure 5.11 presents an example where an activity graph representing the checksum computation, is refined using TACO operations. For instance, the sub-activity state denoted 'G' is split into three action states: 'G.1', 'G.2' and 'G.3', which match the granularity of and will be implemented using the TACO FPs.

Identifying the TACO Qualitative Configuration

The PSM1.1 model resulting from the previous transformation becomes the input (i.e., the PIM1.1) of the second transformation. In this transformation, the TACO components (i.e., the FUs) required to implement the selected TACO FPs are identified. The process is facilitated by having the correspondence between the TACO functional primitives and the TACO FUs stored in the TACO UML library. We have automated this transformation using a script that searches the TACO Library for those functional units that provide each FP of the PIM1.1 model. Based on the identified resources, a class model and subsequently, an object model of TACO are created. The TACO object model provides a qualitative configuration of the processor, in which only one FU of each identified type is present.

One exceptional case has to be attended in the transformation. In certain situations, the same FP may be provided by more than one FUs. For instance, the

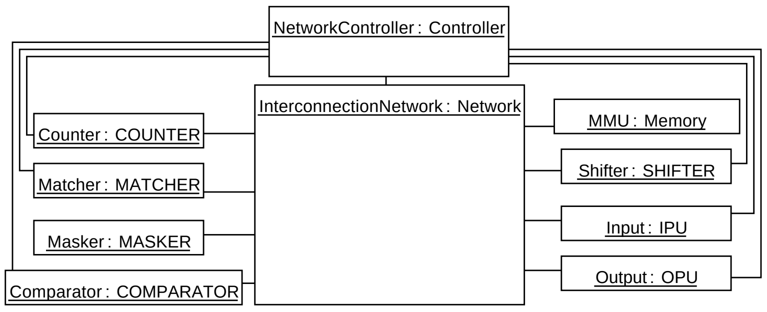


Figure 5.12: Qualitative configuration of TACO for the IPv6 router

selection of the 16 LSB of a 32-bit field, may be achieved using either a MASKER FU or a SHIFTER FU. In such cases, two approaches could be followed: either the user intervention is required and the she/he manually selects the desired FU, or a "preferred" component may be used for a given FP. In our implementation, we have decided to follow the former approach, in which the user intervention is required.

The TACO qualitative configuration resulting by applying this transformation to the IPv6 router specification is shown in Figure 5.12. We mention again that the presented model depicts a qualitative configuration of the processor where only one FU of each type is used. If a quantitative configuration is to be obtained from this transformation, performance requirements have to be taken into consideration in the application specification phase. Such an approach is subject to future work.

Creating the Application Code

The PSM1.1/PIM1.1 model of Figure 5.10 models the application specification in terms of activity graphs, in which the action states are implemented with TACO functional primitives. To obtain the TACO assembler code for driving the configuration obtained in the previous step, the PIM1.1 model has to be transformed into TACO programming primitives. The transformation is composed of two steps:

a) map the computation model of the application specification into the control structures of the TACO programming model;
b) express the TACO functional primitives in terms of TACO bus transports specific to each FU.

Step a) corresponds to the PIM1.1→PSM1.2 transformation, and it is based on the "natural fitness" between the activity graphs and the TACO programming model, as depicted in Table 5.4. Basically, we regard each action state as a subroutine that is invoked from other parts of the code. The starting point of a subroutine is identified by a *label*, while its end is indicated by a *jump* to the next subroutine. The approach enables us to regard the activity graph as a sequence of subroutines. Consequently, *transitions* between activity states are modeled using *unconditional*

Activity Graph Model	TACO Prog. Model
action state	moves
branch	conditional jump
transition	unconditional jump
send signal	subroutine call
receive signal	label

Table 5.4: The correspondence between the control structures of the Activity Graph and of the TACO programming model

jumps, while *guarded transitions* are implemented as *conditional jumps*. Beside control primitives, each subroutine also consists of a TACO functional primitive, supporting the functionality of the subroutine. The transformation process is based on the following steps:

a.1 each *activity state*, *branch state*, *send signal*, *receive signal* state is transformed into a TACO *subroutine* respectively, where the first instruction is a *label*;

a.2 a *transition* between two blocks is transformed into an *unconditional jump* instruction, where the address of the jump corresponds to the *label* of the target activity;

a.3 *guarded transitions*, only allowed in combination with branch states, are transformed into *conditional jumps* pointing at the *label* of the target subroutine.

The result of applying the transformation to the example in Figure 5.11 is shown below. In this example, the TACO FP implementing each action state has been replaced with the tag (e.g., *A.1*) of the corresponding activity states, in order to provide a better view of the structure of the presented code.

```
label A.1;          D.1;                                    JUMP J.1;
A.1;                JUMP E.1;           label G.2;
JUMP C;                                 G.2;                label J.1;
JUMP B;             label E.1;          JUMP G.3;           J.1;
                    E;                                      JUMP K.1;
label B;            e: JUMP F.1;        label G.3;
B.1                                     G.3;                label K.1;
JUMP ...;           label F.1;          JUMP H.1;           K.1.
........            F.1;                                    JUMP B.1;
                    d: JUMP G.1;        label H.1;
label C.1;          !d: JUMP I.1;       H.1;                label B.1;
C.1;                                    JUMP F.1;           B.1;
JUMP D.1;           label G.1;                              JUMP B.1;
                    G1;                 label I.1;         .....
label D.1;          JUMP G.2;           I.1;
```

In step b), corresponding to the PIM1.2→PSM2.2 transformation, the TACO FPs, used to describe the action states, are expanded into TACO *moves* (i.e., bus transports), based on the specification of each FP stored in the TACO UML Library. We recall that one or more transports are used to set up the input registers of the FU, to trigger and read the result of the computation. Thus, the transformation queries the library for each primitive in the PIM1.2, reads the bus transports of each operation, and replaces the parameters of the operation in the parameters of the bus transports. The following example presents a fragment of the code resulting from this transformation. Note that the “#” symbol is used to specify commented code.

```
LABEL G.1;
#read(addr_sum; sum); MMU FU
+00 > OPMM;         #base address for variables
addr_sum > TRMM;    #offset address
RMM > sum;          #read the result into sum
JUMP G.2;

LABEL G.2;
#read(dtg_addr, index; dtg); MMU FU
dtg_addr > OPMM;    #base address of the datagram
index > TRMM;       #offset of the next 32-bit word to be read
RMM > dtg;          #read the result into 'sum'
JUMP G.3;

LABEL G.3;
#add(sum, dtg; sum); COUNTER FU
sum > TSC;          #initialize counter with the 'sum' value
dtg > TIC;          #increment counter value with 'dtg'
RC  > sum;          # read result of the addition
JUMP H.1;

LABEL H.1;
#inc(index); COUNTER FU
index > TSC;        #initialize counter with the 'index' value
+01 > TIC;          #increment counter value by 1
RC  > index;        #read result of the addition into 'index'
JUMP H.2;

LABEL H.2;
#write(addr_index); MMU FU
+00 > OPMM;         #base address for variables
addr_index > TRMM;  #offset address of the 'index'
JUMP F.1;

LABEL F.1;
#cmp(index, len, 0; d); COMPARATOR FU
len > OPC;          #
index > TLTC;       # index < len, d is guard signal
d: JUMP G.1;        # conditional JUMP
!d: JUMP I1;
....
LABEL K.1;
#write_par(addr_chk, chksum); MMU FU
+00 > OPMM;         #base address for variables
addr_chk > TWMM;    #offset address
RMM > sum;          #write the result into memory
JUMP B.1;

LABEL B.1;
#read_par(addr_chk, chksum; chk); MMU FU
+00 > OPMM;         #base address for variables
addr_chk > TRMM;    #offset address
RMM > sum;          #read the result into sum
JUMP ...;
```

We remark that the *send signal* and *receive signal* activities constitute a special case. A shared memory location is used for passing the message from the sender to the receiver. Basically, the send signal activity writes the value of the message into a given memory location (lines 35-40) and the receive signal activity reads (lines 41-46) the value of the message from the same memory location.

Several remarks can be made regarding the code above. Firstly, each TACO instruction has been expanded into a number of bus transports based on its definition in the TACO UML Library. Secondly, an FU (e.g., COUNTER FU) may provide more than one operation, and each operation has its own implementation in terms of bus transports. Thirdly, the obtained TACO assembly code should not be seen as final code of the application. Before being compiled into executable code, optimizations are performed. For instance, the most obvious situation is the one of an unconditional jump to the next line of code. Such an instruction is completely unnecessary and implies additional clock cycles to the execution of the code. Other optimizations of the code may be performed with respect to the TACO architecture ([68, 32]), but we leave them as a task for the TACO compiler. At the moment, the TACO compiler is under development, and currently all the optimizations have to be applied manually by the designer.

A Library of IPv6 Processing Tasks. During the specification of the IPv6 router application, we have identified several functionalities of the application that have to be supported by any architecture used to implement the application. During this phase, the functional requirements of such processing task have been analyzed in detail and their implementation in terms of TACO operations has been specified. In our case study, several processing tasks required by the router have been identified:

- IPv6 header validation,
- IPv6 classification,
- Checksum computation,
- Routing table lookup,
- ICMPv6 signaling, and
- Routing table update.

These processing tasks may be seen as programming primitives of a platform-independent DSL for IPv6 routing. They can be used to specify other applications in the same family. In addition, these tasks may have custom implementations targeted to specific platforms. From a platform-independent perspective, in our approach these tasks are specified in terms of activity graphs. Each activity graph may spawn over several levels of detail (i.e., granularity), in which complex subactivity states are decomposed into simpler subactivity and/or action states. When a platform-dependent specification of each task is created, its action states are expressed in terms of functional primitives of the target platform. We have shown in the previous section how these tasks are refined and implemented with TACO FPs.

Taking advantage of the fact that the resulting specifications (activity diagrams) are graphically represented and stored as models provides us with the opportunity to reuse them when specifying applications in the same family (IPv6 routing). Consequently, we grouped these specifications into a UML-based *TACO IPv6 processing library*, where the specification of each processing task is stored as an activity diagram similar to the one in Figure 5.11. When the specification of a new application is created, and the need for a given processing tasks is detected, its activity graph is simply reused without having to perform the PIM→PSM1.1 transformation, which is a non-automatable step of the mapping process. We have used this approach to implement an IPv6 client application on TACO, in order to serve as a network interface for a networked multimedia processing SoC design [5, 135].

Another important benefit from the approach is that, by having already implemented the processing tasks in terms of TACO operations, we may easily determine the list of resources each of them requires, before creating the TACO configuration. Based on this approach, a rough estimation of the minimum physical characteristics of the underlying hardware needed to support a given task may be obtained, before performing the mapping process. We also mention that several specifications of the same processing task are allowed in the library, each of them using a different algorithm for supporting the task, or being tuned to use specific sets of resources of the existing TACO components.

5.6 Summary

In this chapter, we have proposed a DSL for the TACO architecture. The DSL has been defined as a UML profile and implemented into an existing UML tool (i.e., Coral), to provide rapid tool support for the TACO design process.

The profile models several abstraction layers of the TACO architecture. At the highest level, a programming model for TACO has been defined, to allow the designer to focus on the functionality of the architecture, rather than on its physical characteristics. On a different abstraction level, the class diagram of the TACO Profile provides a structural view of the TACO components and of their properties. In turn, object diagrams have been employed to create configurations of TACO, due to their low level of detail and easiness of instantiating TACO components.

The TACO Profile integrates several perspectives of the TACO framework like physical architecture, programming model, estimation models and simulation/synthesis models. These models complement each other, but there is no strict requirement that all of them are to be implemented or used at once. For instance, one may create a TACO model from scratch without including the programming model, simulation or synthesis information. In this case, the resulting model can only be used for evaluating the physical characteristics of different configurations.

The profile enables tool support for the TACO design flow (Figure 5.3) by reusing a number of UML-based editors of Coral, to create and edit TACO configurations. In addition, means have been implemented to enforce the consistency of the created models, with respect to the TACO architecture. Several reusable model transformations have been proposed for supporting the transitions between the steps of the TACO design flow. Some of these transformations have been automated by using model scripts implemented in Python, the scripting language of the Coral tool.

One of the main ideas behind the TACO framework is the use of libraries (at different levels of abstraction) to provide support for automation of the process and reuse of the components defined in previous applications. Using the TACO Profile, a UML-based library has been defined (i.e., the TACO UML Library), in order to encompass programming information, estimation details and simulation/synthesis information of the TACO components. This library may be seen as a high-level abstraction model of the TACO design framework presented in Virtanen's Ph.D thesis [132]. Following the TACO Profile definition and the associated well-formedness rules, lets one not only to store the library components as UML models, but also to verify their consistency every time a new element is added to the library. The approach also allows for several domain-specific libraries to be implemented, following a similar approach.

The mapping process discussed in Section 5.5 provides a bridge between our research on building the application specification (Chapters 3 and 4) and the modeling of the TACO architecture discussed in this chapter. The process has been regarded from a model driven perspective, in which well-defined models of both application and architecture have been used. Using these models, the main emphasis of the mapping process has been put on defining a PIM-to-PSM transformation that provides a systematic approach in identifying the architecture configuration and the corresponding application code. The transformation takes advantage of the abstraction layers of the TACO architecture. Among these, the TACO programming model plays an essential role, since it provides a software-based perspective of the architecture, closer (in terms of concepts and abstraction level) to the ones of the application specification.

By its nature, the mapping process, and thus, the corresponding transformations, are difficult to automate. In consequence, one of the goals behind this work has been to split the transformation into smaller parts and try to automate as many of them as possible. Taking advantage of the abstraction layers of the architecture, we have decomposed the PIM-to-PSM transformation into four atomic transformations, each performed at a different level of abstraction.

Although the presented mapping process has been customized for the TACO architecture, we consider that by applying the same principles we can define mapping processes for different other programmable architectures. As it resulted in our example, one key issue in mastering the complexity of the mapping process is using a well-defined programming model of the architecture that maps well on the computational model of the application specification.

Finally, a UML-based library for storing the mappings (i.e., design decisions) between the application specifications and architectural specification is proposed. The purpose of this library is to encode the implementation of the most common processing task of a given application domain (e.g., IPv6 routing), on a given architecture. On the one hand, the approach spares the designer the effort of going through the tedious mapping process by reusing already specified mappings. On the other hand, the set of physical resources needed by a given processing task may be identified a priori to the implementation of the entire application.

Based on the work in Section 5.5, several conclusions may be drawn:

- Taking advantage of the TACO UML Library, which encodes several abstraction layers of the architecture, enables fast access to the information of the TACO framework and provides the prerequisites for automating the mapping process.
- The use of the TACO programming model clearly helps in narrowing the implementation gap between the application and the architecture. In addition, having a natural fitness between the concepts of the two specifications facilitates the mapping of their operational semantics, and the selection of the functional primitives of the architecture that implement the application.

The secondary goal of the research presented in this chapter has been the evaluation of UML's suitability for defining and providing tool support for domain specific languages, that are target to programmable architectures. Several conclusions could be drawn:

- A graphical DSL for TACO could be defined by taking advantage of the UML extensibility mechanisms. The approach enabled us to use an existing UML tool (Coral) for providing tool support for the DSL, without requiring additional tool customizations. Furthermore, model transformations have been implemented in Coral to assist the design process by taking advantage of the scripting facilities of the tool.
- To create a UML profile, a good level of understanding of the UML standard is required, as well as of the OCL language. Furthermore, familiarity with the features of the UML tool used to implement the profile is necessary. This also implies knowledge of the programming language used by the tool.
- The quality of a profile, with respect to the benefits it provides to the designer, seems to depend very much on the capabilities of the tool. This also influences the UML elements that the designer chooses to express the application domain. Even if the mapping to UML concepts could look unnatural, if the graphics of the tool provide enough expressiveness, they will have priority in taking the implementation decisions.
- Using a "general-purpose" modeling language, like UML, for defining DSLs should allow for porting the profile definition between different UML tools and still maintain the profile usability. At the moment such an approach is not possible, not only because of the difference in how the UML metamodel is interpreted by tools, but also because of the difference in the capabilities of different tools.

- Clearly, the UML tools can be used for editing platform-specific designs, where other diagrammatic tool support is not present, as long as the representation of architecture concepts in UML is not too forced or unnatural.
- The usage of models to specify different parts of a system facilitated the reuse process by allowing us not only to easily create and edit graphical specifications of the system, but also to store and later on query and reuse already created specifications. Such an approach has an important impact on the level of automation of the design process and thus, reduces the development time of new products.
- Last but not least, having the artifacts of both the application and the architecture modeled in UML enabled us to perform the mapping process in the same UML tool and in addition, to benefit from the scripting facilities of the tool to for providing support for automation.

Future work includes several directions, as follows:

- We would like to improve the estimation perspective of the TACO Profile by taking into account the physical properties of all TACO resources. The current approach is mainly biased towards estimation of the FUs.
- The use of a UML profile for SystemC to describe the TACO components will also be considered. Such a profile could provide a graphical modeling tool for the TACO SystemC model. Moreover, it will introduce an intermediate abstraction layer between the TACO Profile and SystemC code, which will support the TACO-to-SystemC transformation in a UML-based environment.
- The TACO programming interface can be further improved by combining it with performance information. For instance, although each FU operation performs its operation in one clock cycle, several bus transports are required to setup and trigger the operation. Integrating this information in the TACO library would allow one to address the performance requirements of the application during the mapping process.
- Tool support for the TACO design flow should also be improved. In this section, several scripts supporting both queries and transformations have been implemented to assist the process. We plan that, in future versions of the profile, we will integrate them with the graphical Coral environment, such that they can be invoked by using buttons and context menus. Currently, they are executed in the scripting shell of Coral.
- We also intend to improve the mapping process, especially the part that is performed manually. Some help could be obtained by addressing the mappings between the data types involved in the application and architecture specifications, which have not been taken into account.
- At the moment the TACO IPv6 processing library does not take into account any performance characteristics of the application, but the idea is considered as a topic for future work. Future work may also look into integrating these processing tasks within the TACO UML library, in order to provide a single library model from which the TACO artifacts are used.

Chapter 6

A Metamodel for the MICAS Architecture

In this chapter[1], we discuss the use of metamodeling techniques for enabling the design of a programmable architecture, called MICAS, targeted to mobile multimedia applications. We mention from the very beginning that the work presented here is part of a larger project, in which eight persons participated at different stages. The work included in this chapter is intended to present only the contribution of the author of this dissertation, although a clear separation of the contributions of each project member is difficult to achieve. Some of the ideas included here, even if they have been suggested by the author of this study, are the result of numerous project meetings or have been subject to improvements based on the suggestions made by other project members. Therefore, we acknowledge the role of managing this project to professors Johan Lilius and Ivan Porres. Ph.D student Marcus Alanen also made an important contribution to the definition of the metamodel, and in particular, to the definition of the *MICAS Functional Library*. In addition, the excellent contribution of the master students Torbjörn Lundkvist and Tomas Lilqvist in providing the implementations and tool support for the solutions devised is greatly acknowledged. Last but not least, Kim Sandström, the architect of MICAS platform, suggested from the initial stages of the project valuable solutions for various design issues, which contributed significantly to the accomplishment of the project.

The chapter proceeds as follows. We start by briefly introducing the basic concepts of the MICAS architecture. Then, we propose a design process for MICAS, pointing out what deliverables are obtained at different steps of the process. A metamodel for the MICAS architecture is defined afterwards. The metamodel encodes the steps of the design process and provides several levels of abstractions at which the MICAS hardware is modeled. In addition, a programming model for the MICAS architecture is defined and integrated within the metamodel. The

[1]Some material included in this chapter was published in [P.8].

MICAS design process is accompanied by several libraries, defined at several abstraction levels and also integrated within the MICAS metamodel. Such libraries are intended to provide prerequisites for automation and reuse of MICAS components. After introducing the MICAS metamodel, we show how the design of MICAS architecture is supported by the metamodel implementation in the Coral tool. We also discuss the design decisions that we have taken in customizing the Coral editors, in order to support the design process. Throughout this chapter, the specification of a Digital Audio/Video Recording (DAVR) device is used as a case study.

6.1 The MICAS Architecture

***Mic**rocode **A**rchitecture For a **S**ystem On a Chip (Soc) (MICAS)* [109] is a novel concept developed at Nokia Research Center, Helsinki, Finland, which proposes both a SoC architecture for sequential data streaming processing systems (e.g., multimedia applications, personal video recorders, media streaming applications, etc.) and a method for controlling the hardware accelerators of such architectures. Several goals are pursued in MICAS:

- to separate the data-flow from the control-flow of the architecture, by using dedicated hardware units (*HW processes*) to assist data processing tasks, and *controllers* to drive the activity of these units;
- to decentralize the control communication from the "main processor" of the system, typically running a *real-time operating system* (RTOS), and distribute it to dedicated controllers which only control "local" resources;
- the use of *microcode* (i.e., software running on controllers) to control the functionality of the HW processes and the data streaming between them. The microcode provides a *hardware abstraction layer* (HAL) of the architecture, which enables to create data streams between HW processes and to invoke the functionality of a given HW process without having access to its hardware implementation.

MICAS may be seen as a programmable architecture not only in the sense that different combinations of processing tasks and data streams are used concurrently, but also in the sense that each HW process may be reprogrammed to perform different tasks. This enables that the same hardware configuration of the MICAS architecture to be used at implementing several applications in the same family.

6.1.1 Static Considerations

An overview of the MICAS architecture is given in Figure 6.1. A MICAS configuration comprises several *domains*. A *domain* represents a collection of hardware processing elements situated on the same physical silicon chip and controlled by the same *controller*. Domains provide fast processing speed for dedicated tasks.

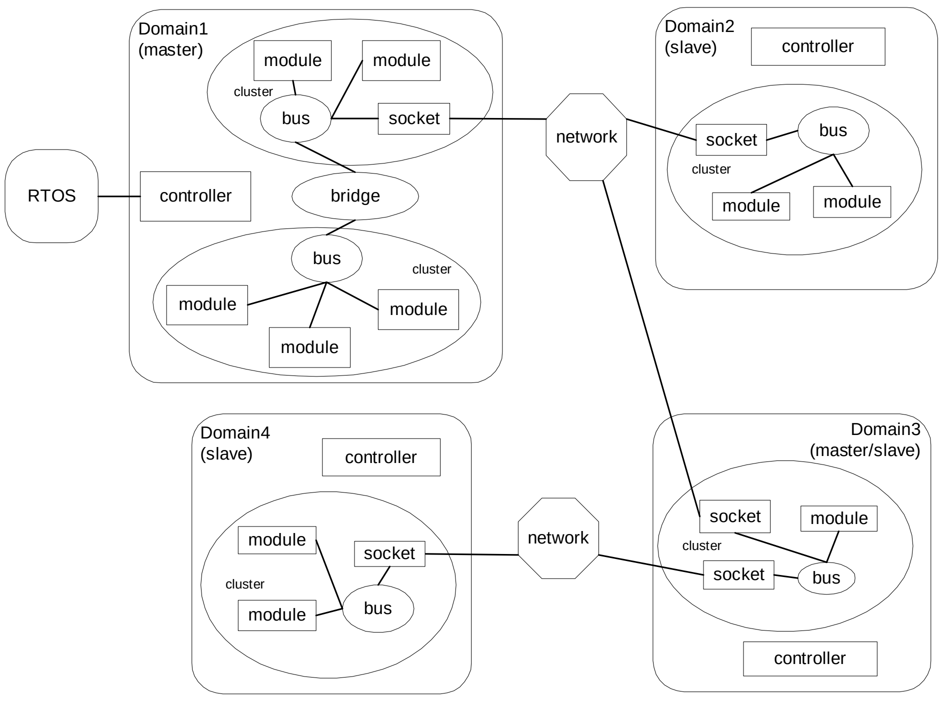


Figure 6.1: Generic view of the MICAS architecture

They are interconnected by off-chip external networks using for instance, serial, Bluetooth or WLAN technology.

Each domain may contain several HW processes, which implement dedicated tasks in hardware. HW processes are universally interconnected via *buses* and may be grouped into *clusters*. There are three types of HW processes inside a domain: *bridges*, *sockets* and *modules*. Buses belonging to different clusters may be connected to each other through *bridges*. *Sockets* intermediate and transform the on-chip communication into off-chip communication, whereas *modules* implement dedicated processing tasks over data streams.

The organization of domains is hierarchical, following a master-slave relationship. Typically, one domain of a given MICAS configuration may be attached to a GPP running an RTOS (e.g., Symbian – `http://www.symbian.com/`). Such a domain is called *master domain*. Domains connected to a master domain are considered to be *slave domains*.

A given domain may have both a master and a slave role with respect to other domains. For instance, in Figure 6.1, *Domain3* has a slave role with respect to *Domain1*, and a master role with respect to *Domain4*. Consequently, *Domain1* is considered to be *'the master'* of *Domain4*. Since a domain has a single controller to manage its resources, we can also classify the controllers into *master controllers* and *slave controllers* based on the relative role of their domains. In MICAS, a

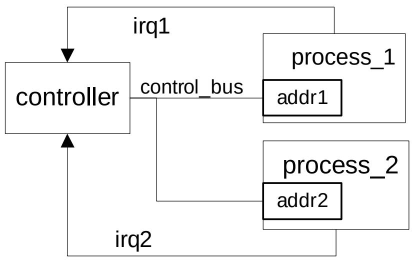


Figure 6.2: Communication between the controller and HW processes in MICAS

master domain requests processing tasks from its slave domain. In practice, the master controller issues commands to its slave controllers which, in turn, program the local HW processes to perform the requested tasks.

The communication between the controller and HW processes is implemented using an interrupt-based mechanism in one direction, and a *control bus* in the opposite direction (see Figure 6.2). Each HW process has assigned an IRQ (*Interrupt ReQuest*) number which is used to interrupt the controller, and a unique address space to which the controller writes commands to program that HW process.

The communication between HW processes is supported by so-called *data buses*, which are compliant with the *Open Core Protocol* (OCP) standard specification [99]. Throughout this chapter we use the terms *MICAS data bus* and *OCP bus* interchangeably, they both referring to a bus connecting two HW processes. HW processes have master-slave relationships over the data buses. As such, each HW process has assigned an address space to uniquely identify itself on a given bus. Moreover, a HW process may be connected to several buses and it may play a different role (i.e., master or slave) on each bus. Figure 6.3 provides an example of four HW processes connected to two different OCP buses. In this example, *process_2* has a master role relative to the *OCPBus1* bus, and a slave role relative to *OCPBus2* bus.

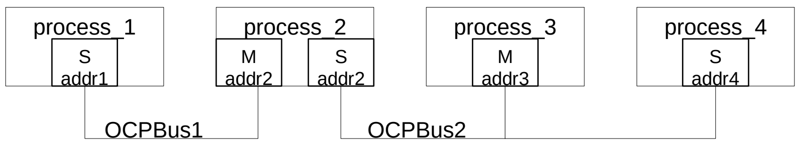


Figure 6.3: Communication between HW processes over OCP buses

6.1.2 Dynamic Considerations

The dynamic aspect of the MICAS architecture relates to how the data streams between HW processes are implemented on top of the hardware architecture. Two aspects are involved. On the one hand, the communication between modules has to be dynamically controlled in order to change the source and sink HW processes of each data stream at run-time. On the other hand, the HW processes have to be programmed in order to provide the corresponding processing function for each particular type of stream. The entire dynamic behavior of a given domain is controlled by the software (i.e., microcode) running on the controller.

Controllers serve as a control interface to any external entity (i.e., domain or RTOS). Any requests for processing tasks received from the external environment of a given domain are handled by the controller, which dispatches the corresponding commands to the appropriate HW processes. Three communication primitives (i.e., *microcommands*) are used to support this communication:

- *inquiry* – used in the inter-controller communication to interrogate the resources of a given domain for their status, availability, functionality provided, etc.;
- *setup* – configures local HW processes to perform a specific task;
- *tunnel* – transports one of the previous microcommands over the external networks, in order to be executed by the slave controller.

These three microcommands provide a HAL for each domain, which may also be seen as a low-level programming model of MICAS. Using the MICAS HAL, the architecture may be programmed to provide support for several applications, without modifying the hardware configuration of domains. A conceptual example of the MICAS HAL is presented in Figure 6.4.

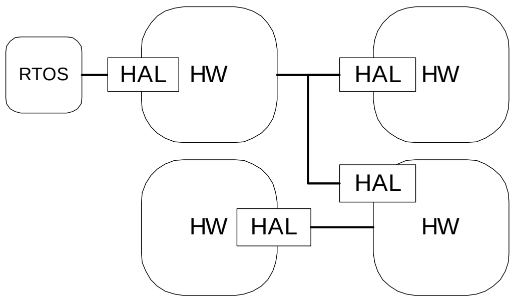


Figure 6.4: The hardware abstraction layer in MICAS

6.1.3 Control- and Data-flow Separation in MICAS

The MICAS architecture focuses on the separation of control and data-flows by moving away from the controller the data processing part to specialized hardware units and by implementing data streaming between these units. Thus, two distinct flows of information are present in a MICAS configuration: a) a sequence of commands (i.e., *control-flow*) issued by the controller to program master HW processes, which in turn program their slave HW processes; b) a sequence of data streams (i.e., *data-flows*) between HW processes.

An example is shown in Figure 6.5. To set up a data transfer between two HW processes, the controller first issues a command (*1. transfer_request*) to the master HW process (i.e., *process_1*) in order to request data from a specified slave HW process (i.e., *process_2*). The master HW process initiates the transfer over the bus by requesting (*2. req_data*) the slave to provide data and, consequently, the bus transfer is performed. When the data transfer is completed, the slave notifies the master (*3. data_sent*), which, in turn, notifies the controller (*4. transfer_done*) that the transfer is completed, by raising the interrupt signal. For the opposite direction of the data-flow, the exact same sequence of commands can be used, only that the slave HW process is programmed to receive data.

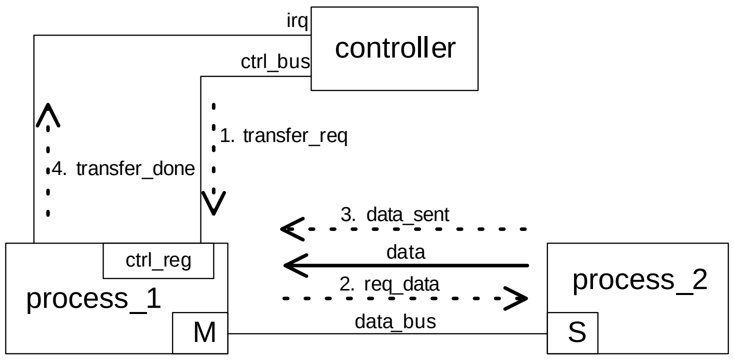


Figure 6.5: Separation of control (dotted arrows) and data (continuous arrows) flows in MICAS. Solid lines represent physical transfer connections like buses and signals

6.2 The MICAS Design Process

In this section, we define a design methodology (Figure 6.6) for the MICAS programmable architecture. The methodology relies on both the static (i.e., hardware configuration) and the dynamic (i.e., programming interface) perspectives of the MICAS architecture. The flow is composed of three phases (i.e., *Configuration*, *Design* and *Implementation*), each of them modeling the system at different levels of abstraction. Component libraries are defined, at each abstraction level, in order to provide reuse and support for automation.

A list of complex processing tasks (*services*) necessary to implement the application are identified from mapping the application specification to the MICAS architecture. The MICAS architecture is then configured, in the *Configuration phase* to provide the requested services. During this phase, the designer selects the static (hardware) resources necessary to implement services and distributes these resources to different domains. If already defined domain configurations, which provide some of the required services, are present in the *Domain Library*, they are

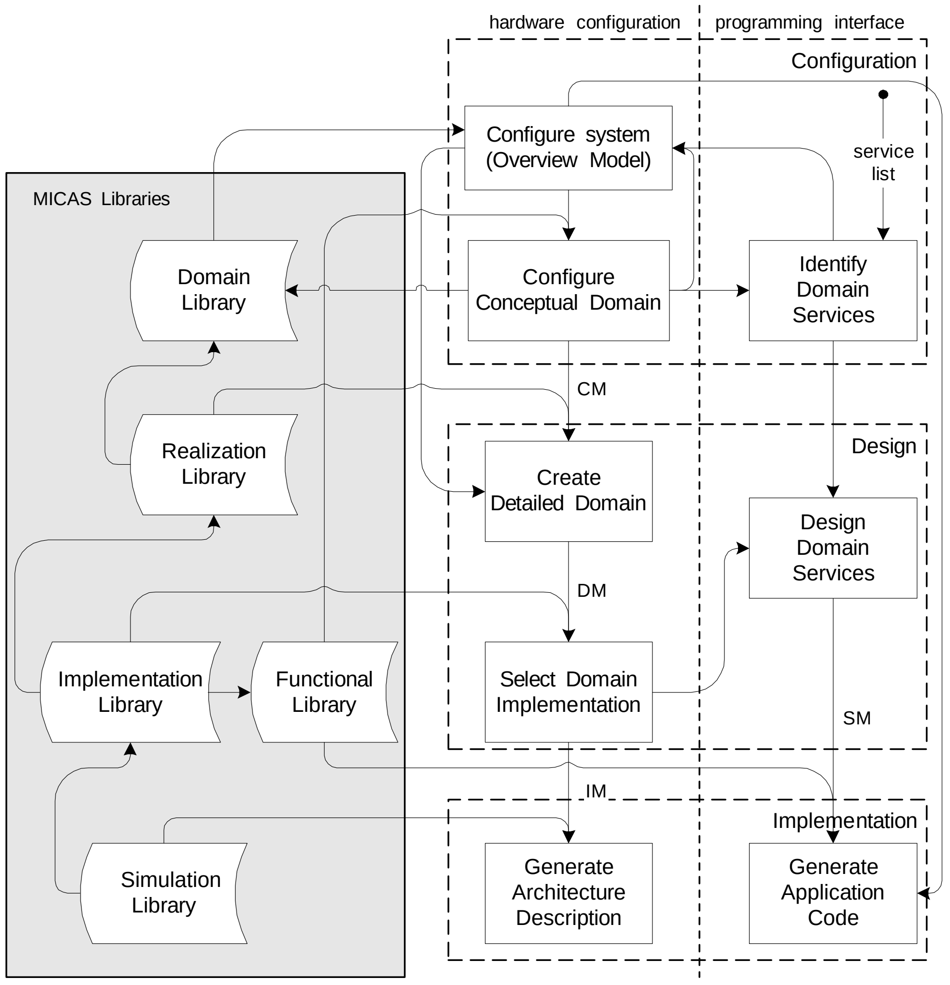


Figure 6.6: The design process of the MICAS architecture

added to the system configuration (i.e., the *Overview Model*). Alternatively, new domains are created and their functional configuration is defined by the designer. It is worth noting that predefined domains may provide additional services than the ones required by the application specification and thus, the *service list* may be updated or refined, when predefined domains are added. In addition, a redistribution of services onto domains may be necessary. The approach is currently performed manually, based on the experience of the designer.

During the configuration process, one identifies not only the internal configuration of each domain, but also how domains are interconnected to each other and to the RTOS. As such, three artifacts result from this phase: a) a top-level configuration of the systems in terms of domains – *Overview Model* (OM); b) an internal configuration of each domain, seen from a functional perspective – *Conceptual Model* (CM); c) a list of processing scenarios (i.e., *services*) for each domain – *Service Model* (SM) – in which each service is assigned to a domain and expressed in terms of data-streams between HW processes.

In the *Design phase* several design decisions are taken regarding the implementation of the functional components in hardware. The process is based on two steps. In the *Create Detailed Domain* step, the control communication is refined using specific mechanisms and a *detailed model* (DM) of each domain is obtained. In the *Select Domain Implementation* step, IP-based components are selected for implementing each detailed component, and consequently, an *implementation model* (IM) is obtained. Some of the IP components are accompanied by a list of commands (their programming interface), which is used by the controller to program them for processing different stream types.

The *Implementation phase* deals with transforming the artifacts resulting in the previous phase into an executable specification, which is later on used for simulating the system. There are basically two deliverables resulting from this phase: the *architectural description* of the devised MICAS configuration, and the *application code* to run on this configuration.

Several libraries are used to support the design process at different abstraction levels. At the lowest level, the *Simulation library* provides executable specifications of the MICAS hardware components. The *Implementation library* represents an abstraction of the SystemC component specifications by depicting their structure in terms of ports, interfaces and behavior. On a higher abstraction level, the *Realization library* encodes communication mechanisms which refine the control communication between functional components and controllers. Finally, a *Domain library* is used to store and provide already configured domains, which can be used to create top-level configurations of MICAS. The devised MICAS libraries are built incrementally in a layered fashion. This means that library components on lower levels are used to refine the library components on higher levels.

6.3 The MICAS Metamodel

In this section, we present a metamodel for the MICAS architecture and we discuss the main design decisions taken during its definition. The metamodel has been devised to support the steps of the MICAS design process by defining what MICAS concepts may be used at each abstraction level, and in addition, by imposing a certain order in which different MICAS models are created and used. Furthermore, the metamodel includes definitions not only for the static and dynamic perspectives of MICAS, but also for the libraries used to support the design process.

The MICAS metamodel is intended to provide a graphical DSL, which may be used by the MICAS designer without prior knowledge of UML. Nevertheless, the design decisions taken during the metamodel specification have also been biased towards facilitating the metamodel implementation in the Coral tool.

The metamodel is split into three parts:

- the *Static perspective* is concerned with modeling those hardware resources of the MICAS architecture that are found inside a given domain (e.g., controllers, modules, buses, etc). There are two levels of abstraction at which the MICAS hardware may be specified. The *Conceptual level* depicts the basic hardware resources based on their functionality without taking into account their concrete hardware implementation. The *Detailed level* takes into account communication related issues, like communication mechanisms and IP-components used to support this communication;
- the *Dynamic perspective* of the metamodel, defines a programming model of the MICAS architecture, in terms of services and their implementation (data-flows between MICAS resources);
- the *Domain Overview perspective* presents the existing domains in a MICAS configuration, and their static interconnection. In addition, the dynamic properties (i.e., services) of each domain, are provided as a programming interface (API) of the configuration.

6.3.1 General Considerations

The topmost element of the metamodel is the *MicasSystem*, specifying any kind of MICAS architectural configuration. A configuration consists of one or more "groups" of components. Such a group is modeled by an abstract *Container* element. A specialization of this element, namely a *DesignContainer*, encompasses all the elements in a given domain. A *ModelElement* generically describes any kind of MICAS element present in a domain. As such, the *Domain* is defined as a specialization of the *DesignContainer*, and consequently, it may contain any kind of *ModelElements*. The relationships between these elements are shown in Figure 6.7.

6.3.2 The Static Metamodel Definition

Figure 6.7 also presents a part of the MICAS metamodel definition, which models the static perspective of the MICAS architecture. The elements conform to the description of the MICAS architecture, as presented in Section 6.1. Based on this description, we can divide the hardware resources of MICAS into two categories: *Components* and *Links*. Basically, they provide the basic framework for creating and interconnecting any kind of MICAS static resources.

Conceptual-level Elements

At the conceptual-level, the components of the MICAS architecture are classified into three types: *Microcontroller*, *HWProcess* and *Bus*. Any domain has exactly

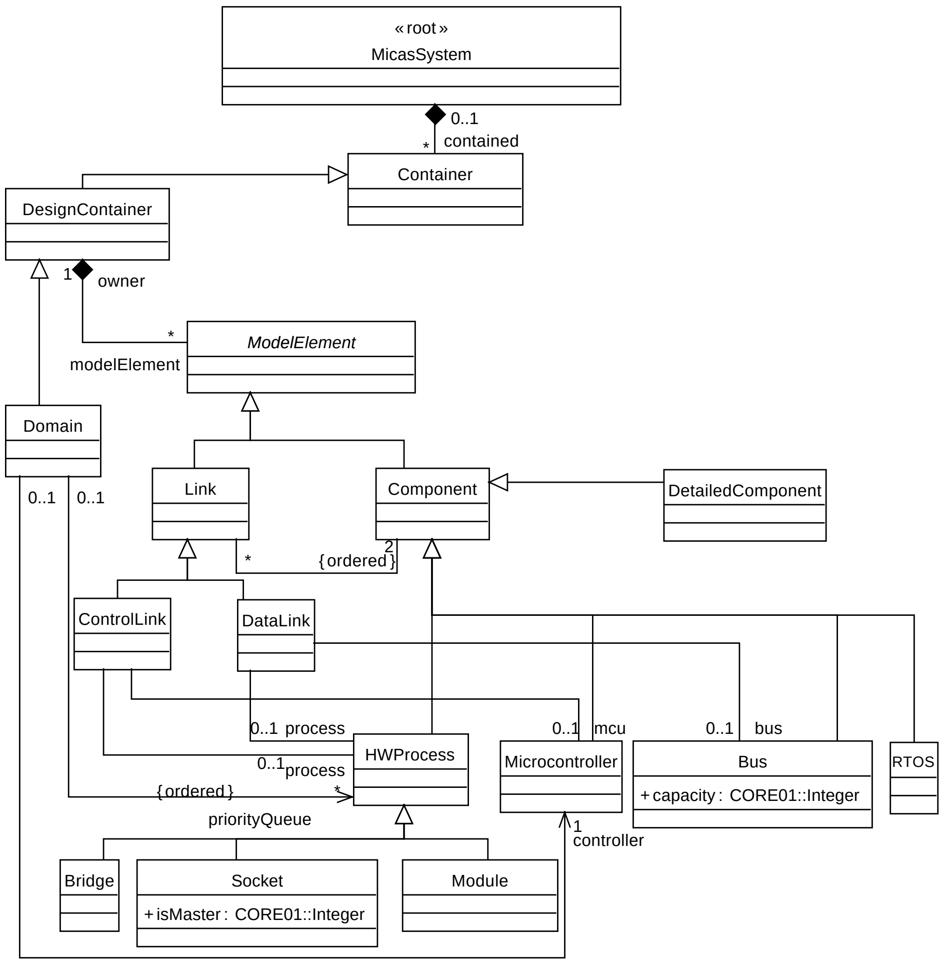


Figure 6.7: MICAS Static metamodel definition

one *Microcontroller* element and therefore a directed association has been used to point at it. A process can be classified either as a *Bridge*, a *Socket* or a *Module*. Some HW processes may be connected to the controller through an interrupt signal. Each such HW process must have a unique IRQ number assigned to it. The list of the IRQ numbers in a domain provides the *priorityQueue* of the interrupt mechanism of a given controller. Although theoretically, this queue should be regarded as a property of the controller, we have decided to model it as a property of the *domain* element. Such an approach enables the IRQ numbers to be dynamically assigned, when the configuration of a domain is created, even if a controller is not yet present in that model.

There are two types of buses in a domain: control and data buses, respectively. In our approach, we have defined data buses, modeled by the *Bus* element, as a specialization of a *Component*. The motivation for this decision is to enable the connection of several MICAS components to the same bus. A *DataLink* element has been defined, in order to model the interconnection between HW processes and data buses. In addition, a property *capacity* has been added to the *Bus* element, to allow the specification of the physical transfer rate of the bus either based, for instance, on estimation models or as a function of the bus width and bus clock. In turn, control buses are used to depict the connection between the controller and master HW processes, in a point to point manner. We have decided to model them as a kind of *Link*, namely as *ControlLink*. Finally, an *RTOS* element has been defined to model the operating system communicating with the master controller. For convenience reasons, we have decided to define the *RTOS* as a component of the domain.

Detailed-level Elements

The focus of the MICAS detailed level is to refine the communication between conceptual components by using dedicated communication mechanisms. Following a "by the book" approach, a new set of metamodel elements would have been needed for classifying the exact types of components found at this level. Nevertheless, we have decided that one single element, namely the *DetailedComponent* (see Figure 6.7), suffices in abstracting all possible MICAS detailed components. This approach is based on the following observations: a) due to the large variety of element types that can be used at this level, the MICAS metamodel definition may become quite complex; b) this level of abstraction is only supposed to be an intermediate step before the code generation process, and therefore, the editing of these models by the designer should be limited.

Since the *DetailedComponent* is a specialization of the *ModelElement*, it is possible to interconnect any *DetailedComponent* using already specified *Link* elements. Therefore, we can conclude that, at the detailed level, only two static metamodel elements are needed to cover the modeling of MICAS systems: *DetailedCompoment* and *Link*.

The MICAS Realization Library

A *Realization library* has been devised for supporting the transition between the conceptual and detailed models of MICAS. The library encodes the correspondence between conceptual element types and detailed element types. This correspondence comprises two kinds of information: a) *types* of implementation components used to refine the conceptual model, and b) their communication mechanisms (e.g., memory mapped, interrupt based, etc). The two kinds of information are strongly dependent on each other, in the sense that certain component types necessitate the use of a specific communication mechanism.

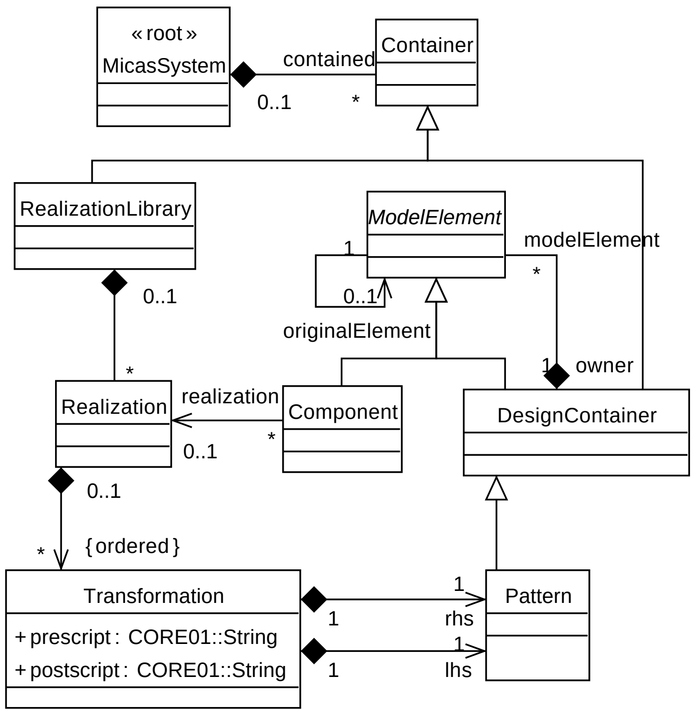
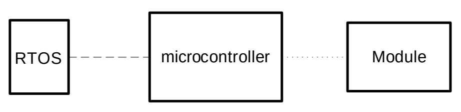


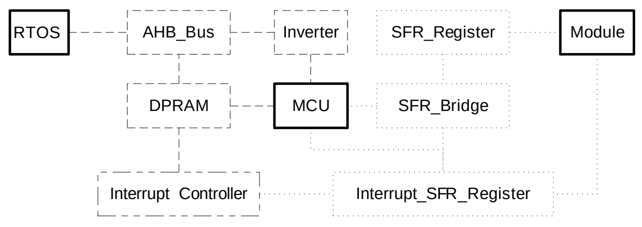
Figure 6.8: Realization Library definition

The definition of the Realization library is shown in Figure 6.8. The *RealizationLibrary* element itself is a specialization of the *Container* element. The library contains several *Realization* elements, each modeling a design decision for a "possible choice" to architect the system. A *Realization* is based on one or more atomic operations, called *Transformations*. It is the *Transformation* element that encodes the correspondence between sets of conceptual and detailed elements. This correspondence is graphically represented in terms of left-hand-side (LHS) and right-hand-side (RHS) patterns. The *Pattern* element has been defined as a specialization of the *DesignContainer* element; thus, it may contain any of the MICAS static elements previously defined. An example is given in Figure 6.9. Each element in Figure 6.9-a may be seen as the LHS pattern of a given transformation, whereas the elements in Figure 6.9-b that are drawn with a similar line pattern, may be regarded as the corresponding right-hand side (RHS) pattern of each transformation.

In this example, the three interconnected elements at conceptual level (i.e., RTOS, microcontroller, Module) are refined into their corresponding detailed elements, depicted in a thicker line pattern. Thereafter, the communication between conceptual elements, is refined into groups of detailed components supporting this communication (see elements drawn in dashed and dotted line patterns).



(a). Conceptual model



(b). Detailed model

Figure 6.9: Realization example for MICAS

Specifying realizations as collections of LHS/RHS patterns, enables us to create and store these realizations in a graphical manner, such that they can be documented and reused, whenever necessary. We mention that we have designed the Realization library to allow the specification of the *Transformations* in a generic manner, independently of the algorithm used to implement the transformations.

We also emphasize that all the MICAS components included in the library are regarded from the point of view of their type with respect to the MICAS architecture, and not from the point of view of their specific functionality. In other words, at this point, it is more relevant for the designer to know that a *Module* communicates with an *MCU* via an *SFR_Register*, rather than knowing that the same *Module* is an "encoder" or an "FFT" module.

The MICAS Implementation Library

If the Realization library focuses on the types of detailed MICAS components and their interconnections, the *Implementation library* provides implementation solutions (i.e., IP components) for each detailed component type. The main purpose of the library is to specify the general structure of such components, in order to facilitate, later on, the generation of the simulation models of a given MICAS configuration. The specification of the implementation components depicts their structure in terms of interfaces and ports. In addition, it provides pointers to the specification files used for simulating the components.

Figure 6.10 presents the metamodel definition behind the *ImplementationLibrary*. The library contains *LibraryElements* of two types: the *Specification* element "models" and points to the corresponding hardware specification (SystemC in our case) of each component, while *RealizationInterface* specifies reusable col-

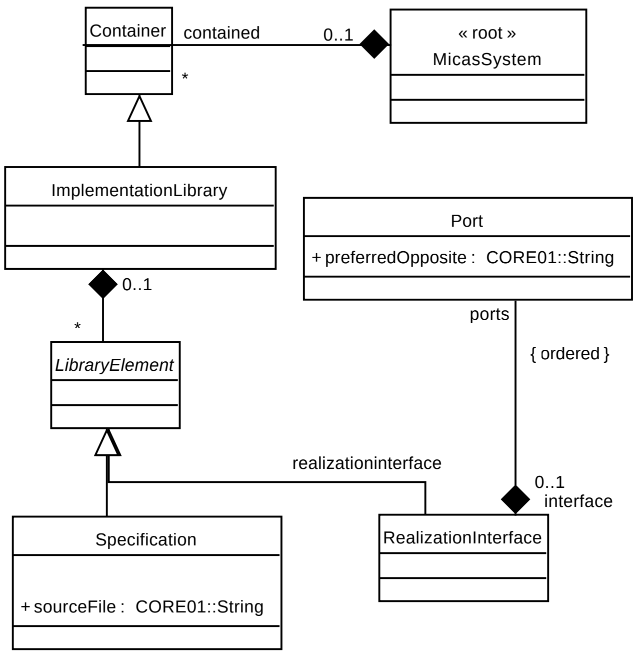


Figure 6.10: Implementation Library definition

lections of *Ports*. One may notice that the library specification allows for the same set of interfaces of a given implementation component to be used in combination with different SystemC specifications of that component, or even with different specification languages (e.g., VHDL).

Having an exact description of the interfaces and ports of each component available, enables us to automate the generation of the simulation code, where ports belonging to interfaces of different modules are interconnected. In special situations, rules can be defined to help in connecting “atypical” interfaces. For instance, ports with the same type and belonging to the same interface type can be directly connected, but in certain cases this approach cannot be employed. In this scope, a *preferredOpposite* property of the *Port* element has been specified, in order to explicitly point to the opposite port.

6.3.3 A Programming Model for the MICAS Architecture

The MICAS specification [109] suggests the use of *microcode* for programming the MICAS architecture. We recall that three microcommands (i.e., *setup*, *tunnel* and *inquiry*) are defined to exploit the functionality of the MICAS hardware. Of the enumerated microcommands, only *setup* is used to program the functionality of the HW processes. To implement an application on the MICAS architecture, a sequence of *setup* commands is required for programming the resources of each domain and the data streams between them. The approach is similar to the one in TACO, with the difference that several domains, and consequently, several controllers are used.

The set of commands of a given MICAS configuration provides the *programming interface* of that configuration and is used by the RTOS to implement a given application on MICAS. Although such an approach can work perfectly well, several issues may be seen as problematic. On the one hand, the level of abstraction of the MICAS API is relatively low and thus the programming of the architecture becomes difficult at RTOS level. In addition, the communication between the RTOS and the master controller becomes too complex, due to the large number of requests for small pieces of functionality. On the other hand, the microcommands provide a control-oriented view of the system that does not model appropriately the data streaming perspective of MICAS. Therefore, in this section, we suggest a graphical *programming model* for MICAS. This programming model can be seen as having two levels of abstraction:

- At application level, the programming model consists (similarly to the TACO programming model) of two categories of primitives: *functional primitives* and *control primitives*. Functional primitives are represented by the *services* provided by each domain, whereas control primitives are operations of a domain used to support the invocation of these services (i.e., *inquiry*, *allocate*, *deallocate*, *activate*, *deactivate*).

- At domain level, services are implemented in terms of data streams between HW processes. As such, a DFD-like diagram is proposed as a graphical modeling language for the MICAS programming model. We have presented the main concepts behind DFDs in Section 4.1. DFDs map fairy well on the concepts of the MICAS architecture: HW processes are transforming input streams into output streams, similar to the DFD processes transforming input data-flows into output data-flows. The use of a data-flow based programming model in MICAS allows the designer to program the functionality of a domain (i.e., its *services*) in terms of *streams* of data transferred and processed by HW processes. Consequently, the three concepts used by the MICAS programming model at domain level are the *services*, *streams* and *HWProcesses*. Following a similar approach would not have provided additional benefits in the case of the TACO architecture (see Chapter 5), since in TACO there is not a clear separation of the control and data paths. In fact, all the control and data communication in TACO, are basically done over the same interconnection network.

In this chapter, we mainly focus on specifying the MICAS programming model at domain level, since this is the part relevant for designing the MICAS architecture. In change, the application-level programming model is presented in Chapter 7, where the simulation of the MICAS architecture is discussed.

The Dynamic Metamodel Definition

The programming model of the MICAS architecture is also designated as the *MICAS Dynamic perspective*. As previously mentioned, three concepts are modeled by the dynamic perspective: *services*, *streams* and *HWprocesses*. A *service* models a complex processing task provided by a domain. As such, the set of services provided by a given domain may be regarded as the programming interface of that domain. *Streams* model data-flows between the HW processes of a domain. Two kinds of streams are used. *BasicStreams* model data-flows between adjacent HW processes (i.e., HW processes connected by the same physical bus). A consistent combination of basic streams forms a *CompositeStream*, which is seen as a possible implementation of a *Service* (as we will discuss in more detail in Section 6.4.7). Consistency refers to the fact that this combination allows the transformation of all input streams into output data streams. In our approach, we regard the streams as internal entities of a domain, not visible beyond its border. A given *BasicStream* may exist as a stand alone element, and may be used by several *CompositeStreams*.

Figure 6.11 presents the dynamic part of the MICAS metamodel. The building brick of the dynamic perspective is provided by the *BasicStream*. A basic stream has a source and a sink HW process, respectively, modeled as an ordered set, in which the first element is always the source and the second is always the sink. The data-flow perspective of MICAS abstracts away the physical connection between HW processes (i.e., the buses). In practice, each basic stream is transported over a physical bus. Since during the stream design process the designer should have access to the properties of the underlying hardware, a *bus* property has been added to the basic stream, in order to indicate the physical bus, over which the stream is transported. In addition, the basic stream is characterized by a transport capacity (i.e., *requiredCapacity*) depicting the number of Mbps required for that stream during transfer. We mention that the *requiredCapacity* is a design requirement and is different from the physical capacity of the bus.

Defining services per domain basis enables the designer to try out several overview configurations in order to create different combinations of services (scenarios). The metamodel has been designed such that it allows the combination of services either at configuration-time (i.e., scenarios created by the designer) or at run-time (i.e., the user/application combines services provided by a given configuration into scenarios). In the former case, scenarios can be created by specifying how services of master domains use the services of their slaves, in order to provide the requested scenario. Hence, the concept of *subservice* has been defined to depict a "slave" service used by a "master" one. Both approaches need to use a service discovery mechanism to identify, at application-level, existing services provided by the configuration. The difference between the two is that the former performs the service discovery process only for the master domain (the rest of the services being used as subservices), while the latter discovers the services provided by all domains in a given configuration.

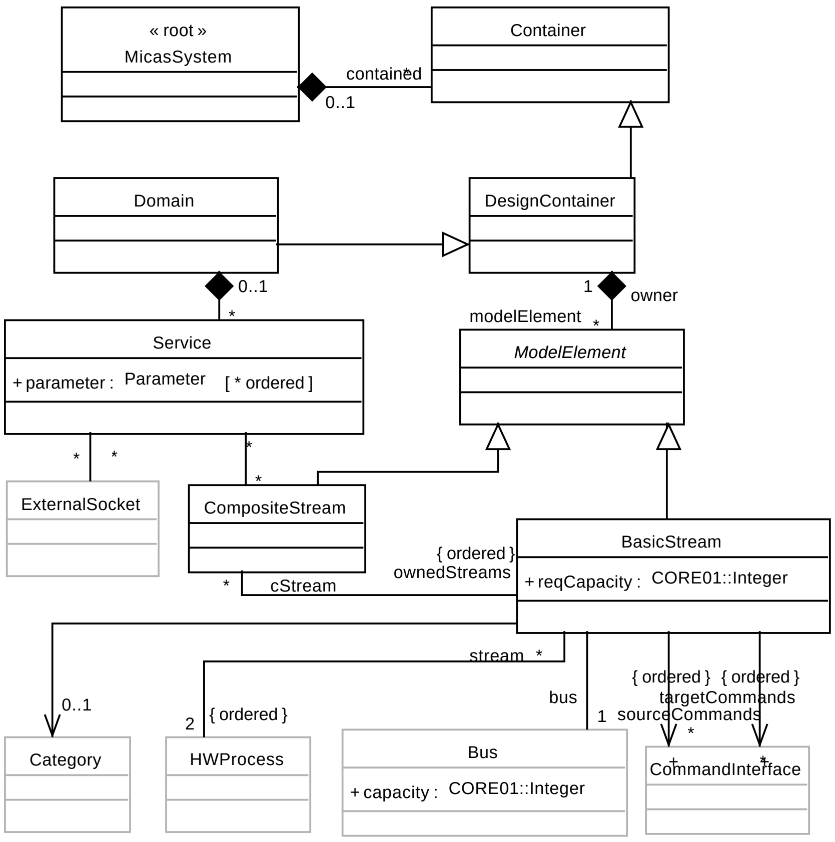


Figure 6.11: MICAS Dynamic metamodel definition

For ensuring a connection between the static and dynamic aspects of the MICAS architecture, the concept of *Category* has been introduced (Figure 6.12). From a data-flow point of view, the category depicts the type of data transported by a basic stream (e.g., audio, video, etc.). From a hardware perspective, it facilitates to characterize the functionality of a given HW process, in terms of input (i.e., *acceptedCategory*) and output (i.e., *providedCategory*) stream types, that it can process. Each HW process may process more than one type of input stream, or it may apply several computations to the same type of input stream. For the sake of simplification, and based on the observation that MICAS sockets and bridges are HW processes that simply transform input flows into output flows without affecting their type, we have defined the *acceptedCategory* and the *providedCategory*, only as properties of the *Module* elements.

The *Category* is used also as a bridge between the conceptual and detailed level representation of a HW process. If at conceptual level, a given HW module is characterized by the *acceptedCategory* and by the *providedCategory*, at detailed level, the implementation (that is the *Specification*) of the HW process may be charac-

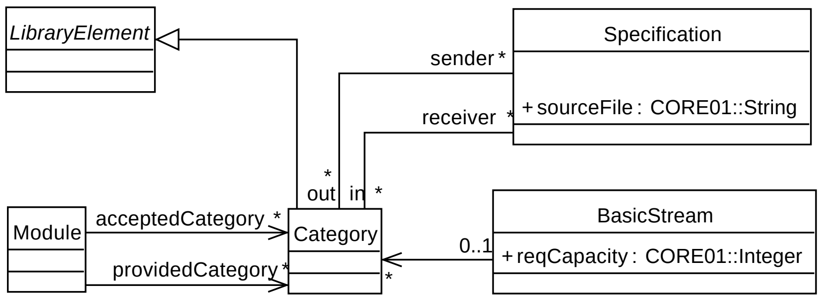


Figure 6.12: Relationships between *Category*, and the static and dynamic elements of the MICAS metamodel elements

terized in a similar manner. Thus, we have defined input (i.e., *in*) and output (i.e., *out*) categories as properties of the *Specification* element. The approach facilitates the selection from the library of implementation modules for a given HW process based on their functionality.

Furthermore, in order to provide reuse of already defined categories in a MICAS configuration, we store the *Category* as a component of the Implementation library, that is the *Category* is defined as a specialization of the *LibraryElement*.

MICAS Functional library

There can be several modules providing identical functionalities in a MICAS system, but using different implementations. One such example can be the one of a "Camera" module that, as generic functionality, captures video from the environment. However several implementations of the "Camera" module may be available from different vendors. To allow the designer to focus on the functional details of the HW processes, and in addition, to create a correspondence between the functionality provided by a conceptual element and the implementation of this functionality in terms of commands, a *Functional library* is defined.

The library (Figure 6.13) defines a *FunctionalType* to characterize the functionality a given HW process (e.g., "Camera"). A functional type provides a generic description of several pieces of functionality that a given HW process may provide. Each such piece of functionality is implemented by a specific command (i.e., *CommandInterface*) of that HW process. For instance, a "Camera" module may use a "CaptureLRV" command to record low-resolution video and a "CaptureHRV" command to record high-resolution video. We mention that these commands should be seen as generic commands of the "Camera" module, and not as commands specific to a particular implementation of it. In turn, at implementation level, each module uses specific commands (i.e., *commandImplementation*) to implement the functionality of a given generic command. For instance, the "CaptureHRV" generic command may be implemented by writing command 'a' to the

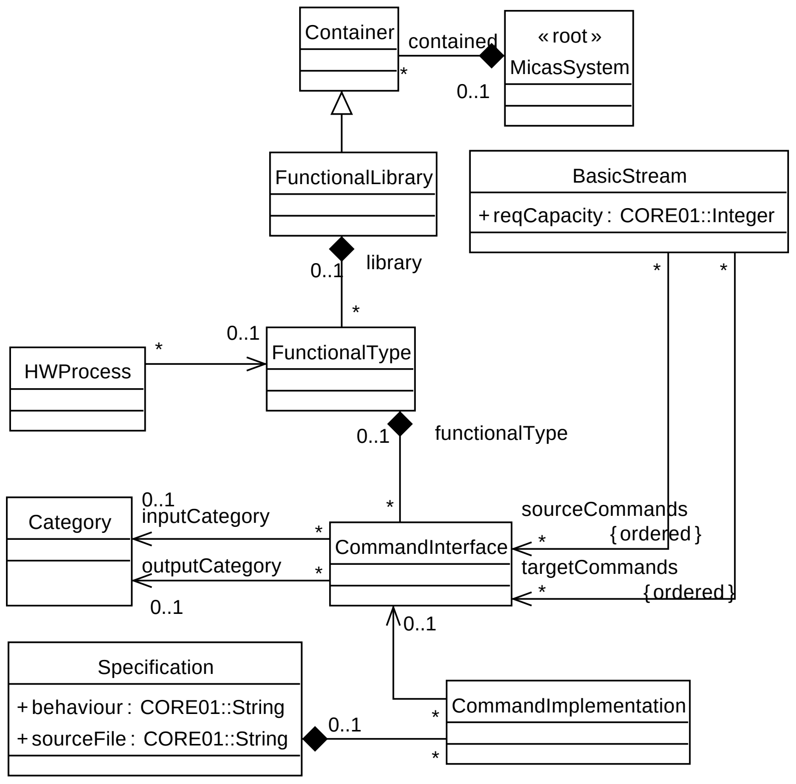


Figure 6.13: The MICAS Functional Library metamodel definition

control register of a given module implementation, whereas a different command 'b' may be required to trigger the same functionality on a different implementation of the same module.

The *FunctionalType* and the *CommandInterface* elements create a link between the control and data-flows of MICAS in the following sense: a *BasicStream* is realized in practice by one or more commands defined as *sourceCommands* and *targetCommands* respectively, issued to its HW processes by the controller. Therefore, one can create a dependency between the category of a stream and the functionality provided by the HW processes supporting the stream. This dependency will be used at design time, to select which of the *commandInterface* elements provided by the source or sink HW process, is selected to implement the stream. To support the approach, two new properties (i.e., *inputCategory* and *outputCategory*, respectively) of the *CommandInterface* element have been defined (see Figure 6.13).

6.3.4 The Overview Metamodel Definition

The definition of the overview metamodel (Figure 6.14) describes the way the domains are interconnected, providing a high-level perspective of the system. We

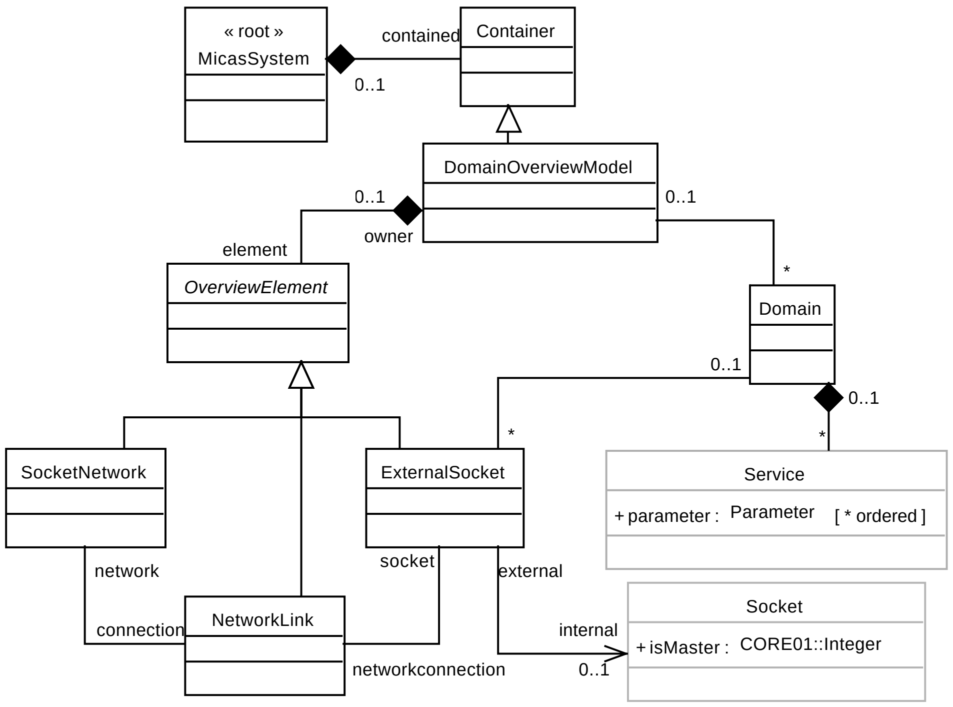


Figure 6.14: MICAS Overview metamodel definition

intentionally left this part of the metamodel to be presented last, since it encompasses the concepts and elements of both the static and dynamic perspectives.

The top level element of the overview metamodel is a container element, namely the *DomainOverviewModel*, composed of *OverviewElements*. The interface of a domain to the external environment is represented by an *ExternalSocket*, which has as a corresponding pair element a *Socket* inside that domain. We have chosen to have two different elements implementing the MICAS socket, in order to allow one to select separate implementation technologies for the internal and external communication. For instance, the implementation of the internal socket has to support the OCP bus communication inside the domain, while the implementation of the external socket deals with protocols of the external communication. domains are interconnected via *SocketNetwork* elements, which in real-life realizations may be technologies like Bluetooth, WLAN, USB, etc. *SocketNetworks* and *ExternalSockets* are connected through *NetworkLinks*.

Keeping the *Domain* element separate from the *Overview* metamodel, allows for the domain (with both its static and dynamic information) to exist in a stand alone manner. This approach enables us to organize the domains in a *Domain library* (see Figure 6.6), in order to use them thereafter for rapid creation of new architectural configurations.

6.3.5 Additional Metamodel Constraints

The MICAS metamodel enforces to a large extent the architectural constraints of the MICAS architecture. We could have enforced all the architectural constraints using abstract syntax with only a few additions to the graphical metamodel definition, but this approach would have imposed unwanted restrictions for the designer during the creation of MICAS models. Therefore, we decided to specify additional constraints separately. Several examples are given below. Some of these rules enforce architectural consistency of the platform, whereas others encode design guidelines to be used during the modeling process. In the fist category we mention:

```
Bridge connects to two DataLinks
Each ExternalSocket has a corresponding internal Socket
A SocketNetwork is connected to at least two NetworkLinks
Each Domain requires a Microcontroller
Only one RTOS per design
External to internal socket pair ``belongs'' to the same Domain
A Domain can only have one Microcontroller
BasicStreams have source and sink HW processes from the same Domain
Each HWProcess is connected by a ControlLink to the Microcontroller
```

while in the second one:

```
Each BasicStream has a Category
Each BasicStream has a bus from the same Domain
Each conceptual element has a realization
```

The presented list is not exhaustive, and is only given here in order to exemplify the approach. We also mention that the well-formedness rules are only defined at Conceptual level, from where we assume that the model transformations supporting the MICAS design process preserve the consistency of the architecture. However, nothing prevents us from adding more constraints for other abstraction levels of the metamodel and to its libraries, in case they are needed.

6.4 Modeling the MICAS Architecture

The MICAS metamodel discussed in the previous section defines a DSL for specifying the MICAS architecture. In this section, we discuss the models used by the MICAS design process at different levels of abstraction and how they are tool-supported by the MICAS Coral profile. In addition, we motivate different tool customizations intended to facilitate the design activity. We take our examples from a Digital Audio/Video Recording device case study.

6.4.1 The MICAS Tool Support

To provide tool support for the MICAS metamodel, the Coral modeling framework [102] has been used. Coral is a customizable open-source modeling tool based on the OMG standards. The main feature of Coral is that it is metamodel-independent.

Coral is built at the meta-metamodel layer (i.e., OMG's layer M3), allowing the user not only to use already defined metamodels like MOF, UML and XMI[DI], but also create new models and metamodels that can be plugged into the tool at runtime without modifying the tool itself. Metamodels are represented in Coral using the Simple Metamodel Description (SMD) language, which similarly to MOF may be seen as a metamodel for describing metamodels.

The MICAS metamodel has been defined using UML class diagrams, as we discussed in the previous section. The resulting model has been transformed into an SMD representation that allows the metamodel definition to be interpreted by Coral. Beside the metamodel implementation, a number of editors have been devised for assisting the MICAS designer. A *diagram editor* contains the set of tools, toolbars, buttons, context menus and property editors to manipulate the elements of a given metamodel (e.g., MICAS). *Property editors* (PE) play an important role, since they enable the editing of those element properties which are relevant at a given stage of the design process. The metamodel implementation and the associated editors are regarded as the *MICAS Coral profile*.

Implementing the MICAS metamodel in the Coral tool is not a contribution of the author of this thesis, therefore we omit it here. Instead, we discuss in the following, the rationale behind the design of different editors of the MICAS Coral profile, and the impact of these decisions on the design process, from the point of view of design guidelines and automation.

6.4.2 The Requirements of the Digital Video/Audio Recording Device

The *Digital Audio/Video Recording* (DAVR) system is a gadget-like personal device, used to record and play multimedia content. The device has a number of basic features (display, local memory, etc), while additional functionality can be added through plugable external devices connected over a standardized technology like Bluetooth, USB, etc.

The video is previewed on the incorporated LCD screen, which is able to display streams encoded in a specific encoding format (e.g., MPEG). Several external devices may be plugged in: an audio recoding device to record audio from the environment, a video recording device to record images, or an external storage device to increase the storage capacity of the main device. Local storage facilities are provided for the received media, and additional processing features (e.g., image processing) for the video media are supported.

Using DAVR, the user should be able to preview, record and store video captured with the incorporated camera (if an external video device is connected to the system). Additionally, video can be displayed and transferred from an external storage. If audio capabilities are available, the user can record audio files and store them in the local memory of the device. Furthermore, when both the audio and video recording facilities are available, the video is recorded and combined with sound.

6.4.3 The Overview Model

The overview models of the MICAS system provide a general view of what domains exist in given configuration and what functionality they provide. According to its metamodel definition, each *Domain* element encompasses both static and dynamic information. This means not only that the designer can access this information from the overview model, but also that one should be able to select the perspective to focus on, at a given stage of the design process.

Combining Services into Scenarios

Services are the most important element of the MICAS architecture in rapport with the application residing at the RTOS level, because they provide the *programming interface* of the architecture. Therefore, it is important how these services are specified, on what level of granularity and how easily they can be used by the designer to implement the application.

From the perspective of the application, it is important that the programming interface is specified in as simple terms as possible, in order to facilitate an easy selection of the required services. For instance, a service like *CaptureVideo* should encompass all the functionality of the platform regarding video recording by the Video domain, while a *PreviewVideo* service of a different domain should encompass the functionality of that domain for displaying video streams. To create an application level scenario, like *PreviewVideo*, which uses the former service to capture and the latter to display video, the *CaptureVideo* and *DisplayVideo* services have to be combined in an orderly fashion.

Several aspects arise from this approach. Firstly, a given MICAS configuration will present to the application all the services existent in the system and the application will have to combine them in a consistent manner. For instance, starting the *CaptureAudio* and the *DisplayVideo* services would not provide a valid scenario, as the two services are incompatible. Secondly, one service (e.g., *DisplayVideo*) may have multiple variations, based both on the source and on the type of the video stream to be displayed. This means that a number of parameters are needed to specify the variations of the service, when it is invoked. One example could be to request the *DisplayVideo* service from a given domain, which provides a stream encoded in MPEG format according to a number of encoding parameters like resolution and frames per second. Having a number of generic parameters to characterize a service is not trivial, since each service in part can require completely unrelated sets of parameters. Especially for configurations with a large number of services, this approach can prove difficult. The third aspect is that, especially in the area of programmable architectures, it is more important to have highly customized

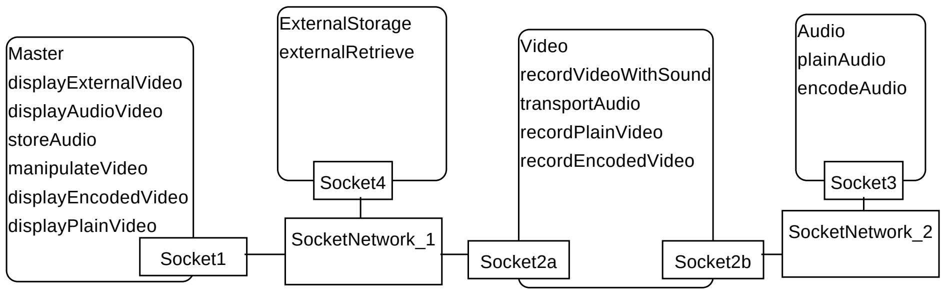


Figure 6.15: Overview model of the DAVR system

solutions for different application families, which offer an optimal implementation, rather than a generic solution.

Based on these aspects, we have decided that a fair approach is for the designer to manually create different variations of the same service and to provide them as scenarios at RTOS level, leaving to the mapping process the responsibility of choosing the right variation of the service instead. The MICAS metamodel definition presented in Section 6.3 supports this approach by defining a *subservice* property, which represents a remote service used by a local service. Following this approach, we promote the services of the master domain as scenarios of the entire system and the services provided by the other domains become subservices of the master services.

There are four domains in the DAVR system (Figure 6.15), as resulted from the case study specification. The *Master* domain provides services to store data, to preview video and to process images. The *Master* domain is connected to two slave domains *ExternalStorage* and *Video*, respectively. The *ExternalStorage* domain provides services for retrieving and storing data. In the real world, this domain may be any kind of external storage device like memory sticks, memory cards or even hard disk drives. The *Video* domain, has capabilities to record video, possibly in an encoded format, and to provide it to the neighboring domains. The fourth domain, *Audio*, has the capability to record audio from the environment and to provide it to its connected domains, possibly in an encoded format.

It is worth noting that the *Video* domain has a slave role relative to the *Master* domain and a master role relative to the *Audio* domain. In the same time, the *Video* and the *ExternalStorage* domains have a slave relationship to the *Master*, but even if they are interconnected by the same network, they cannot invoke directly each other's services. The master-slave relationship between domains is not explicitly shown in the diagram, yet it can be easily inferred.

Analyzing the functionality of each domain, the following services have been identified. The *Audio* domain captures audio and provides it to the environment, both in the WAV and MP3 encoding. This functionality may be seen as two distinct services for capturing audio: *plainAudio* and *encodedAudio*, respectively. The main functionality of the *Video* domain is to provide video to the DAVR device, possibly combined with audio. As such, we obtain the following services: *CapturePlainVideo* captures video and provides it to the environment in the camera supported format (e.g., MPEG), *CaptureEncodedVideo* captures video initially in the camera supported format, encodes it into a given format (i.e., AVI) and provides it to the environment; *CaptureVideowithSound* captures video and combines it with audio streams from a connected domain, encodes both streams in a compressed format (e.g., AVI) and provides the result to the environment. The *transportAudio* service is only used by the *Video* domain in order to transport audio from a remote slave domain to a master domain without any processing. The *ExternalStorage* domain provides only one service, *externalRetreive*, for retrieving data stored in this domain. The *Master* domain has the largest number of services:

- *displayPlainVideo* – displays MPEG video received from the *Video* domain;
- *displayEncodedVideo* – decodes AVI video received also from the *Video* domain and displays it on the incorporated display;
- *displayExternalVideo* – displays video received from the *ExternalStorage* domain;
- *storeAudio* – stores audio streams received from the *Audio* domain in a local memory or file system;
- *manipulateVideo* – retrieves a video file from the local memory, processes it and saves it back to the same local memory or file system.

We acknowledge that our approach to define services does not scale well for applications requiring large number of services, but this is not an impediment, since we are targeting applications that require a relative small number of services. A more elaborated service definition in the metamodel would enable us to automatically create variations of these services using their properties, like source, destination, type, transfer rate, etc. Such an approach would require the definition of a more complex framework for specifying streams and implicitly, services of the MICAS architecture, that has to be further investigated.

As a final observation, one can argue that the *transportAudio* in the *Video* domain is not a real service, but rather a hidden intrinsic functionality of the *Video* domain. If we chose not to specify this service, the RTOS application will have to program this transfer anyway by individually triggering the required domain components. Even more, a general guideline for the designer can be that for all the domains playing both master and slave roles and therefore, being connected to many socket networks, one has to automatically define transport services for all the stream types provided by the slave domains.

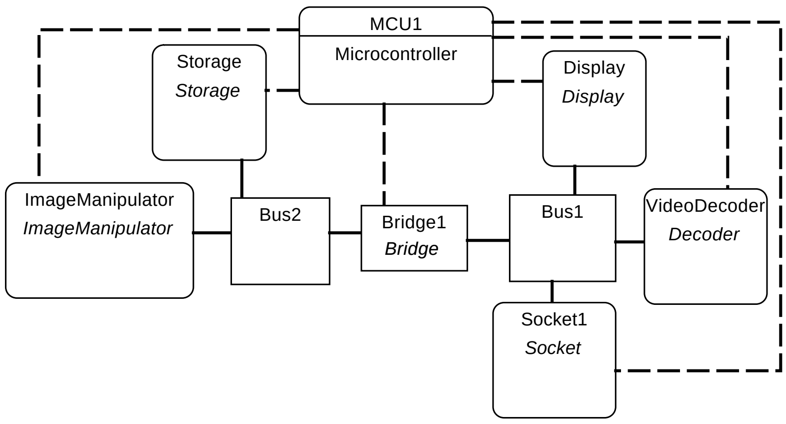


Figure 6.16: Conceptual model of the Master domain

6.4.4 MICAS Conceptual Models

The conceptual model of a domain depicts the MICAS resources from the point of view of their functionality, without focusing on the underlying hardware communication infrastructure. The conceptual model is obtained from the service list of each domain, in the *Configure Conceptual Model* step of the MICAS design process (Figure 6.6). During this step, the designer not only has to populate the domains with conceptual components for providing the required processing capabilities to support the services, but also to interconnect these components using buses. A dedicated diagram editor has been implemented in Coral, in order to support the editing of the MICAS conceptual models. The conceptual models discussed in the following are designed using the MICAS Static diagram editor of Coral.

Figure 6.16 presents the conceptual model of the *Master* domain of DAVR. To display video streams a *Display* module is used. The video stream can be encoded either in a format that is supported by the *Display* module, or a *Decoder* module is required to convert the video stream. In addition, a socket component, namely *Socket1*, intermediates the communication with the external environment. For performance reasons, the *Display*, *Decoder* and *Socket1* HW processes are collocated in the same cluster (i.e., are connected by the same bus *Bus1*). An *ImageManipulator* and a *Storage* modules are included, in order to provide image processing and storage capabilities, respectively. Since these two HW processes handle lower priority tasks, they are situated in a different cluster and interconnected by the *Bus2* component. Finally, a bridge component, namely *Bridge1*, interconnects the two clusters of the domain.

The *ExternalStorage* domain (Figure 6.17) contains only two HW processes: a *MemoryManager* module for storing data and, a socket (*Socket4*) to intermediate the communication with the environment. The two components are interconnected by the *Bus1* component.

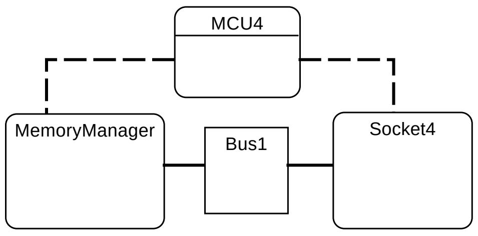


Figure 6.17: Conceptual model of the ExternalStorage domain

The *Video* domain (Figure 6.18-left) presents a couple of interesting aspects. Firstly, since it is connected to two different socket networks, it requires two different socket components, *Socket2a* and *Socket2b*, respectively. Secondly, this domain has to capture image through its *Camera* module and to eventually encode it into a desired format using the *Encoder*. The main reason for encoding the video streams, or other stream types in the system, is that, typically, the off-chip buses interconnecting the domains support lower transfer rates as compared to the on-chip buses of a domain. In order to support the transfer rates imposed by the application requirements, the interdomain communication has to be encoded using a compressed format. Thirdly, this domain not only receives audio streams from the *Audio* domain, but also has to forward these streams to the *Master* domain, or even more, to encode audio streams along with video ones into a combined audio/video stream.

Finally, the *Audio* domain (Figure 6.18-right) records sound from the environment, through the *SoundRecording* component, and provides it to its master domains. For similar reasons as in the *Video* domain, the audio streams may be encoded into a compressed format using the *AudioEncoder* module. A bus is used to interconnect all the HW processes of the domain to *Socket3*.

As one may note, some of the elements in Figure 6.16 have an additional string (in *italic* font) in their graphical representation, as compared to the other conceptual models of DAVR. This string depicts the functional type of each element. Currently, the association of functional types to a conceptual model is strictly based on the designer's experience. The main purpose of this activity is the identification, in the subsequent phases of the design, of the microcommands that each implementation of a HW process provides.

A Coral Editor for the MICAS Static Diagrams

Several customizations of the Coral editor have been performed, in order to facilitate the design activity. For instance, since the metamodel specifies the exact type of link between the components of a domain, when a connection is drawn between any of the components, it is automatically specialized into either a *DataLink* or a *ControlLink*, based on the type of the connected components. Another example is

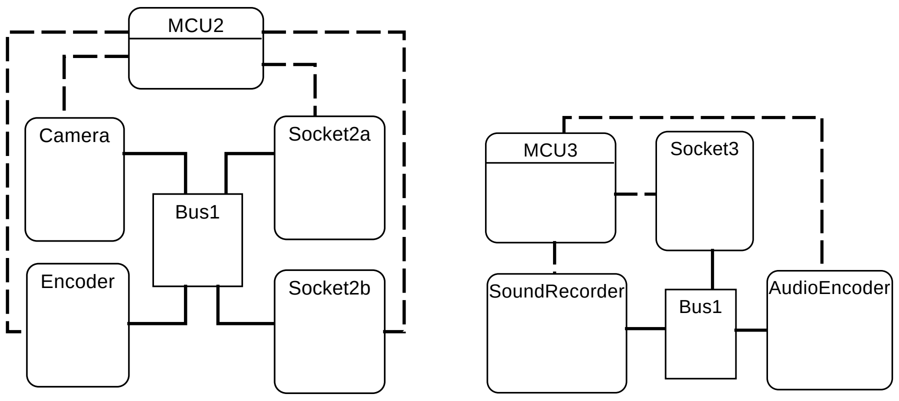


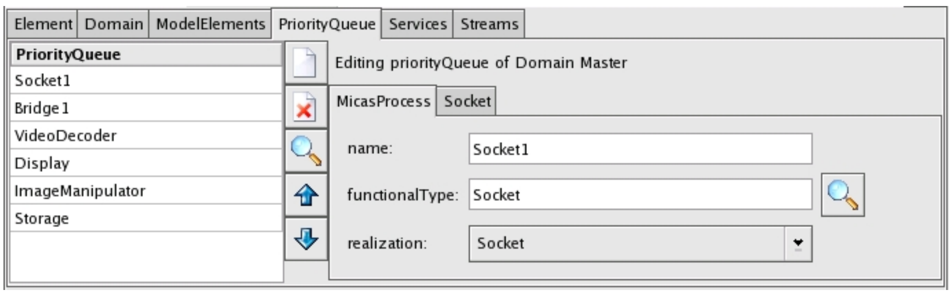
Figure 6.18: Conceptual models of the Video (left) and Audio (right) domains

the one of adding the functional types to the conceptual elements. Taking benefit from the Coral tool features, the designer can simply select a functional type from the Functional library and then drop it onto the diagrammatic representation of a given conceptual element. As a result of this action, the chosen functional type is automatically assigned to the conceptual element in question.

Other design aids may be envisioned here, but they have not been implemented yet. For instance, another aid could be provided to the designer in case of connecting a domain *microcontroller* to the HW processes. Since all HW processes in the domain have to be connected to the *microcontroller*, the editor could automatically create the *ControlLinks* for each newly added element. Moreover, in order not to clutter the design with too many graphical elements, the designer could have the option of hiding or showing these "default" connections.

One provided design aid is the possibility to edit the priority of different HW processes. The priority of a HW process is encoded and stored in the metamodel (Figure 6.7), via the *priorityQueue* property of the *Domain*. Whenever a new process is added to a conceptual model, it is assigned a position in the priority queue. The priority of a HW process will reflect, in the subsequent design steps, in the order in which the IRQ numbers are assigned to HW processes. We have customized the MICAS Static diagram editor with two new features:

- a back-end processing script has been implemented for monitoring the addition of new elements to the queue, such that socket components are always assigned the highest priority (i.e., lowest IRQ numbers). The approach relies on the fact that sockets handle critical communication between the domains and their controllers.
- in addition, a PE (Figure 6.19) has been devised for enabling the designer to manually change the order of the elements in the property queue.



Editing priorityQueue of Domain Master

name: Socket1

functionalType: Socket

realization: Socket

Figure 6.19: Caption of the Priority Queue property editor

6.4.5 MICAS Detailed Models

In the following, we discuss the *Create Detailed Domain* step of the MICAS design process (Figure 6.6). At this step, conceptual components are transformed into detailed components, and the communication between them is refined to use specific communication mechanism. The transformation is supported by the MICAS Realization library, which encodes patterns to refine this communication. These refinements may be seen as design decisions, taken for architecting the system.

The MICAS architecture has, by default, four predefined realizations (stored in the Realization library) to transform conceptual components into detailed components. Other alternative realizations can be defined in case other communication mechanisms are used by the designer. Although, the realizations are part of the MICAS architecture definition, and consequently they are not a contribution of our work, we introduce them at this point of discussion, in order to give a better understanding of their use.

The first realization (Figure 6.20), encodes the bi-directional communication between controllers and HW processes, like modules and bridges. In this case, bridges and modules communicate with the controller through interrupt requests, whereas the controller sends commands to HW processes via a control bus. The latter communication is supported by the *sfr_bridge* and *sfr_reg* components, respectively.

The realization in Figure 6.21 depicts the communication mechanism between the RTOS and the master controller. An *AHB_bus* component is used for writing data to a dual-port memory (DPRAM) and the controller is notified through an interrupt mechanism. The controller sends data back by writing into the *DPRAM* component, while the RTOS reads this data through the *AHB_bus*.

The third and the most complex realization (Figure 6.22), specifies the communication mechanisms between the controller and the sockets. The mechanism is similar to the one in the controller-process communication, except that the conceptual socket element generates an additional component *socket_ctrl-*_sfr_reg, which intermediates the control communication between controllers. This additional element is also connected to the controller through an interrupt based mechanism in one direction and a control bus in the other direction.

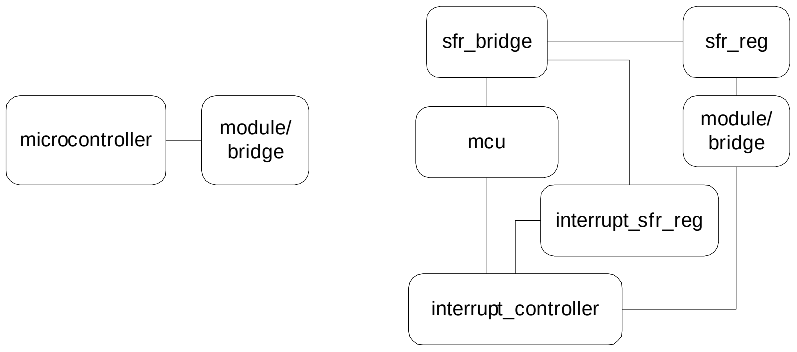


Figure 6.20: The microcontroller-process communication realization

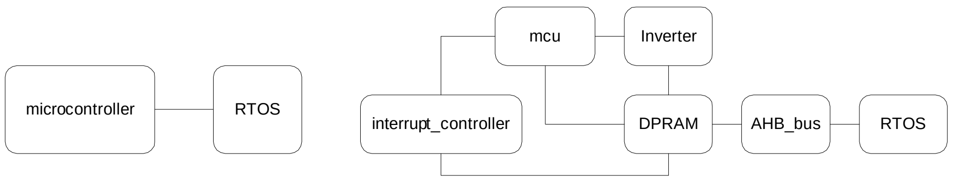


Figure 6.21: The microcontroller-RTOS communication realization

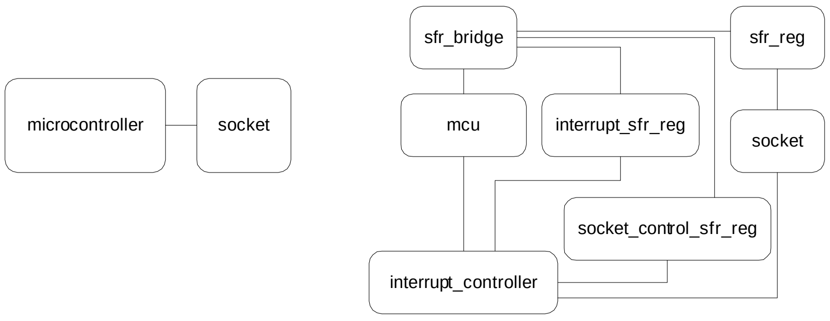


Figure 6.22: The microcontroller-socket communication realization

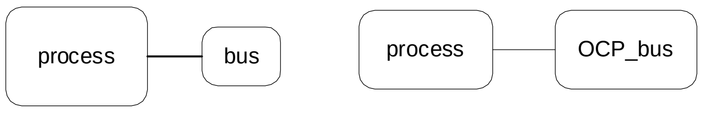


Figure 6.23: The inter-process communication realization

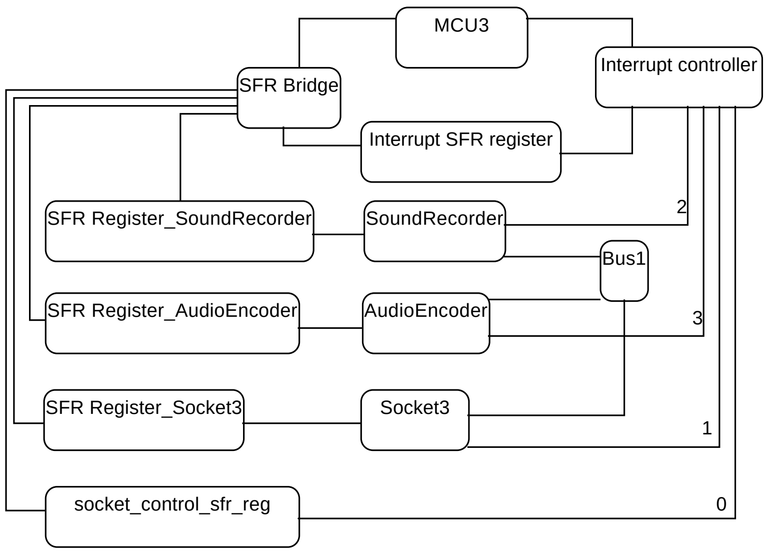


Figure 6.24: Detailed model of the Audio domain

Finally, the fourth realization (Figure 6.23) specifies the communication between different HW processes connected in our case by an OCP bus.

Figure 6.24 presents the detailed model of the *Audio* domain obtained after applying these realizations to the conceptual model in Figure 6.18-right.

Taking advantage of the fact that detailed elements (i.e., *Link* and *Detailed-Component*) are specializations of the *ModelElement*, we have been able to reuse the already implemented MICAS Static diagram editor for displaying detailed models, thus no additional diagram editors had to be implemented.

In the current version of MICAS, there is only one combination of realizations used for creating detailed models, as discussed above. Therefore, having only one possible design choice, this transformation is easy to automate. In case other realizations are added to the library, and consequently more than one design decision is available, the user intervention will be required.

6.4.6 MICAS Implementation Models

The elements included in a detailed model do not represent concrete implementation components (i.e., IP-blocks), but rather abstract types of the implementation components. At the *Select Domain Implementation* step of the MICAS design process, the designer has to decide upon a specific implementation for each detailed component by assigning it a corresponding element from the MICAS Implementation library. Figure 6.25 shows the result of the *Select Domain Implementation* step, applied to the detailed model of the *Audio* domain.

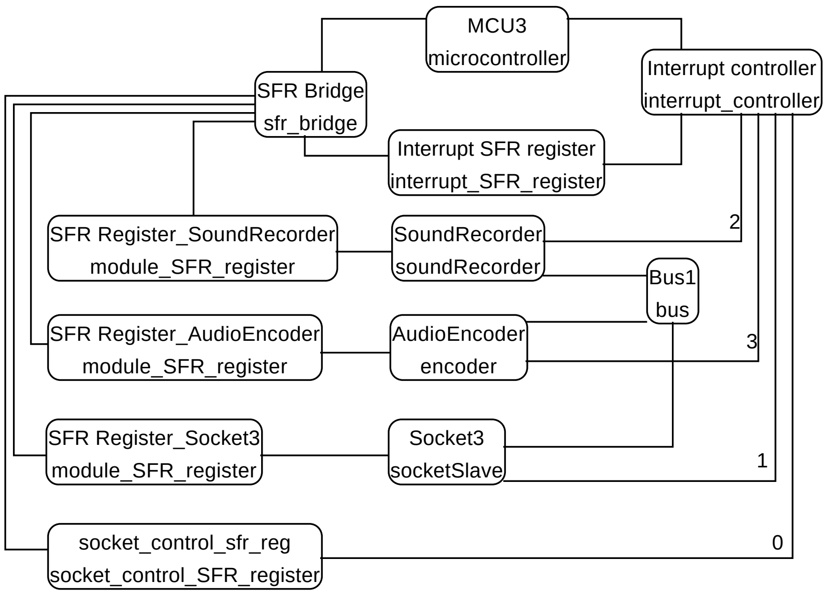


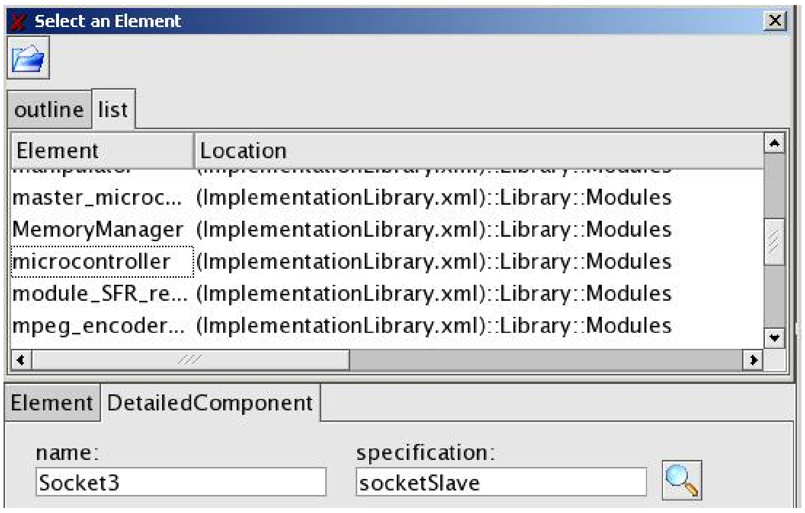
Figure 6.25: Detailed model of the Audio domain marked for implementation

The approach is similar to the "PIM marking" process of MDA and hence, difficult to automate. Yet, several design aids can be provided. For instance, the Static diagram editor has been customized to support drag-and-drop operations over elements, based on which *Specification* elements from the Implementation library can be assigned to detailed elements. Alternatively, the designer can use the associated PE (Figure 6.26) for manually selecting the required *Specifications* from the library.

Other design aids may be envisioned. For instance, we can see the elements present in a MICAS detailed model as falling into two groups. The first group contains generic elements that support the MICAS communication infrastructure in general, but they do not contribute to the functionality of the system (e.g., control buses, registers, etc.). The other group includes detailed versions of the conceptual (functional) components (e.g, HW process). To facilitate the designer's work, the editor can be customized such that when an implementation is selected for a given detailed component in the first group, all the other elements of the same type are automatically assigned the same implementation.

6.4.7 The Dynamic Models of MICAS

This section discusses the process of modeling the dynamic perspective of a MICAS configuration, perspective which is also designated as the MICAS programming model. As discussed in the previous section, each domain provides a pro-



Element	Location
master_microc...	(ImplementationLibrary.xml)::Library::Modules
MemoryManager	(ImplementationLibrary.xml)::Library::Modules
microcontroller	(ImplementationLibrary.xml)::Library::Modules
module_SFR_re...	(ImplementationLibrary.xml)::Library::Modules
mpeg_encoder...	(ImplementationLibrary.xml)::Library::Modules

name: Socket3
specification: socketSlave

Figure 6.26: A property editor for selecting implementations for Detailed Components

gramming interface (i.e., services) and, in turn, each service is specified in terms of streams between HW processes.

The modeling tool of the dynamic perspective is represented by the *stream diagram*. A *stream diagram* provides a snapshot of the dynamic aspects of a domain. The diagram abstracts away physical/static details of the MICAS architecture hence, allowing the designer to focus only on the streams between HW processes. There are only two graphical element types represented in this diagram: *HW processes* and *basic streams* interconnecting them. Ideally, and as a design guideline, each stream diagram is used for designing the streams of only one service, but nothing prevents the designer to include all the streams of all the services in the same diagram. There may be several stream diagrams created for the same domain and even sharing the same HW processes and streams. We have promoted this approach in order to reduce the complexity of the design, allowing the designer to work on less cluttered models. As one may notice, the stream diagram is quite similar to a data-flow diagram as proposed by the structured methods. The main difference is that the elements in this diagram have a richer list of properties, as defined by the MICAS metamodel. For instance, a basic stream is not only characterized by a data type (i.e., category), but also by a required capacity and a physical bus supporting the stream.

As an example, we take the *CaptureVideowithSound* service of the *Video* domain, and explain the creation of its data-flow perspective. This service captures video using the *Camera* module, and audio using an external audio device (in our case the slave domain *Audio*). Both streams are combined and encoded together in a given format (i.e., AVI). Four basic streams have been defined (Figure 6.27),

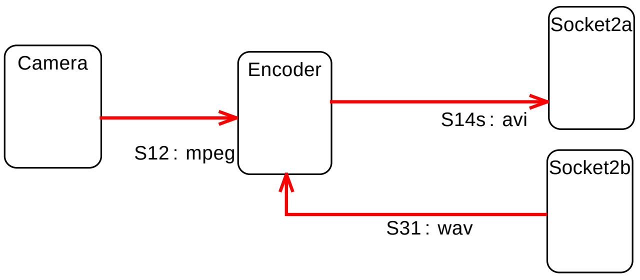


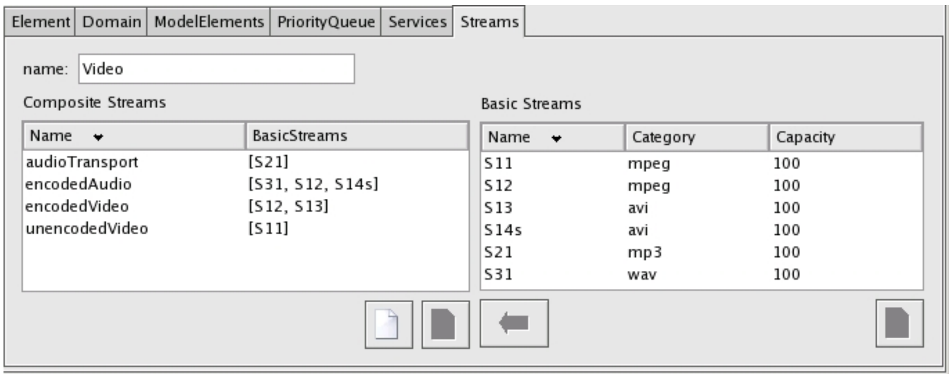
Figure 6.27: Streams of the *CaptureVideowithSound* service

for implementing this service. The stream *S12* is in charge of transporting data from the *Camera* module to the *Encoder*, encoded in the MPEG format. The *Encoder* receives a stream *S31* transporting audio (encoded in WAV format) from the *Socket2b* socket, which intermediates the communication with the *Audio* domain. The *Encoder* combines the video and the audio streams into a single stream *S14s*, which is output into the AVI format to *Socket2a*, in order to be sent further to the *Master* domain.

The Stream Diagram Editor in Coral

A *stream diagram editor* is supporting the service design process in Coral. The editor allows the designer to graphically create new or to add HW processes from existing conceptual models, and furthermore, to interconnect them through either new or already defined basic streams.

We have devised two custom PEs to assist in editing service and stream properties. The *Streams PE* (Figure 6.28) displays all the basic streams of a domain. Whenever a new basic stream is graphically created, it is automatically added to



name: Video

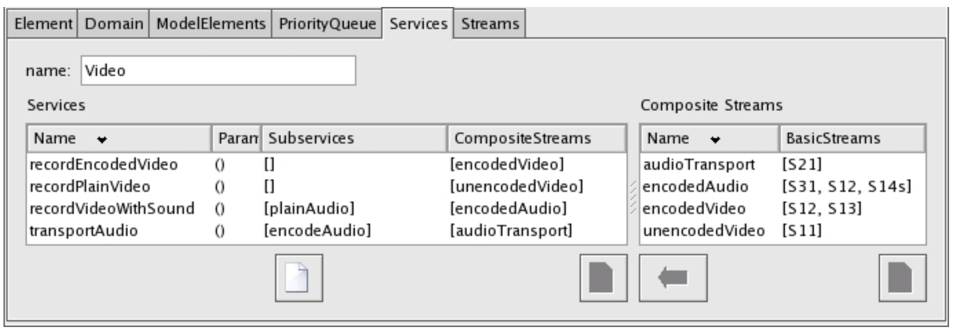
Composite Streams

Name	BasicStreams
audioTransport	[S21]
encodedAudio	[S31, S12, S14s]
encodedVideo	[S12, S13]
unencodedVideo	[S11]

Basic Streams

Name	Category	Capacity
S11	mpeg	100
S12	mpeg	100
S13	avi	100
S14s	avi	100
S21	mp3	100
S31	wav	100

Figure 6.28: Caption of the Streams property editor



name: Video

Services

Name	Paran	Subservices	CompositeStreams
recordEncodedVideo	()	[]	[encodedVideo]
recordPlainVideo	()	[]	[unencodedVideo]
recordVideoWithSound	()	[plainAudio]	[encodedAudio]
transportAudio	()	[encodeAudio]	[audioTransport]

Composite Streams

Name	BasicStreams
audioTransport	[S21]
encodedAudio	[S31, S12, S14s]
encodedVideo	[S12, S13]
unencodedVideo	[S11]

Figure 6.29: Caption of the Services property editor

the right-hand side panel of the PE, where stream properties, like *Category* and *Capacity* can be edited. Composite streams (e.g., *encodedAudio*) are obtained by grouping together several basic streams (e.g., *S31*, *S12*, *S14s*). We have decided that a graphical representation of the composite streams is not necessary, as it will not significantly improve the understandability of the model without cluttering it. Instead, composite streams can be created and populated with basic streams using the left-hand side panel of the Streams PE, where basic streams can be added to composite streams by simply dragging them from the right-hand side panel onto the *BasicStreams* field of each composite stream. It is also important to remark that due to the way the MICAS metamodel has been defined allows for the same basic stream to be included in several composite streams.

As defined by the MICAS metamodel, a composite stream represents a possible implementation of a MICAS service. Optionally, several composite streams can be assigned to a service, and the one to be used at run-time is selected based on certain heuristics. In addition, a *Services PE* (Figure 6.29) has been devised for managing the services and the composite streams of a domain. There are two panels in this PE. The right-hand side panel provides the list of composite streams existing in a domain and their basic streams. The left-hand side panel enables one to create new services by selecting different composite streams as alternatives for their implementation. The selection of a remote service to be used as subservice is also supported.

In addition, following the metamodel definition, it is possible to characterize a service based on a *Parameter*, that would enable an automated selection of a corresponding composite stream. For the moment, this approach is not implemented. Nevertheless, the *Parameter* field can be used for eventually documenting the service (e.g., capacity, category, textual description, etc.).

Using Services to Program HW Process Functionality

A MICAS service models a complex processing task of a given domain. This processing task is supported by several HW processes, which implement and process data streams between themselves. As such, a service may be regarded from two perspectives: one that depicts the basic streams between HW processes implementing the service – the *data-flow perspective*, and one that provides the list of commands used to configure the HW processes for supporting these streams – the *control perspective*.

Thus, one can infer the list of HW process commands required to implement the service from the basic streams of that service. We recall that a basic stream transports data between two HW processes, of which one is master and the other slave, with respect to that stream. We also recall that only master HW processes are controlled by the controller, whereas slave processes are controlled by their master HW process. Therefore, in order to implement a basic stream between two processes, only the master process has to be programmed by the controller. A HW process may support several processing tasks, according to its functional type (e.g., Camera, Display, etc.). We remind the reader that the functional type of a HW process is abstracted by the *FunctionalType* metamodel element. In turn, each processing task of a HW process has a corresponding HW process command, generically specified by the *commandInterface* element.

This approach enables the service designer to manually select from the commands of its master HW process the generic command that implements a given basic stream. However, our MICAS metamodel definition does not differentiate, at conceptual level, between master and slave HW processes, which is implementation-dependent information. Two alternatives could have been followed: a) to provide the designer with a list of HW process commands (i.e., *CommandInterfaces*) for both the master and the slave processes and she/he decides which commands to select; b) to allow the designer to specify, already at conceptual level, which of the HW processes have a master or a slave role relative to a physical bus. This would have meant a specialization, in the metamodel definition, of the *DataLink* element into a *masterDataLink* and a *slaveDataLink*, respectively. We have decided to follow the former approach, since the MICAS metamodel and the current implementation of the MICAS Coral profile did not require any modifications. We consider that the latter approach may be left for future versions of the project. But if such an approach is followed, the specialization of the *dataLink* would have allowed the user to select the desired HW process commands only from the ones of the master HW process.

As an example, we present the service *encodeAudio* of the *Audio* domain. The service is implemented by the composite stream presented in Figure 6.30. There are two basic streams implementing the data-flow perspective of the service, *S12* and *S13*, respectively. The *SoundRecorder* has the functional type *SoundRecorder*, providing two commands: *record_sound_rawa* and *record_sound_wav*, which record sound, in either RAWA or WAV sound format, respectively. The *AudioEncoder* belongs to the functional class *Encoder*, and provides the following commands: *encode_avi_2_mpg*, *encode_mpg_2_avi*, *encode_rawa_2_mp3* and *encode_wav_2_mp3*. We mention that although the *AudioEncoder* module is required to process audio

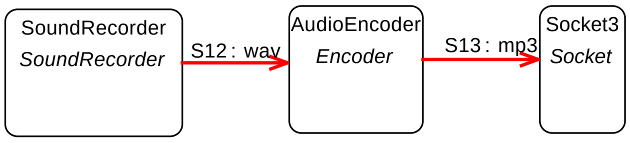


Figure 6.30: Composite stream implementing the *EncodeAudio* service

streams in the *Audio* domain, it belongs to a wider functional class (i.e., *Encoder*) that can also process video streams. Finally, the *Socket3* HW process provides two commands *transmit_data_to_domain* (to send data over the socket network), and *send_data* (to dispatch data inside the domain).

In order to create the control perspective of this service, the designer has to select the HW process commands implementing each stream, that is, *record_sound-_wav* for stream *S12* and *encode_wav_2_mp3* for stream *S13*. Based on the direction of the stream, the sockets have to be programmed also for sending or receiving data. In our example, this implies that *transmit_data_to_domain* command has to be used. Since in the current version of the MICAS dynamic metamodel, we do not model streams outside the domains, we have decided that in the specific case of the streams originating or entering sockets, to allow a second command to be attached to the streams. For instance, stream *S13*, beside having attached a command for encoding data, will also have the command for sending that data through the socket (i.e., *transmit_data_to_domain*). This approach is motivated by the fact that sockets may be seen as "masters" of the external networks, where they are initiating transfers to remote domains (i.e., "slave" sockets).

Attaching commands of the HW processes to streams is a manual task based on designer's experience, but certain aids and guidelines may be provided. A HW process, and in particular a MICAS module, basically transforms input data into output data. Consequently, a HW process command may be characterized by the type (i.e., category) of its input and output data. For instance, we can consider that a command corresponding to encoding MPEG video streams into AVI video streams can be regarded under the generic name of "MPEG2AVI". Two new properties could have been added to the *CommandInterface* meta-element, namely the *acceptedCategory* and *providedCategory*, respectively, in order to provide a more comprehensive description of each command. Their implementation in the metamodel could be realized by simply adding two associations between the *Command-Interface* and the *Category* meta-elements. The approach would allow, when selecting commands for a given stream, to filter out the commands that are not able to accept as input or to provide as output the corresponding stream category.

Once having all the basic streams specified in terms of commands, the control perspective of the service is easily obtained, by replacing each basic stream with the corresponding *setup* (micro)commands. A *setup* microcommand uses three parameters to implement the stream:

- *m_address* depicts the control register address of the master HW process that implements the stream;
- *m_command* represents the command which triggers the processing task that the master HW process has to execute;
- *s_address* points to the bus address of the slave HW process, to/from which the processed data is sent/received.

The result of transforming the composite stream depicted in Figure 6.30 into controller microcommands is shown below:

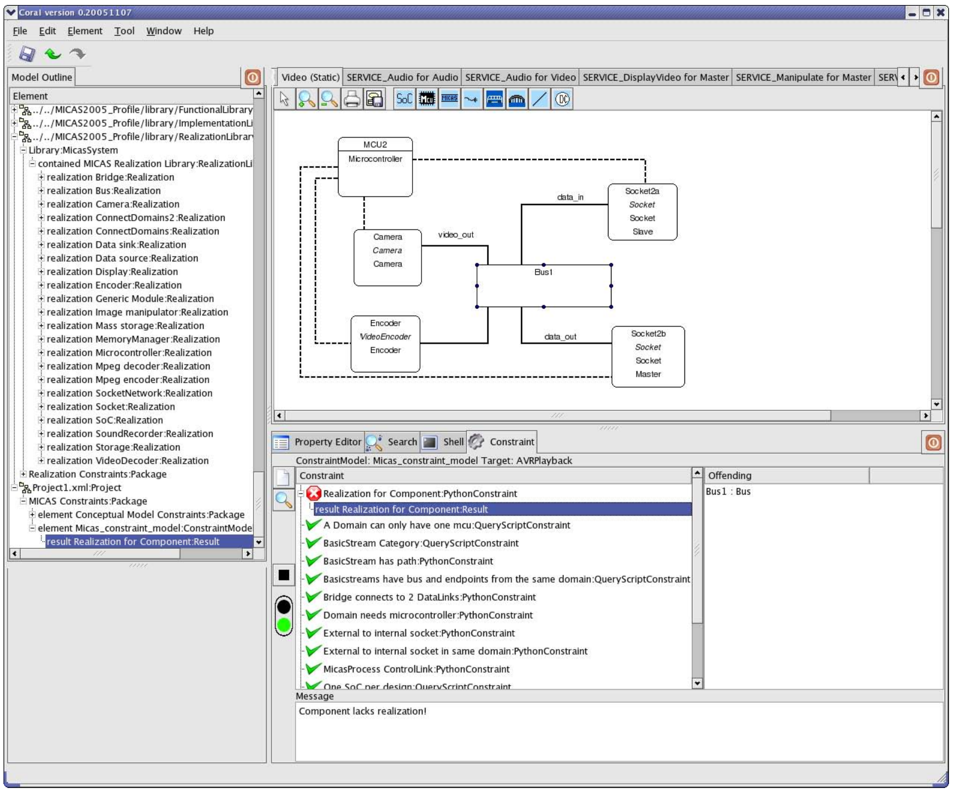
```
setup(m_addr_sr, record_sound_wav, s_addr_enc);
setup(m_addr_enc, encode_wav_2_mp3; s_addr_sck);
setup(m_addr_sck, transmit_data_to_domain, 0);
```

We mention again that, at this point, we are talking about generic commands (i.e., *commandInterfaces*) provided by different types of HW processes (i.e., *FunctionalTypes*) and, not about real implementation commands. When a implementation component is selected for a given HW process, a corresponding *commandImplementation* is assigned to the *CommandInterface*. For instance, at the implementation level, the *encode_wav_2_mp3* generic command is implemented as value "2" written to the control register of the *Encoder* module.

6.4.8 Enforcing Architectural Consistency

Tool support for checking the well-formedness rules associated to the MICAS metamodel is provided via the CQUERY plug-in of Coral [80]. The plug-in allows one to verify model constraints (as the ones we have discussed in Section 6.3.5) on-the-fly, without interfering with the model creation process. An example of constraint checking is presented in Figure 6.31. The constraint editor is placed at the bottom of the screen and it displays the status of all constraints associated to the model. The constrains that are met are displayed in green, whereas the ones that are violated are displayed in red. In the above mentioned example one constraint is violated, namely "Each conceptual component has a realization". The constraint editor displays the list of elements that are violating a given constraint (in our case the *Bus1* element) on the right-hand side panel. The implementation of the MICAS constraints in the CQUERY plug-in is a contribution of the author of this study and thus, it is not discussed here.

Several design aids are provided to the designer by using the MICAS consistency rules. For instance, a basic stream transports data over a physical bus characterized by a capacity. The basic stream also is characterized by a required capacity, which represents the minimum transfer rate the stream requires for transferring data over the bus. Therefore, it is important that, when new basic streams are added to a domain, the capacity provided by the underlying hardware resources is verified. If the capacity required by the stream exceeds the capacity provided by the bus the designer is notified. We will discuss in Chapter 7 how the controllers



```
Element
+ ../../MICAS2005_Profile/library/FunctionalLibrary
+ ../../MICAS2005_Profile/library/ImplementationLi
- ../../MICAS2005_Profile/library/RealizationLibrar
  - Library:MicasSystem
    - contained MICAS Realization Library:RealizationLi
      + realization Bridge:Realization
      + realization Bus:Realization
      + realization Camera:Realization
      + realization ConnectDomains2:Realization
      + realization ConnectDomains:Realization
      + realization Data sink:Realization
      + realization Data source:Realization
      + realization Display:Realization
      + realization Encoder:Realization
      + realization Generic Module:Realization
      + realization Image manipulator:Realization
      + realization Mass storage:Realization
      + realization MemoryManager:Realization
      + realization Microcontroller:Realization
      + realization Mpeg decoder:Realization
      + realization Mpeg encoder:Realization
      + realization SocketNetwork:Realization
      + realization Socket:Realization
      + realization SoC:Realization
      + realization SoundRecorder:Realization
      + realization Storage:Realization
      + realization VideoDecoder:Realization
  + Realization Constraints:Package
- Project1.xml:Project
  - MICAS Constraints:Package
    + element Conceptual Model Constraints:Package
    - element Micas_constraint_model:ConstraintMode
      result Realization for Component:Result
```

ConstraintModel: Micas_constraint_model Target: AVRPlayback

Constraint	Offending
✗ Realization for Component:PythonConstraint	Bus1 : Bus
result Realization for Component:Result	
✓ A Domain can only have one mcu:QueryScriptConstraint	
✓ BasicStream Category:QueryScriptConstraint	
✓ BasicStream has path:PythonConstraint	
✓ Basicstreams have bus and endpoints from the same domain:QueryScriptConstraint	
✓ Bridge connects to 2 DataLinks:PythonConstraint	
✓ Domain needs microcontroller:PythonConstraint	
✓ External to internal socket:PythonConstraint	
✓ External to internal socket in same domain:PythonConstraint	
✓ MicasProcess ControlLink:PythonConstraint	
✓ One SoC per design:QueryScriptConstraint	

Message
Component lacks realization!

Figure 6.31: MICAS constraint checking example in Coral using CQUERY

manage the allocation of several streams, at the same time, over the same bus. The required capacity is not a characteristic of the system, but a requirement of the design (i.e., a goal that has to be fulfilled). The required capacity is computed manually based on experience of the designer and on the application specification.

Another design aid that can be provided by taking advantage of the well-formedness rules of the MICAS metamodel, is ensuring that the category of a basic stream belongs to the sets of provided and accepted categories of the stream's source and respectively, sink modules. Based on the MICAS metamodel definition, each module can process and provide a number of categories, specified in the metamodel as *providedCategory* and *acceptedCategory*, respectively. Basic streams originate and are targeted to HW processes and, their data type is described by a *Category*. Therefore, it is important for the designer to be notified by the tool, if the category of a basic stream is not supported by its processes, at the stream design time.

6.5 MICAS Code Generation

Once the design of a MICAS configuration is completed, the resulting specification (i.e., containing both the static and dynamic perspectives) has to be transformed into a SystemC executable specification to enable the simulation of the configuration. The MICAS specification, which is obtained from the MICAS implementation phase, encompasses two categories of information:

- *structural configuration* – depicting the structure of the system in terms of hardware components and their interconnections at port level;
- *functional configuration* – information encompassing both a static and a dynamic perspective of a given MICAS configuration, other than the one regarding the hardware structure of the system. The *static functional configuration* specifies non-hardware properties (like address spaces, IRQ numbers, etc.) of different MICAS hardware components. The *dynamic functional configuration* specifies the programming interface (i.e., services) and its implementation (i.e., streams) specific to each domain.

The structural configuration of MICAS is transformed (i.e., generated) into concepts of the SystemC language by mapping the MICAS implementation components into SystemC concepts. In change, there is no direct mapping between the elements of the functional configuration and the concepts of SystemC. Nevertheless, both the structural and the functional configuration have to be taken into consideration during the SystemC simulation. The functional configuration of each domain is intended to be used, at run-time, by controllers for driving the components of their domains. Since SystemC is an extension of C++, the easiest way to integrate this information with the simulation model, is to represent the functional configuration in terms of C++ language constructs. The approach also facilitates the definition of reusable model transformation for generating the simulation model of a given MICAS configuration.

6.5.1 A C++ "metamodel" for the MICAS Architecture

The MICAS C++ "metamodel" specifies the MICAS resources using type definitions of the C++ language. A *struct* data type is used to define these resources, while the fields of each *struct* type are used to represent their properties. The approach enables the use of different MICAS resources as properties of other MICAS resources, similarly to a metamodel definition.

Modeling the Static Functional Configuration

A MICAS configuration is composed, at top-level, of several domains. Thus, a *Domain* data type is defined as follows:

```
struct Domain {
  std::string name;
  unsigned int domain_id;
  Bus* busList[10];
  int b_no;
  Process* processList[10];
  int m_no;
  struct Process* master_domain_socket;
  struct Process* master_domain_ctrl_socket;
  struct Process* slave_domain_socket;
  struct Process* slave_domain_ctrl_socket;
  unsigned int DPRAM_int;
  unsigned int int_ctrl_reg_addr;
};
```

A *name* and a numeric *domain_id* are used to identify the domain during the simulation. The domain contains *m_no* number of processes stored in the *processList* array, each of them being characterized in turn by specific information. In addition, *b_no* number buses are present in each domain, and they are similarly stored using a *busList* array. The external connections of the domain are modeled directly by the sockets present in that domain, specifying whether these sockets are connected to a master or to a slave domain. The socket description may be seen as a *routing table* for the inter-domain communication. In our current MICAS implementation, we have assumed that a domain may have at most two sockets communicating with its master or slave domain, but since the metamodel allows it, a more general approach may be followed. We recall that domains have a hierarchical relationship to each other, being possible that each domain has a master and, at the same time, being itself master to another (slave) domain. Two sets of pointers are modeling this information. The *master_domain_socket* and *master_domain_ctrl_socket* are used to indicate to the controller the socket through which the data and respectively, the control communication with the master domain has to be directed. Similarly, a pair *slave_domain_socket*- *slave_domain_ctrl_socket*, indicate to the controller the sockets through which the data and the control communication, respectively, with the slave domain has to be forwarded to. We mention that this information is generated from the detailed model of the system, where the inter-domain communication is modeled via data and control sockets. A *null* pointer in one of this fields means that no master and respectively, no slave domain is connected to the domain in question.

An RTOS is connected to the MICAS master domain, through a DPRAM module (see Figure 6.21) using an interrupt-based mechanism. We decided to model it as a separate entry not only because this is a high-priority interrupt, but also for specifying explicitly if an RTOS is connected to a domain. A non-valid value assigned to this field indicates that no RTOS is connected to the domain in question.

Finally, each controller has an interrupt controller, through which it communicates via a control bus (see Figure 6.21). When an interrupt is raised by any of the domain components, the corresponding interrupt number is passed to the controller via a control register, whose address is modeled by the *int_ctrl_reg_addr* field.

Each HW process included in the *processList* of a given domain is characterized by its own set of properties, as shown in the following type definition:

```
struct Process{
  std::string name;
  enum micas_process_type type;
  unsigned int ctrl_reg_addr;
  unsigned int master_reg_addr;
  unsigned int slave_reg_addr;
  unsigned int irq;
  unsigned int slave_data_buffer;
  unsigned int master_data_buffer;
};
```

The *name* is used during the simulation for debugging purposes, whereas a *type* property specifies whether the HW process is a module, a bridge or a socket, based on the definition of the *micas_process_type* enumeration, which we omit here. Based on its placement relative to the other elements in a domain, a HW process is characterized by other types of information, like address spaces used for communication purposes. Address spaces typically regard the conceptual components. We recall that HW processes are controlled by the controller via a control bus, to which the HW process is connected by a control register. To be able to uniquely identify each HW process on the bus, each control register has assigned a unique identifier, namely the *control_register_address*. When the controller issues a command to a given HW process, in fact it writes the command identifier to the control register address.

The communication on the data bus between different HW processes is handled in a similar fashion. Each HW process is connected to the bus through a master or slave interface, and in addition, it has an unique identifier with respect to that bus. Thus, two such identifiers are defined *master_reg_addr* and *slave_reg_addr*, respectively. The communication between HW processes and controller is done via an interrupt based mechanism. The controller uses an interrupt controller for receiving interrupt signals from HW processes. Each module has an unique identifier (i.e., *irq*) corresponding to the interrupt signal to which it is assigned.

Similarly to HW processes, buses are stored into a *busList*, which contains elements with the following structure:

```
 struct Bus  {
    std::string name;
    unsigned int maxCap;
    unsigned int avCap;
  };
```

Beside the *name*, the total capacity of the bus (*maxCap*) and the available capacity at a given moment (*avCap*) are included as properties.

One decision that we took was to group all the generated information in a single file, rather than create separate files for each domain description. Therefore, at simulation-time, the controllers of different domains will share this information from the same file. We do not consider this to be an impediment, since the generated information is read only, and as such, it does not pose the problem of arbitrating the access to it. As such, all the domain descriptions included in a given MICAS configuration are grouped in a *domainList* array.

```
Domain *DomainList[];
```

Modeling the Dynamic Functional Configuration

On the highest abstraction level of the dynamic perspective of MICAS stands the *Service*. Each domain provides its own service list (i.e., *serviceTable*), in which a number of *s_no* services are stored. Consequently, two new properties have been added to the *Domain* definition, enabling the domain description to encompass both static and dynamic information of a configuration.

```
struct Domain {
  Service* serviceList[10]
  unsigned int s_no;
}
```

The *Service* type is characterized by a *name*, a list of *CompositeStreams*, a pointer to a *subservice* from a remote domain, and an *allocated* flag to be used at run time to keep track if the service is enabled at a given moment in time. The definition of the *Service* is shown below.

```
struct Service {
  std::string name;
  struct Subservice *subservice;
  CompositeStream* compositeStreams[10];
  unsigned int allocated;
};
```

In turn, the *Subservice* is characterized by the identifier (*remote_domain_id*) of the remote domain, from which it can be accessed and the service identifier (*remote_service_id*) in that remote domain. In addition, pointers to the local sockets are provided to indicate to the controller, where to “route” the commands for using a given subservice (*local_ctrl_socket*), and from or to what socket (*local_socket*) it can access or send the data provided by the service. The *Subservice* definition is given in the following.

```
struct Subservice {
  unsigned int remote_domain_id;
  unsigned int remote_service_id;
  Process* local_socket;
  Process* local_ctrl_socket;
};
```

A service is supported by one or more composite streams depicting the data-flow perspective of that service. In turn, each composite stream is implemented by a number (*b_no*) of basic streams, which are included in the *basicStreams* array.

```
struct CompositeStream {
  std::string name;
  struct BasicStream* basicStreams[10];
  int b_no;
};
```

As discussed in the previous section, a basic stream (i.e., a data-flow between two HW processes) provides an intrinsic perspective on the associated control-flow needed to setup the adjacent HW processes. Thus, there is a need for thoroughly characterizing the properties of each basic stream. Similarly to the MICAS metamodel definition, a basic stream has a *name*, a *category* and a *capacity*. In addition, each stream transfers data over a physical *bus*, between a source HW process (*src_process*) and a destination HW process (*dst_process*), which are represented as pointers to the corresponding elements. From a control perspective, a basic stream is equivalent to one or more HW process commands that trigger the data transfers over the bus. These commands are gathered in the *microcommands* array and executed every time the basic stream is triggered.

```
struct BasicStream {
  std::string name;
  enum category cat;
  unsigned int Capacity;
  struct Bus    *bus;//pointer to the bus supporting the stream
  struct Process *src_process;//
  struct Process *dst_process;
  Microcommand* microcommands[10];
  unsigned int  m_no;
  };
```

The *Microcommand* is characterized by a *name* and an implementation (*impl*), corresponding to the *CommandInterface* and respectively, to the *CommandImplementation* elements of the MICAS Functional library. In turn, the implementation consists of a command identifier (*command*), which is a numeric value to be written by the controller to the control register (*master_address*) of the master HW process. The microcommand will also specify the address (*slave_address*) of the slave HW process to which the master has to communicate over the bus.

```
 struct Microcommand {
   std::string name;
   struct impl {
    unsigned int command;
    Module* slave_address;
    Module* master_address;
    } impl;
  };
```

We mention that these type definitions and their data structures are independent of specific configurations that can be created in MICAS. They are intended only to provide a common framework to specify the MICAS architecture in C++. Although specified using a textual notation and a programming language, these type definitions may be regarded as a textual "metamodel" for MICAS, that is used at simulation level.

6.5.2 Generating the Simulation Model

From the models describing the static and dynamic perspectives of a given MICAS configuration, two kinds of information are extracted, in order to be used during simulation.

Generating the Functional Configuration

Due to the strong equivalence between the MICAS model elements and the data types defined by the "C++ metamodel", the code generation process is straight forward. It only implies the interrogation of the MICAS models and generating the corresponding information at code level.

We mention once again that a single file is generated for the entire MICAS system and that this file is shared at simulation-time by different controllers. It is also worth noticing that the generated file has two parts, declaration and initialization, similar to a C++ program. In the *declaration* part, the MICAS components of the generated configuration are declared using the C++ data types defined previously. An example corresponding to the *Audio* domain (see Figure 6.25) of the DAVR case study is presented below.

```
Domain Audio;
Module Audio_Socket3;
Module Audio_socket_control_sfr_reg;
Module Audio_AudioEncoder;
BasicStream Audio_S11;
Microcommand mc_S11_Audio_record_sound_wav;
Microcommand mc_S11_Audio_transmit_data_to_domain;
BasicStream Audio_S13;
Microcommand mc_S13_Audio_encode_wav_2_mp3;
Microcommand mc_S13_Audio_transmit_data_to_domain;
CompositeStream Audio_encodedAudio;
Module Audio_SoundRecorder;
Bus Audio_Bus1;
BasicStream Audio_S12;
Microcommand mc_S12_Audio_record_sound_wav;
CompositeStream Audio_unencodedAudio;
Service Audio_encodeAudio;
Service Audio_plainAudio;
```

In the *initialization* part the properties of each declared component are initialized with data extracted from the MICAS models. Due to the large size of the generated code, we only show the properties of the *Socket3* HW process and of the *S13* basic stream, corresponding to a static, and respectively, to a dynamic view of the system.

```
Audio_Socket3.name = "Socket3";
Audio_Socket3.master_ctrl_reg_addr = 10;
Audio_Socket3.slave_reg_addr = 1;
Audio_Socket3.slave_data_buffer = 32;
Audio_Socket3.master_data_buffer = 32;
Audio_Socket3.irq = 2;
Audio_Socket3.type = SOCKET;
.......
```

```
Audio_S13.name = "S13";
Audio_S13.bus = &Audio_Bus1;
Audio_S13.src_module = &Audio_AudioEncoder;
Audio_S13.dst_module = &Audio_Socket3;
Audio_S13.Capacity = 100;
Audio_S13.cat = MP3;
Audio_S13.microcommands[0] = &mc_S13_Audio_encode_wav_2_mp3;
Audio_S13.microcommands[1] = &mc_S13_Audio_transmit_data_to_domain;
Audio_S13.m_no = 2;
```

We discuss in Chapter 7 how the generated code is embedded and used within the MICAS SystemC simulation framework.

Generating the Structural Configuration

The process of generation the structural description of a MICAS system implies several steps: initially the Detailed MICAS model (Figure 6.32-(top)) is transformed into an implementation model (Figure 6.32-(bottom)), in which each detailed element is represented in terms of *Specifications* and *Interfaces* of the implementation component. These elements are extracted from the MICAS Implementation library. A given interface type is composed of ports, which are automatically interconnected based on their definition. Currently, the ports are paired based on their name, which proved a sufficient approach in our case study. If a more elaborated approach is required, other port related information like type, direction, etc. may be considered.

We also mention that, since the implementation model is an intermediary model that is not edited by the user, there is no tool support provided for graphically editing its elements. The diagrams presented in Figure 6.32 are only schematic representations of the underlying models, used for illustrating the approach.

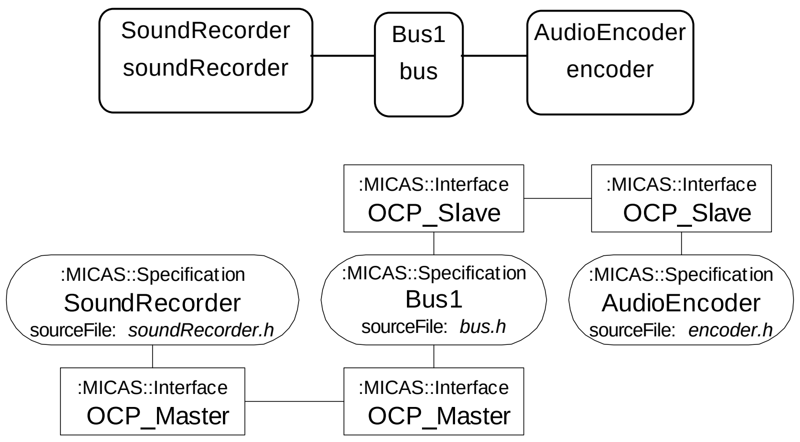


Figure 6.32: Examples of a Detailed model (top) and its corresponding Implementation model (bottom)

It is important to remark that using a well-defined and structured Implementation library allows us to use a completely generic port connecting engine for automatically generating the top-level configuration file of the simulation.

Since the result of the transformation follows the SystemC syntax, we will discuss it in more detail in Chapter 7, along with the simulation framework of the MICAS architecture.

6.6 Summary

In this chapter we have proposed a graphical DSL for the MICAS architecture and a collection of tools to assist the design of MICAS configurations. We have defined a DSL using metamodeling techniques and we have employed the use of a metamodeling tool (i.e., Coral) to provide tool support for the DSL. The combination of the DSL and tool support may be regarded as a *design framework* for the MICAS architecture.

The MICAS metamodel encodes several abstraction layers of the MICAS architecture. The MICAS hardware is designed at three abstraction levels: *conceptual* – modeling the functional components in the system, *detailed* – modeling the communication mechanisms, and *implementation* – modeling the physical components in terms of interfaces and ports.

In addition, a programming model has been proposed to abstract away the hardware details. In the previous versions, the MICAS architecture featured a control-oriented programming model, in which a set of three commands have been used for controlling the functionality of the MICAS configurations at run-time. In this chapter, we have proposed a data-flow oriented programming model, which not only allows the designer to work on a higher abstraction level, as compared to the previous model, but also provides a more natural representation for the data-flow nature of the multimedia processing applications. The programming interface proposed by this model is specified in terms of processing tasks (i.e., services) provided by each domain, which, in turn, are implemented in terms of data streams.

A design process has been proposed for modeling the MICAS architecture. The process specifies how different models of MICAS are obtained and transformed at different abstraction levels. Model transformations have been specified between the steps of the process. The MICAS metamodel encodes the design steps by limiting the design decisions that the designer can make during the process. We would like to see these limitations in a broader sense, namely as guidelines for the design process. The proposed MICAS design process follows a library-based approach, in order to provide support for automation and reuse. It is important to notice that the libraries used during this process are also modeled at different abstraction levels.

Tool support for creating and editing MICAS models has been achieved by customizing the Coral modeling framework. Taking benefit from the Coral's configurability, we have proposed several designs aids for assisting the design process. In addition, a number of systematic transformations have been defined to evolve the abstract specifications into more concrete ones. Some of these transformations have been automated, others that are strongly dependent on the designer's intervention are performed manually, although some design aids could be suggested.

From the work presented in this chapter, several conclusions may be drawn. Metamodels provide the designer with DSLs that are completely independent of UML and other modeling languages. In change, from the perspective of the metamodel designer, a strong familiarity with metamodeling principles and MDA standards is necessary. Defining a metamodel for a given application domain is not enough, but an important effort has to be put into implementing tool support for the metamodel. Moreover, the metamodeling capabilities of the tool and the easiness in providing customized editors for a given application domain are also important. Another observation stands in the portability of a metamodel definition between tools. If the class diagram of the metamodel definition and the associated OCL constraints can be imported in other tools, it is still an open question how and if the tool customizations can be exchanged.

The introduction of the *Category* as a linkage between the static and the dynamic aspects of the MICAS metamodel, facilitated the selection of the HW process' functionality and consequently, of its commands based on the combination of the input/output data-flows that it has to process. For instance, in case of a video encoder that receives an input data-flow of type RAW video and transforms it into an output data-flow of type AVI, the "RAW2AVI" functionality has to be invoked. The approach is momentarily manual, but the effort in automating it seems to be rather small. Another benefit of using the *Category* as an attribute of *BasicStreams* is that it enables one to ensure the consistency of groups of basic streams (i.e., *compositeStreams*). For instance, the designer/tool can check at design time, that the category of a given stream is supported by both the source and the target HW processes.

We can also remark that the MICAS DSL is intended to be used by MICAS designers. Therefore, several decisions taken during the metamodel implementation, as well as the design aids and graphical notations provided, are intended to make the DSL easy to use by the MICAS designer.

Future work may look into the addition of several features to the MICAS design framework. Nevertheless, we consider that these features may be easily added on top of our exiting work by using similar concepts and techniques. For instance, the MICAS Implementation library (that is, the existing specifications and their interfaces) is obtained from the MICAS Simulation library. At the moment, the process of building the former library is performed manually, but future work may include an automated approach for generating/updating the elements of the Implementation library from existing SystemC specifications. This would facilitate the plugging in of new SystemC libraries (eventually from third party vendors) into the MICAS design framework.

Finally, we remark that applications targeted to the MICAS architecture can also be subjects to performance requirements and physical constraints. Similar to the TACO project, area and power consumption of the MICAS components can be embedded within the MICAS metamodel, more specifically within the MICAS libraries, to provide support for the design space exploration at system-level. However, the approach has not been thoroughly explored in the current work, but it constitutes subject of future work.

Chapter 7

Simulation and Exploration of Programmable Architectures

Due to the increasing complexity of the system specifications, new methods are required for detecting design errors in the early stages of the development, as well as for insuring the physical characteristics of the final product. System-level simulation and estimation of specifications have become necessary tools for the designer, in order to enable the evaluation of the system specifications against requirements at early stages of the development before going to hardware implementation. The approach eliminates costs and shortens the design life cycle of new products. According to Moretti [88], most of the integrated circuits developed today require at least one return to early phases of the development, due to errors.

Simulation allows one to execute the system specification at different levels of abstraction. There are two conflicting trends in simulating embedded systems. On the one hand, the simulation should be performed as early in the development process as possible, in order to avoid the propagation of design errors to later phases. On the other hand, simulating the system at later phases of development, when the system specification is more complete, allows one to check a wider range of the system's properties.

Besides simulation, preliminary evaluations of the system's physical characteristics are required. Such evaluations provide an approximation of the final implementation and are typically obtained via estimation techniques. System-level estimation is of particular importance to embedded systems, since by their definition, they must comply with tight constraints, in terms of size, energy consumption and cost. Similarly to simulation, the same conflicting trends apply. The earlier the estimation process is performed, the less precise estimates are obtained. Nevertheless, the approach lets us detect and remove initial errors, before being propagated to the later phases of development.

In this chapter, we discuss the simulation and estimation of programmable architectures in the context of the two architectures introduced in the previous chapters, namely TACO and MICAS. The chapter is structured in two parts. In the first part, we briefly introduce the SystemC language, and then we discuss the simulation environment of the MICAS architecture, emphasizing how the different perspectives MICAS have been specified and integrated within the simulation environment. In the second part, we exemplify the architectural exploration process of programmable architectures by showing how several configurations of TACO are evaluated for supporting an IPv6 router application.

7.1 Simulating the MICAS Architecture

In this section, we discuss the relevant aspects behind the MICAS simulation process, and we show how the static and dynamic perspectives of MICAS are combined at simulation level. In addition, we present different design decisions behind the implementation of MICAS Simulation library.

7.1.1 The SystemC Language

Several languages [51, 105] have been proposed in the last decade to serve as executable system specifications. One such language is SystemC [100], an object-oriented extension of C++ for hardware specification.

SystemC provides a number of macros for hardware-oriented concepts (e.g., ports, signals, clocks, modules, etc.) and the notion of timing for creating cycle accurate specifications. There are two main artifacts defined in SystemC to specify a system: *modules* and *processes*. *Modules* are the building blocks of the system, allowing one to decompose the system in a hierarchical manner. Each module consists of an interface and an internal behavior. The interface, which is not explicitly modeled in SystemC, is composed of SystemC *ports*. These ports may be input, output or both, and may have either a SystemC-defined data type or any other user-defined data type. The ports of different modules are interconnected via *signals*, which provide the basic physical infrastructure for inter-module communication. *Processes* are pieces of behavior of the system, which run concurrently with other processes. Processes are located inside modules and communicate with processes of other modules via the ports.

Therefore, from a structural perspective, SystemC models a system as a set of modules interconnected through signals, whereas from a behavioral perspective, the same system is modeled as a group of processes, that interact to each other in order to provide the overall functionality of the system.

From a syntactical point of view, a module is represented as a C++ class, called *SC_Module*. Such a class may contain attributes, methods and has a constructor method. The attributes may be other SystemC defined ports (*sc_in*, *sc_out*, *sc_inout*) or regular C++ attributes that are used internally by the module. Class methods are specified using plain C++ syntax. Finally, the constructor is used to create class instances at run-time. A small difference can be met in the way the constructor is specified and used. The constructor of the module registers one or more of the class methods as concurrent processes. In addition, it declares the list of ports, to which each process is sensitive.

The SystemC language imposes the separation of a module specification into two files: header (e.g., *module.h*) and body (e.g., *module.cpp*). The header file declares the structure of the module in terms of ports, internal signals and processes (including their reactiveness to signals). The body provides implementations of the processes declared in the header file.

The structure of a system in terms of module instances and their interconnection is specified by the *main()* function, which is placed in a separate file, namely the *main.cpp*, also depicted as the "SystemC top-level file". Simulation related information, like clock resolution, simulation running time, etc., may also be included in this file.

The SystemC simulation process relies, at run-time, on two phases: *elaboration* and *simulation*. In the *elaboration phase*, the components of the system are instantiated from module classes and their ports are interconnected via signals. In the *simulation phase*, once all the system resources are in place, the dynamic behavior (i.e., processes) is started and executed.

7.1.2 The MICAS Simulation Library

The *MICAS Simulation library* is used for providing ready-built SystemC specifications of the MICAS resources. Following this approach, only a top-level configuration file of the SystemC model has to be generated, in order to specify the configuration of the MICAS architecture. The library is designed to meet the following goals:

- to maximize the reuse capabilities provided by the SystemC language by providing reusable versions of the modules, which may be used in several architectural configurations without requiring modification;
- to promote a library organization that easily integrates with the MICAS design methodology, such that the simulation model of a given MICAS configuration can be automatically generated;
- to facilitate the co-simulation process by easily integrating the application code with the simulation model.

The components of the MICAS Simulation library are implemented using the SystemC language features discussed in the previous subsection. We will not present the whole library in detail, but only those aspects that are specific to the MICAS architecture and that are relevant for this work. Each hardware resource of the MICAS architecture has a corresponding implementation as a SystemC class.

In order to improve the reusability of module specifications, we have pursued two main directions when redesigning the components of the MICAS Simulation library: reusable module interfaces and reusable behavior specifications.

Providing Reusable Module Interfaces

Reusable module interfaces have been created to facilitate the automatic generation of the SystemC top-level file from MICAS implementation models. As such, we have identified port collections (i.e., interfaces), which are shared by different module specifications. An interface comprises several ports with well-defined name, type and direction. The main source of information for this process is given by the various technology standards (e.g., OCP specification) used in MICAS.

The SystemC language does not provide a mechanism for the explicit separation of module ports into interfaces. Therefore, we only make sure that the description of the ports belonging to the same interface type is consistent for different modules of the specification. The approach should be seen more as a guideline to be followed during the module specification process, rather than a systematic method. However, although this approach implies an apparent extra effort in building the modules, this effort is outweighed by the benefits in terms of code consistency, error elimination and automation. Furthermore, we have been able to use the identified interface types for creating 'high-level' descriptions of the SystemC modules by taking advantage of the concepts defined by the MICAS Implementation library. As we have discussed in Chapter 6, the Implementation Library promotes reusable interfaces and ports, where each interface type is only once declared and specified, and then assigned to different component specifications that are using it. This approach facilitates not only the addition of new SystemC modules to the MICAS Implementation library, but also the automated port interconnection, at SystemC level, during the code generation process.

As example, we present the *socketMaster* module (Figure 7.1). A socket has several interfaces: one to the external environment, through which the communication to the socket network (i.e., off-chip buses) is implemented, and one to the internal environment, namely to an OCP bus. In addition, a control interface that is used for communicating with the domain controller may be present, in which case the module is regarded as a *master* module (i.e., it is controlled by the controller

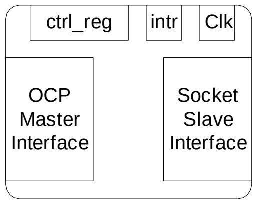


Figure 7.1: Interfaces of the *socketMaster* module

and initiates transfers on the OCP bus). The corresponding SystemC header file (*socketMaster.h*) of this module is presented bellow.

```
SC_MODULE (socketMaster) {

  // OCP Master interface
  sc_in  <bool> M_SCmdAccept, M_SDataAccept;
  sc_in  <sc_uint<8>> M_SResp;
  sc_in  <sc_uint<b_size>> M_SData;
  sc_out <bool> M_MDataValid, M_MRespAccept;
  sc_out <sc_uint<8>> M_MCmd, M_MBurst, M_MAddrSpace, M_MByteEn;
  sc_out <sc_uint<32>> M_MAddr;
  sc_out <sc_uint<b_size>>M_MData;

  // Socket Master interface
  sc_in <sc_uint<32>>data_in;
  sc_in <bool> data_valid;

  // Interrupt interface to MCU
  sc_out <bool> intr;

  // Control interface to MCU
  sc_in <sc_uint<8>> ctrl_reg;

  // Clock interface
  sc_in_clk Clk;
  .....
 }
```

Providing Reusable Module Specifications

One of the main features of SystemC is that it promotes the reuse of module specifications by allowing one to create several instances of the same module. This approach enables one to reuse the same module specification for implementing (simulating) several hardware components of the same configuration. However, each module instance has to be made "aware" of its configuration settings in terms of assigned address spaces, IRQ numbers, parameters, etc. This information has to be passed to the instances at instantiation time, or following the SystemC terminology, at elaboration time.

As we have briefly mentioned in Chapter 5, a similar approach has been followed for assigning address spaces to sockets in the TACO simulation. In TACO, these address spaces are dynamically assigned by the controller (i.e., the interconnection network controller), at elaboration time. Such an approach cannot be applied in case of the MICAS architecture, due to several reasons. Firstly, not all address spaces can be accessed by the MICAS controller (e.g., module addresses on the OCP bus). Secondly, the designer manually assigns some of the address spaces (e.g., the IRQ numbers) based on certain heuristics (i.e., HW process priority). Thirdly, the MICAS controllers are distributed over several domains and thus a centralized approach is difficult to follow.

Therefore, in MICAS we have employed a different approach in which we assign the address spaces based on the information extracted from the models of the MICAS DSL. For this purpose, we have proposed (in Chapter 6) a C++-based description of the MICAS hardware properties, namely the *functional static configuration*. Using the information provided by the functional static configuration, we can configure, at elaboration time, SystemC module instances with specific information. The approach enables us to reuse the same SystemC module specification in several architectural configurations. For instance, in case of our DAVR configuration, two MICAS modules *VideoEncoder* and *AudioEncoder* are used, each of them belonging to different MICAS domains and, consequently, controlled by different controllers. If a generic SystemC *encoder* module is used to simulate both components, it has to be instantiated once for each of the two MICAS modules, and the functional information characterizing each MICAS module has to be passed to its corresponding instance.

Couple customizations have been applied to the components of the MICAS Simulation library. Firstly, we have defined a mechanism that enables us to pass the configuration properties to module instances at elaboration time. Secondly, the processes implementing the behavior of modules have been customized to take this information into account.

Passing functional information to module instances. As mentioned previously, each SystemC module specification is basically a C++ class. As such, beside the SystemC specific constructs, one can define additional properties of that class, like attributes and methods.

We have declared several C++ data types (e.g., *Domain*, *Process*, *Bus*, etc.) in Section 6.5.1, each of them specifying the static functional properties of a specific type of MICAS hardware resource. We integrate each such data type with the corresponding SystemC module by declaring a *mymodule* attribute of the module class. For instance, classes specifying MICAS processes contain a *Process mymodule* attribute, whereas classes specifying bus modules have a *Bus mymodule* attribute. An example is given in the following:

```
SC_MODULE (socketMaster){

  SC_CTOR (socketMaster){
   ....
  }
public:
 Process mymodule;
};
```

During the elaboration phase, when modules are instantiated in the *main.cpp* file, the information is passed to a given module in the following way:

```
socketMaster Socket1("Master_Socket1");
Socket1.mymodule = *this_domain->moduleList[Socket1_id];
```

We mention that both the *main.cpp* and the functional configuration data file are automatically generated from MICAS implementation models. Hence, although the resulting *main.cpp* is not subject to errors, which may eventually be introduced by a manual approach, we have wanted to retain the possibility of previewing the *main.cpp* file in a user-friendly manner, by passing the name of the module (e.g., *Master_Socket1*) "in clear" to its instance.

It is also worth mentioning that we have followed a similar approach in case of the modules implementing the MICAS controllers, with the difference that the entire functional configuration of a domain is passed as an attribute to the controller (i.e., *MCU1.mymodule = this_domain;*). We have employed this approach since the domain controller manages the resources of the entire MICAS domain and therefore, it requires access to the properties of all domain resources.

Customizing module behavior. Having configuration information passed, at elaboration time, to SystemC modules also requires the customization of the SystemC processes modeling the behavior of each module, such that they take into account the fields of the *mymodule* data structures.

For instance, the previously discussed *socketMaster* module uses two methods (*transfer_over_socket()* and *transfer_to_slave()*) to specify its internal processes, as shown below:

```
void transfer_to_slave();
void transfer_over_socket ();

SC_CTOR (socketMaster){

  SC_METHOD (transfer_over_socket);
  sensitive_pos (Clk);

  SC_CTHREAD (transfer_to_slave, Clk.pos ());
}
```

The *transfer_over_socket()* method manages the data transfer from the socket over the external socket network, while the *transfer_to_slave()* method handles data transfers on the local OCP bus. The constructor of the module (i.e., *SC_CTOR*) registers each method as a process, using one of the process types defined by SystemC: *SC_METHOD* or *SC_THREAD*. An *SC_METHOD* process executes to completion each time a signal from its sensitivity list (in our example *Clk*) changes its value. In change, an *SC_THREAD* process executes, whenever a change in the sensitivity list occurs, until the first *wait()* statement in its specification. Consequently, its execution is suspended until a new occurrence of an event resumes the process from the point where it was suspended. An *SC_CTHREAD* process is a special type of *SC_THREAD*, that is only sensitive to the clock signal.

Each module process has a corresponding implementation, which is situated in the *.cpp* file of the module specification. For the sake of example, an excerpt of the code implementing the *transfer_to_slave()* process is shown below. The presented code reads the control register address of a MICAS component (*mymodule.master_ctrl_reg_addr*) and writes it to the *M_MData* port of the *socketMaster* instance.

```
void socketMaster::transfer_to_slave() {
 ....
 M_MData.write(mymodule.master_ctrl_reg_addr);
 ....
 }
```

Therefore, using the functional properties of the module as variables, instead of having them hardcoded in the process specification, enables us to reuse the same process in a generic manner.

As a final note, we mention that although the process of upgrading the library required some additional effort, the benefit of the approach is twofold: a) it enables for different instances of the same module specification not only to be instantiated in different architectural settings, but also to reuse the same module for implementing several MICAS components; b) it facilitates the automated generation of the SystemC top-level file, as we discuss in the following.

7.1.3 The SystemC Top-level File

Based on the previous customizations of the MICAS Simulation library, the process of generating the SystemC top-level file for a given configuration is fully automated. The implementation of this process is not a contribution of the author of this thesis, and hence, we omit it here. However, for the sake of the example, we present below the SystemC code generated from the DAVR *Audio* domain presented in Figure 6.25.

```
 //main.cpp
 #include "microcontroller.h"
 #include "interrupt_SFR_register.h"
 #include "AHB_bus.h"
 #include "bus.h"
 #include "soundRecorder.h"
 #include "socketSlave.h"
 #include "encoder.h"
 #include "inverter.h"
 #include "interrupt_controller.h"
 #include "sfr_bridge.h"
 #include "module_SFR_register.h"
 #include "socket_control_SFR_register.h"
 #include "config1.h"

namespace MicasSystem {
  namespace Audio {
    microcontroller MCU3("Audio_MCU3");
    socket_control_SFR_register socket_control_sfr_reg("Audio_socket_control_sfr_reg");
    socketSlave Socket3("Audio_Socket3");
    interrupt_controller Interrupt_controller("Audio_Interrupt_controller");
    encoder AudioEncoder("Audio_AudioEncoder");
    soundRecorder SoundRecorder("Audio_SoundRecorder");
    sfr_bridge SFR_Bridge("Audio_SFR_Bridge");
    module_SFR_register SFR_Register_Socket3("Audio_SFR_Register_Socket3");
    module_SFR_register SFR_Register_SoundRecorder("Audio_SFR_Register_SoundRecorder");
    bus Bus1("Audio_Bus1");
    module_SFR_register SFR_Register_AudioEncoder("Audio_SFR_Register_AudioEncoder");
    interrupt_SFR_register Interrupt_SFR_register("Audio_Interrupt_SFR_register");
  } // Audio namespace end
} // MicasSystem namespace end

int sc_main(int argc, char* argv[]) {
  sc_clock TestClk ("TestClock", 10, SC_NS, 0.5); // Static
  initialize();

  { using namespace MicasSystem::Audio;
```

```
    MCU3.Clk(TestClk);
    socket_control_sfr_reg.Clk(TestClk);
    Socket3.Clk(TestClk);
    Interrupt_controller.Clk(TestClk);
    AudioEncoder.Clk(TestClk);
    SoundRecorder.Clk(TestClk);
    SFR_Bridge.Clk(TestClk);
    SFR_Register_Socket3.Clk(TestClk);
    SFR_Register_SoundRecorder.Clk(TestClk);
    Bus1.Clk(TestClk);
    SFR_Register_AudioEncoder.Clk(TestClk);
    Interrupt_SFR_register.Clk(TestClk);

    Domain* this_domain = domain_list[Audio_id];
    MCU3.mydomain = *this_domain;

    socket_control_sfr_reg.socket_ctrl_register_addr =
        this_domain->moduleList[socket_control_sfr_reg_id]->master_ctrl_reg_addr;
    Socket3.mymodule =
        *this_domain->moduleList[Socket3_id];
    SFR_Register_Socket3.module_sfr_register_addr =
        this_domain->moduleList[Socket3_id]->master_ctrl_reg_addr;
    AudioEncoder.mymodule =
        *this_domain->moduleList[AudioEncoder_id];
    SFR_Register_AudioEncoder.module_sfr_register_addr =
        this_domain->moduleList[AudioEncoder_id]->master_ctrl_reg_addr;
    SoundRecorder.mymodule =
        *this_domain->moduleList[SoundRecorder_id];
    SFR_Register_SoundRecorder.module_sfr_register_addr =
        this_domain->moduleList[SoundRecorder_id]->master_ctrl_reg_addr;
    Interrupt_SFR_register.int_ctrl_SFR_register_addr =
        this_domain->int_ctrl_reg_addr;

    //connect ports
    .....
  }
  {//connect domains
  .....
  }//end ELABORATION PHASE
  int n = 600000000;
  if( argc > 1 ) std::stringstream(argv[1], std::stringstream::in) >> n;

  sc_start (n); //START SIMULATION
  return 0;
} // sc_main end
```

7.1.4 Integrating the MICAS Programming Model within the Simulation Environment

As we have discussed in Chapter 6, the MICAS programming model is defined at two abstraction levels. At domain level, the programming model consists of a DFD-like view, in which the services are implemented in terms of data streams. We recall that during the design process, the services of each domain are identified and expressed in terms of basic streams. In the code generation stage, the services and their implementations are transformed into a C++ representation, along with the corresponding microcommands implementing each basic stream. The set of microcommands supporting each service consists of pieces of code, which are run by controllers when a given service is used.

At application level, the programming model is composed of programming primitives (i.e., services) and control primitives for managing the services. We have defined five control primitives in MICAS for managing services:

INQUIRY – is a request sent by a master controller to a salve controller to interrogate the list of services available in the slave MICAS domain. In response, the slave controller replies with a list of services and their current status (i.e., *allocated*, *deallocated* or *active*). Each controller propagates the inquiry command to all its slave MICAS domains, in order to detect their services. Consequently, it collects the responses and returns them to its master domain. Future versions of the MICAS Simulation library can enhance these responses, in order to include additional data characterizing the service (e.g., capacity, category, etc). The approach would allow us to combine, at run-time, services into consistent scenarios.

ALLOCATE – when such a request is received, the controller attempts to allocate a given service by reserving capacity over buses. In the current MICAS implementation, it is assumed that the communication bottleneck resides in the bus communication only. In a more elaborated version, other resources (e.g., processes) could be taken into consideration. As already mentioned, a service is composed of a sequence of basic streams, each stream requiring a certain transport capacity over the physical bus. In certain cases, it may happen that streams belonging to several services are using the same bus and therefore, the controller has to ensure that, by allocating a new stream, the basic capacity of the bus is not exceeded. If any of the basic streams implementing the service cannot be allocated, the allocation process is canceled and, an error message is returned to the master controller.

DEALLOCATE – requests a slave controller to deallocate a given service. Consequently, the slave controller frees the bus capacities reserved for the service. In addition, it sends a corresponding message to the master, to notify the accomplishment of the request.

ACTIVATE – request sent to a controller to activate a service in a slave domain. The service must be allocated before being activated. If the service is not allocated, an error message is returned to the master controller. Furthermore, if the service is already active (e.g., activated by a different request) a corresponding error message is sent back. Upon service activation, the controller starts executing the service specification and the service remains in an active state until it is deactivated.

DEACTIVATE – deactivates a service in a local or remote domain by resetting the operation of the hardware components involved. A corresponding message is sent to the master to notify over the completion of the operation.

Using these control primitives, several services can be started and, eventually, combined into scenarios. Currently, there is no means to check whether different services are compatible to each other from the point of view of their data type (e.g., a service providing audio cannot be combined with one displaying video), throughput (e.g., the service receiving data is able to consume in a timely fashion all the data provided by the sending service), or direction (e.g., one cannot use two services only producing or only consuming data, but a "consumer" and a "producer" are required). Such kind of rationale may currently be applied only at application level and based on the designer's experience. On the other hand, we consider that the current approach suffices in providing consistent scenarios by using subservices of a service. Future versions of the MICAS simulation could include a more advanced service description, which may also be integrated within the MICAS metamodel.

The programming model of MICAS is supported at simulation time by MICAS domain controllers. This support manifests in two directions: a) executing the functional primitives (i.e., services); b) implementing the control primitives. For handling these tasks, we have devised a dedicated piece of software to be run by controllers, called *service_manager*. The main functionality of the *service manager* is to receive service requests from master controllers, to process them based on the description of each service, and to provide an appropriate response.

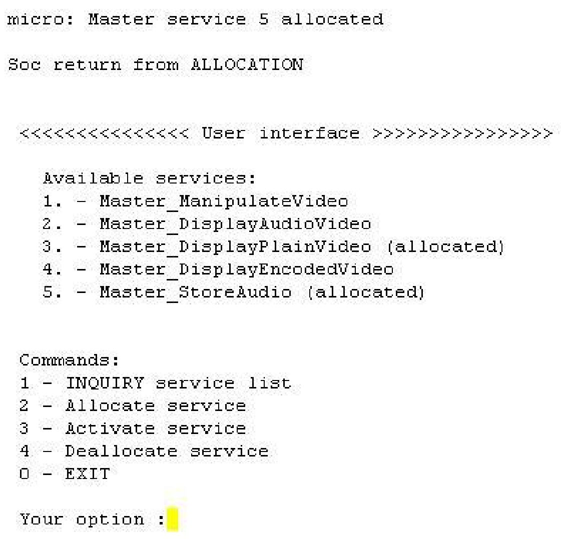
The interesting part of the service manager is the way it uses the functional configuration, more specifically the *dynamic functional configuration*, of each domain. Based on this configuration, the service manager accesses to the properties of all the components (i.e., both static and dynamic ones) of a given domain. For instance, when an activation request is received for an already allocated service, the service manager reads the service description from the *domain.serviceList[]* array. Consequently, each basic stream (read from *service.basicStreams[]*) of the service is transformed into corresponding microcommands, which, in turn, are extracted from the *BasicStream.microcommands[]* array. The resulting list of commands is dispatched to the appropriated modules in an orderly fashion. More details about this approach have been discussed in Section 6.4.7.

The *service_manager* has been defined and implemented in a generic manner to contribute to the reusability of the controller specifications. This enables us to instantiate the same controller specification, and consequently, an identical version of the stream manager, in each MICAS domain, without concern whether the domain has a master or a slave role, or what types of components are present.

7.1.5 The RTOS Application Interface

Taking advantage of the fact that all MICAS domains provide the same control interface (i.e., control primitives) to the other domains, enables us to reuse this interface for supporting also the communication between the RTOS application and the master domain. This approach relies on the fact that MICAS RTOS may be seen as *the master* of the MICAS *Master* domain. Consequently, the same five control primitives are provided to the RTOS, from where applications can employ the functionality of a given MICAS configuration. Being able to use the same control interface both in-between domains and between master and RTOS, independently of the underlying hardware, proved once again the benefits of the MICAS HAL concept (see Figure 6.4) in terms of abstraction, flexibility and reuse.

We have implemented an application, at RTOS level, for using it as a case study for our approach. The application is a simple interactive user interface, integrated within the MICAS simulation environment, through which one can use the MICAS control primitives for enquiring and employing the services of a given MICAS configuration. Figure 7.2 provides a caption of the application simulation, in which the services of the DAVR device have been detected. Two of these services, namely *3.* and *5.*) are reported as *allocated*.



```
micro: Master service 5 allocated

Soc return from ALLOCATION


 <<<<<<<<<<<<<<< User interface >>>>>>>>>>>>>>>>

   Available services:
   1. - Master_ManipulateVideo
   2. - Master_DisplayAudioVideo
   3. - Master_DisplayPlainVideo (allocated)
   4. - Master_DisplayEncodedVideo
   5. - Master_StoreAudio (allocated)


 Commands:
 1 - INQUIRY service list
 2 - Allocate service
 3 - Activate service
 4 - Deallocate service
 0 - EXIT

 Your option :
```

Figure 7.2: Simulation of the DAVR system on MICAS. Caption of the application user interface

When we have discussed the MICAS service design process (Chapter 6), we pointed out that there are two possible approaches in defining services. Either the services of each domain are presented to the RTOS, and the user (or the application) decides how to combine them into consistent scenarios, or scenarios are created at design-time by using the *subservice* property of a service. Both approaches are currently supported by the MICAS simulation framework. However, since we have decided to experiment with the latter approach, only the services of the master domain are presented, as scenarios, to the RTOS. If the former approach is to be followed, it would allow one to combine, for instance, a hypothetical service *Master_CaptureAndStoreVideo* with an the existing *Master_ManipulateVideo* service, and eventually, with a third hypothetical service *Master_DisplayStoredVideo*, in order to obtain a more complex scenario.

7.1.6 Further Considerations

Beside simulating the functionality of different MICAS configurations, other types of information can be obtained from the simulation process. For instance, computing values, like the average or the peak-level bus utilization, might provide a general indication for rearchitecting the configuration or optimizing some of the components, in order to better support the application performance. Such measurements are currently not provided, since they have not been in the scope of this work, yet they can be easily integrated on top of the existing SystemC specification of the MICAS controllers.

7.2 Exploring the Design Space of TACO Processors

This section[1] discusses the design space exploration process of the TACO architecture. The process relies on the simulation and estimation models of the TACO framework, from where the characteristics of different TACO configurations are obtained. The main goal of this section is not to define a rigorous exploration process, but rather to show that, by having at our disposal estimation information at system-level, we can quickly and reliably explore TACO configurations. Out of these configurations, we select the “suitable” one(s) for hardware synthesis.

In Chapter 5, we have discussed the *qualitative configuration* process of the TACO architecture. We recall that our approach targets the functional requirements of the application, without considering performance-related issues. This section discusses the process of identifying and tuning TACO *quantitative configurations*, in order to address the non-functional requirements of the application.

7.2.1 Creating TACO Quantitative Configurations

Our approach to address the non-functional requirements of the application is biased towards exploiting the parallelism of the TACO architecture. The parallelism level of a TACO configuration may be increased by following two procedures: a) multiplying the number of FUs of the same type; b) multiplying the number of buses.

The TACO exploration process starts from a given TACO qualitative configuration and its corresponding application code (see Figure 5.3-(a)), which are obtained from the mapping process. As an initial step, the qualitative configuration is simulated and preliminary performance information is collected, like number of cycles to execute a given application path, bus utilization, register transfers etc. The TACO SystemC simulation framework is not included in the current thesis, yet it has been discussed in previous work [128, 132]. Beside simulation-related information, physical estimates of the configuration are obtained from the estimation process.

Based on the preliminary performance and physical estimation data, one can suggest *quantitative configurations* of the architecture. For each new configuration, the application code is optimized to take advantage of the parallelism of the configuration. The process is followed by the simulation and estimation of the newly created configurations.

The approach is applied iteratively until a “satisfactory” quantitative configuration is obtained. It is this “satisfactory” configuration that will become a candidate

[1]The work presented in this section is based on publication [P.2]

for the synthesis process. At the moment, the design space exploration process is performed manually based on the designer's experience. To exemplify our approach, we present a design case, in which the TACO architecture is customized to support the IPv6 router application specification, which has been discussed in the previous chapters of this thesis.

Suggesting Architectural Optimizations

From the simulation process one can detect bottlenecks in the performance of the system by analyzing different processing tasks of the router. Consequently, optimizations of the TACO configurations are suggested, either by increasing the parallelism level or by designing optimized FUs. By simulating the qualitative configuration of the IPv6 router, we have observed that three processing tasks (i.e., checksum calculation, ICMPv6 signaling and routing table lookup), required optimization in order to improve the performance of the application under study. Consequently, three dedicated FUs have been designed, one for each of the above mentioned tasks.

The Checksum FU computes the Internet checksum of a datagram by reading 32-bit words of the datagram. The ICMPv6 FU builds new ICMPv6 datagrams by prepending received datagrams with an ICMPv6 header (two 32-bit words). This approach saves an important number of bus transports that are normally needed to construct an IPv6 datagram. Finally, a Routing Table FU allows fast access to the fields of the routing table entries stored in the memory. We will not get into the technical details of these FUs, since they are not in the topic of this thesis, yet their detailed description may be found in [132, 134].

It is important to remark though, that each newly designed FU provides its own functional primitives, which may be used to express the application specification during the mapping process, as discussed in Chapter 5. Therefore, the mapping process has to be performed again for those processing tasks that benefit from newly designed FUs.

7.2.2 A Case Study for Exploring the IPv6 Routing Table Design

The goal of our case study is to configure TACO for supporting an IPv6 router application, which provides a 10 Gbps aggregate throughput over Ethernet [38] networks. Due to the limitations of the Ethernet protocol, the largest datagram size that may be transmitted by the router is equal to 1500 bytes (i.e., 12 000 bits). This implies that our router should be able to forward 833 333 datagrams/sec. Based on this value, we can calculate that the maximum processing time for a datagram is 1.2 μs (worst case scenario).

As already mentioned, there are two main data-flows inside the router, corresponding to the routing and to the forwarding processes, respectively. The most time-critical process is the forwarding process, being the one that "measures" the performance of the router. The routing process plays a secondary role, being in charge only for maintaining the routing table. Statistics [79] show that when the topology of the network is stable, the routing table updates appear about every two minutes. Hence, for the sake of simplicity, we neglect the routing process throughout this case study.

The forwarding process relies on several processing tasks of the router, like IPv6 header validation, classification, routing table look up, etc. From simulating the qualitative configuration of the router, we have observed that the routing table lookup plays a dominant role in the forwarding process, accounting for more than 80% of the processor cycles required for forwarding a datagram.

Therefore, in order to optimize the routing table lookup task, we explore several TACO configurations. During this process, we are interested in minimizing the number of processor cycles that are necessary for the forwarding task. In this scope, we examine three different organizations of the routing table. In the first case, we evaluate a sequential organization, which implies a linear searching time. In the second case, a balanced tree organization is simulated, thus resulting in a logarithmic complexity. As a third case, we evaluate an approach where the next destination of a datagram is obtained in a single query of the routing table. A routing table with 100 entries is considered in our study.

In practice, the first two alternatives may be realized using regular memory, at the expense of moving the routing table interrogation process into software. The last approach could be implemented using a hardware-based solution, like a Content Addressable Memory (CAM). Such circuits provide fast matching time in the detriment of a high cost.

For each of the tree routing table organizations we evaluate three different processor configurations obtained by varying the number of buses and the number of functional units used. Table 7.1 presents the resources included in each of these configurations, as well as the physical estimates of the power consumption and physical area of each configuration, obtained from the Matlab estimation model of TACO.

We have simulated each of the three configurations and, based on the simulation results, we have calculated the number of clock cycles that the datagram forwarding process takes to complete in each case. By using this number and the number of datagrams that the router has to forward per second, we compute the required frequency for each configuration. The results are given in Table 7.2.

We mention that the presented estimates are partly based on approximations, as we are interested in quickly identifying the tendencies in which the performance of each routing table organization vary with architectural changes. Out of the simulation process, we also identify processing bottlenecks by looking at the register utilization data. From this data we have concluded that much of the forwarding processing time is spent for matching, comparing, increment and decrement operations. This observation has been, in fact, the motivation for suggesting the Router3 configuration in Table 7.1.

Component type	Router1	Router2	Router3
BUS	1	3	3
Comparator FU	1	1	3
Counter FU	1	1	3
ICMPv6 FU	1	1	1
Input FU	1	1	1
Memory FU	1	1	1
Matcher FU	1	1	3
Masker FU	1	1	1
Output FU	1	1	1
Shifter FU	1	1	1
R'Table FU	1	1	1
Estimated Area [mm^2]	202 731	202 731	292 695
Estimated Power [mW]	97.1	97.1	143.5

Table 7.1: Estimates of the IPv6 router configurations used in the Routing Table case study. Only the estimates of FUs and network controller are included in the calculation. The estimates do not include memory blocks

In addition, the initial application code corresponding to the qualitative configuration is optimized, by performing data and control analysis, to take advantage of the parallelism features of each configuration. Table 7.2 presents the simulation results for each of the three configurations.

In the case of a sequential organization, the results indicate that the required clock speed of the *Router1* configuration should be near 6 GHz. This exceeds the capabilities (which are estimated to be in the vicinity of 1 GHz) of the 0.18 μm standard cell library, that has been targeted in this cases study. By configuring the TACO processor to use three buses, we obtain a required clock frequency of 2 GHz, which also is beyond the capabilities of the implementation technology used. The increase in parallelism, offered by the *Router3* configuration, brought the clock frequency at about 1 GHz, meaning that this configuration could be possible to implement in the 0.18 μm technology.

The balanced tree organization shows an evident gain in performance. Following a similar rationale as in the previous case, the faster two configurations qualify as possible solutions, whereas the slowest one exceeds the capabilities or the current technology.

For the CAM-based solution, we observe a major boost in the performance of the router. As seen in Table 7.2, the speed requirements of the three TACO configurations drop dramatically, in case of the *Router2* configuration. However, increasing the number of FUs, as in the *Router3* configuration, does not anymore seem to offer considerable increase in the performance of the forwarding process. Instead, if this last configuration is selected for implementation, one will have to expect larger area and power consumption, as indicated by the estimates in Table

	Sequential access		**Balanced tree access**		**CAM-based access**	
Config.	Req. speed [MHz]	Bus util. [%]	Req. speed [MHz]	Bus util. [%]	Req. speed [MHz]	Bus util. [%]
Router1	6000	100	1200	100	**118**	100
Router2	2000	97	**600**	97	**40**	98
Router3	**1000**	98	**250**	99	**35**	76

Table 7.2: Required speed and bus utilization for each type of routing table access

7.1. It is also important to mention that the power and area required by the CAM chip are not included in the estimates. Nevertheless, the performance obtained using CAMs recommends them as *the* solution for the routing table implementation. On the downside of this approach, such chips are accompanied by high costs and a larger area footprint, as compared to regular memories.

As a conclusion of the exploration process, any of the six validated configurations (in bold text in Table 7.2) can be chosen to support the IPv6 router implementation on TACO, as long as they comply with the physical constraints (area occupied , power used, etc.) of the application.

7.3 Summary

In this chapter, we have discussed the simulation and exploration of programmable architectures. In the first part of the chapter, we have presented the simulation framework of the MICAS architecture. We have showed how we have customized the components of the MICAS Simulation library by taking advantage of the object-oriented mechanisms of SystemC, in order to provide fully reusable component specifications. The approach allows us to use simultaneously several instances of the same module specification, in order to simulate different components of the MICAS architecture. In addition, it has enabled the automated generation of the SystemC model of a given MICAS configuration, starting from the implementation models of MICAS. We have also integrated the MICAS programming model within the MICAS simulation environment. For this purpose, we have proposed a set of control primitives and a *service manager* to be supported by MICAS controllers, in order to control the invocation and execution of the MICAS functional primitives (i.e., services).

In the second part of the chapter, the architectural exploration of the TACO architecture has been discussed. During this process, we have estimated several configurations of the TACO processor, with respect to their throughput and physical constraints, and some of them have been proposed as candidates for the synthesis process. The entire exploration process is performed, at system-level, before proceeding to hardware implementation. A case study has been presented in order to show that, by having proper tools to estimate TACO configurations at system-level, enables the designer to take design decisions that address the non-functional requirements of the application. We mention that both the estimates and the SystemC code of the explored configurations have been obtained from the UML models of TACO, taking advantage of the transformation scripts provided in Chapter 5. Therefore, the IPv6 router case study discussed in this chapter may also be seen as a validator for the tool support provided by the TACO UML Profile, in order to rapidly create and estimate TACO architectural configurations.

As a final remark, we can conclude from the simulation of the two architectures discussed in this chapter that the system-level simulation of programmable architectures does not provide a magical solution for insuring the “correctness” of the final system, for couple of reasons. Firstly, a system-level executable specification of a system, as complete as it may be, does not represent the “real” system, but only an abstraction of it. This means that, in fact, we are not simulating the system as such, but only its specification. Secondly, since embedded systems are reactive systems (they react to stimuli from the environment and eventually provide response) their I/O behavior has to be tested. This is a non-trivial task, since generating test data to cover all possible scenarios may prove difficult.

Future work is mainly focused towards improving the TACO development process. One research direction is concerned with improving the estimation model in the TACO Profile such that the power consumption and area estimates for the entire configuration can be calculated in UML tools. Currently, only the estimates of the FUs and of the network controller are taken into account in the exploration process. A different research direction may look into combining the TACO UML profile with specialized exploration tools, in order to enable an automated exploration process.

Chapter 8

Concluding Remarks

In this thesis, we have investigated the principles of the model driven paradigm for supporting the development of programmable architectures. Programmable architectures are gaining popularity for providing optimal implementations of today's embedded applications. Similarly to other embedded systems, the designer of programmable architectures confronts with the increasing complexity of specifications and with the pressure of reducing the time-to-market for new products.

The model driven paradigm has become, in recent years, one of the approaches to address the complexity of software systems by promoting *models* to a pivotal role of the development process. Throughout this thesis, we have applied the concepts and mechanisms of the model driven paradigm to the development of programmable architectures.

We have started our study by proposing a methodology for developing programmable architectures, willing to address the characteristics of the specific architectures that we address in this thesis, TACO and MICAS, respectively. The methodology has been defined as an extension of the traditional Y-chart approach [74]. Our work suggests a number of steps in building the system, from initial specifications towards implementation. At each step, design guidelines are proposed in terms of specification languages, concepts, tool support, processes, etc., which are needed to support the methodology. To accomplish this task, we have suggested the model driven paradigm as the general infrastructure for defining and using the previously mentioned artifacts.

This chapter summarizes the solutions that we have proposed here, in order to address aspects related to the development of programmable architectures. The content of this study may be regarded as covering four main topics: application specification, architecture specification, mapping, and architectural simulation and exploration.

Application specification. In order to deal with the increasingly complex application specifications, we have employed a systematic process in which the specification is built starting from the functional requirements of the application. UML plays a central role in the approach, being used as the modeling language throughout the specification phase. The process employs the use case model as the main tool for capturing the requirements. From here, a functional decomposition of the system follows, in which the system is decomposed into subsystems (i.e., objects). For this step, we have adopted an already existing method, namely the 4SRS method [48]. The resulting subsystems are further analyzed by focusing on their internal behavior. Two distinct approaches have been suggested.

In the first approach (Chapter 3), we have proposed the activity graphs as the main UML technique in specifying the object's internal behavior, while collaboration diagrams are used to model the communication between objects. Taking advantage of the hierarchical decomposition of activity graphs, we have proposed a systematic approach for refactoring (e.g. decomposing, grouping, removing, etc.) the objects, based on their internal functionality. The approach lets us discover common behavior that is used by several objects and, eventually, isolate this behavior into separate objects. In addition, the proposed approach could also be seen as a complement of the 4SRS method. An IPv6 router specification has been employed as case study, in order to validate the proposed methodology. UML has been used as a modeling language for the entire process, while tool support has been assumed via existing UML tools. Due to the systematic way in which the analysis is performed, some of the steps are easy to automate, providing prerequisites for speeding up the process.

The second approach (Chapter 4) has been motivated by the observation that an additional perspective of the system has not been adequately represented in the previous case study. This is largely due to the nature of the IPv6 routing application, for which a data-oriented perspective of the system is more natural than a control-oriented one. This problem is not new, several other authors pointing out that UML is not able to properly capture certain views of a system [8, 44]. As such, we have proposed a specification process, in which we have complemented UML with a data-oriented perspective. For this, we have employed data-flow diagrams, the main tool of the structured methods [36]. We have adopted three existing suggestions for combining UML and data-flow diagrams [45] and integrated them in a specification process, which defines how the UML-based and the DFD-based views of the system are interplayed. In addition, we have suggested an approach for transforming some of concepts of each of the two modeling languages into a subset of concepts of the other, respectively, at different abstraction levels. Taking advantage of the availability of a MOF-based metamodel for structured analysis [61], we have been able to use the UML-DFD combination in the same UML tool. Moreover, using the scripting facilities of the SMW tool, we have implemented scripts that support the model transformations of the process.

Architecture specification. The second topic attended in this thesis is the use of the model driven principles to support the specification of programmable architectures. UML has been used as a “hardware modeling language”, yet adapted to provide domain-specific languages for the targeted programmable architectures.

In Chapter 5, we have developed a UML profile for the TACO protocol processing architecture [132]. For this purpose, we have restricted the elements of the UML class diagram for modeling the concepts of the TACO architecture. The TACO Profile has been defined such that, by using UML notations, it enables the specification not only of the hardware structure, but also of the functional (i.e., operations of the architecture) and physical (i.e., physical estimations) properties of the TACO architecture. Benefiting from the TACO Profile definition, we have proposed a UML-based component library, which encompasses in a single UML model several views of the architecture, like simulation, estimation, and synthesis specifications of the components. Having the library implemented and stored as a UML model, we have been able to reuse its elements to rapidly create new processor configurations in the used UML tool (i.e., SMW). In addition, by applying scripting techniques similar to the ones in Chapter 4, we have been able to define a series of transformations to assist us in creating configurations of the processor, in estimating at system-level their physical characteristics, and in generating the simulation and the synthesis models of selected configurations.

A domain-specific language for modeling an industrial programmable architecture, MICAS [109], has been defined in Chapter 6. This time, metamodeling techniques have been employed for defining a MOF-like metamodel. The metamodel definition encodes several perspectives of the system distributed over several abstraction layers. From a hardware (static) perspective, we have modeled the system using three abstraction layers (i.e., conceptual, detailed and implementation). From a dynamic perspective, we have defined a programming model of the architecture, in order to abstract the hardware details of the architecture even more, allowing the designer to focus on the functionality of the architecture. The various perspectives of MICAS have been organized in a design process based on which the initial specification of the system is refined towards implementation. Several libraries have been defined (and encoded in the metamodel) to support the design process on different levels of realization. These libraries not only provide reusable components of the system, but also encode design decisions that can be taken during the development process. The metamodel definition has been implemented in the Coral modeling tool, in order to offer support for editing and transforming MICAS models. We have also devised several customizations of the Coral editors, for assisting the design process. Using the MICAS metamodel implementation in Coral, several steps of the design flow could be automated, including the code generation process. A multimedia processing application has been used as a case study, in order to exemplify the use of the metamodel, for designing the MICAS architecture.

Mapping the application onto architecture. As a third topic of this thesis, we have discussed how application specifications can be mapped onto a given architecture (i.e., TACO). Since our approach has only been targeting the TACO architecture, the discussion has been included in Chapter 5. As a concrete example we have used an IPv6 router implementation on TACO. Out of the mapping process, we have obtained both the qualitative architectural configuration to support the application and the program code to drive this configuration.

Several conclusions follow from this work:

- the use of a programming model to abstract the hardware details of the architecture has allowed us to narrow the implementation gap, such that a systematic transformation process between the two specifications could be proposed;
- the approach has confirmed that using a computation model for the application specification that naturally matches the one of the programming model, facilitates the mapping process considerably;
- by having both the application and architecture specifications models implemented via the same modeling language (i.e., UML), has let us decompose the specification process into several atomic steps, of which some are performed automatically. In addition, the same UML tool has been used for the specifications of both the application and the architecture.

Architectural simulation and exploration. The last topic of this thesis, which has been presented in Chapter 7, focuses on the simulation and exploration of the programmable architectures. In the first part of the chapter, we have discussed the simulation of MICAS architectures. The main contribution of this first part is the construction of a highly reusable simulation library for MICAS by taking advantage of the object-oriented principles of SystemC. Hence, we have built the SystemC modules such that their instances are customized based on configuration-related properties. Moreover, we have defined a hardware abstraction layer for MICAS, for allowing domain controllers of different MICAS domains (including the RTOS) to communicate with each other in a uniform manner. Domain controllers have been specified as reusable components, being able to interpret the service description data independently of the architectural setting of their domain. The customizable and reusable MICAS SystemC modules have enabled an automatic generation of the MICAS simulation model.

The second part of Chapter 7 discussed the design space exploration of the TACO architecture. The main focus of this work has not been on providing a systematic approach in which the process is performed, but rather on showing that the simulation and estimation tools of the architecture, available at system-level, allow us to perform the exploration process within a short time-frame. By simulating and estimating different architectural configurations at the system-level, we have obtained a fast turn-around time in finding well-suited configurations to support the performance requirements and physical constraints of the application. In addition, during the case study presented, we have validated the capabilities of the TACO UML profile to provide tool support for rapid estimation of the TACO processor.

8.1 Discussion

In the following, we discuss issues that we consider relevant to our work, as they have resulted from the research included in this thesis.

Benefits of using model driven techniques in the context of programmable architectures. The main goal of this thesis was to examine the advantage of placing the development of programmable architectures in the context the model driven paradigm. In the following, we briefly discuss the benefits that we consider relevant with respect to this topic.

- *abstraction levels.* By employing model driven techniques, we have been able to define and use several abstraction levels, both for the application specification and for the architecture specification. This approach enabled us to tackle the complexity of the specifications by focusing only on those aspects of the system that are relevant at each step of the development process.
- *graphical modeling languages* may also be seen as an enabler for abstractions of the specification. In addition, allowing the designer to use well-defined language elements at each step of the development process improves the consistency and facilitates the automated refinement of the specifications. Furthermore, by taking advantage of graphical modeling languages we have shown how new or existing graphical DSL may be defined and used in model driven tooling environments, thus facilitating the design process.
- *reuse of tool support.* One of the primary goals of this research has been in reusing or customizing existing model driven tool support (in our case, UML-based tools) to assist the development process. From our research, we may conclude that reusing existing UML-based tools in providing integrated development frameworks for programmable architectures is a viable option. However, the quality and usability of the resulting development framework heavily depends on the capabilities of the selected tool.
- *Automation* is essential in shortening the development time of new products based on programmable architectures. As we have seen throughout this thesis, automation may be employed in all development phases (application specification, architecture specification, mapping, simulation, exploration, etc.). Benefiting from well-defined models on different realization levels enabled us first to define and then to implement automated model transformations. Beside liberating the designer's activity of those mechanical and tedious repeating tasks, automation also improves the quality of the generated artifacts by avoiding the possible errors inherent to a manual approach.

- *IP-based reuse.* Having a well-defined representation of different system components has enabled us to store them as models, to interrogate these models for seeking specific information, and further more, to generate from these models different artifacts necessary in the later steps of the development process. As such, we have been able to define component libraries at several abstraction levels, from where already implemented and tested components can be selected for a given configuration. Moreover, by including in the component descriptions additional information like simulation-, estimation- or synthesis-related, has enabled us to generate system-level models of the system in an automated manner. Thus, reusing IP components has allowed us to shorten the design time and to speed up the design process considerably.
- *libraries of design decisions.* Having at designer's disposal tools like DSLs, automation, and component-based libraries facilitates considerably the development process. However, it is still the designer that has to manually take design decisions at different steps of the process. We have shown in this thesis, that certain design decisions are reusable and thus, they can be encoded and stored in a library for future use. Taking advantage of the model driven framework, we have been able to store such decisions as models and furthermore, to query and reuse them in an automated manner.

UML and object-orientation for hardware specification. Several authors have argued that object-orientation and hardware do not match, because of their different natures [21]. Based on the work presented in this thesis, we may conclude that they are right, only if we regard object-orientation and its associated programming languages as an implementation mechanism for hardware.

However, our opinion is that two of the main features of object-orientation (i.e., the power of abstraction and reuse) are beneficial for abstracting and reusing hardware specifications. These benefits are twofold: a) at system-level, we have employed the object-oriented mechanisms of the SystemC language for creating reusable executable specifications of the system components (in addition, SystemC has been used for abstracting the physical details of hardware, letting the designer focus on and simulate the functionality of the system); b) at conceptual modeling level, UML has been employed as a *hardware description language*. The object-oriented principles behind UML have been used to define DSLs for both TACO and MICAS architectures, respectively. The defined DSLs provide several levels of abstraction of the physical architecture, allowing the designer to focus on relevant details at each stage of the design process.

In addition, we consider that, although initially created for software specification, UML can be extended and applied to other application domains (as long as the concepts of the new domains map naturally to the ones of UML). The approach has allowed us to define DSLs for our target programmable architectures, which provide increased abstraction levels and, at the same time, to take advantage of the graphical notations of UML and of its corresponding tool support.

Profiles vs. Metamodels. Two domain DSLs have been discussed in Chapters 5 and Chapter 6, which rely on the UML extension mechanisms and on the metamodeling techniques, respectively. Several observations can be made to serve as guidelines for future work.

The first observation is that the defined DSLs are intended to be used by domain experts that are not necessarily familiar with the UML notation and modeling techniques. Therefore, the UML notations that are employed by the DSL have to be easy to understand and use by the domain expert. Nonetheless, one of the primary preconditions in creating a UML-based DSL is *the familiarity of the DSL designer with the concepts of the application domain*, such that she/he understands what the domain expert expects from such a language.

A second observation is that the *experience of the DSL designer in developing DSLs* plays an important factor. For instance, after developing the TACO and MICAS DSLs, we have noticed that several design decisions could have been taken differently, in order to improve the usability of the DSLs. If, in case of a profile-based DSL like TACO, additions and modifications of the language definition can be performed relatively easy, in case of a metamodel-based DSL like MICAS, the re-engineering of the tool support would require substantial effort.

Profiling is considered to be a relatively easy mechanism to rapidly create new DSLs. Although this is true to some extent, using profiles necessitates *comprehensive understanding of the UML metamodel and of the OCL constraint specification*. This affirmation is also valid for metamodeling where *comprehensive knowledge of UML and OCL is needed when defining MOF-based metamodels*.

Tool support plays an important role in deciding whether a profile or a metamodel is to be built. Ideally, UML profiles should be able to fully benefit from the support provided by UML tools, without requiring additional tool customization. However, as we discussed in Chapter 5, the capabilities of different tools in providing support for UML profiles vary from one tool to another. This fact has also been claimed by other researchers [54]. By tool support we understand not only graphical capabilities for displaying the properties of the customized UML elements, but also for ensuring the consistency of the models (using OCL constraints) with respect to the application domain modeled. Having all these conditions met, *profiling may be seen as a fast and low effort alternative in designing graphical DSLs*, following the UML notation. In the case of metamodels, additional effort is required for implementing the metamodel definition in the used tool. However, once this task is accomplished, *a completely UML-independent DSL is obtained by using metamodeling*. However, modifying the initial DSL definition is clearly more difficult in case of a metamodel, since it might involve additional work or even a complete re-implementation of the metamodel in the tool.

Another aspect that is worth discussing is the way in which the consistency of the models is enforced. Using profiles, the abstract syntax of UML is (with proper tool support) automatically inherited and thus, some of the architectural constraints of the DSL may be enforced based on the UML metamodel definition.

The downside of this approach is that the mapping between the domain concepts and the UML elements might not be a natural one in all situations. Moreover, constraints of the DSL may be specified in addition to the UML abstract syntax and well-formedness rules. In case of a metamodel, the elements of the new DSL can be used completely independent of the UML metamodel definition. Thus, the resulting language is supposed to be more comfortable for the domain expert. The metamodel definition allows one to encode most of the well-formedness rules in the abstract syntax of the DSL. Also, the tool support can be fully customized to address the specific needs of the domain expert.

Another remark is based on the observation that there is not a unique way in which profiles and metamodels can be used to define DSLs. This might pose a danger in the fact that the same application domain will be described using several different notations and might create misunderstandings and difficulties of usage, for those domain experts not familiar with UML and modeling.

As a final note, we consider that the following aspect needs to be taken seriously into consideration. The main concern is whether the UML usage should be restricted to the software domain only, and in particular, to the object-oriented software development or it can also be used as a modeling tool for all kinds of application areas. In the latter case, we might face the risk of defining a very large number of more or less natural DSLs, which are incompatible with each other. In our opinion, this aspect requires further investigation.

8.2 Limitations and Future Work

The work presented in this thesis is subject to several limitations, which we underline in the following.

We have proposed a model driven development methodology for programmable architectures. The application specification phase (discussed in Chapter 3 and Chapter 4) may be seen as independent of the characteristics of the target programmable architecture. In contrast, the two solutions proposed for the architecture specification phase (Chapter 5 and Chapter 6) have been influenced by the characteristics of the two architectures discussed. As we have advocated in the beginning of this thesis, we consider that in the area of programmable architectures there is no “one size fits all” solution; instead, each specific architecture should be accompanied by its own development methodology that exploits at maximum its characteristics. However, we consider that by employing similar model driven techniques as the ones discussed in this thesis, custom development methodologies may be devised for other types of programmable architectures.

In the application specification phase (Chapter 3 and Chapter 4), the non-functional requirements of the application are not considered. This leaves the introduction of the performance requirements, in the specification of the system, only for the later phases of the development cycle. We intend to address this issue in the future work. Another limitation is that no timing or concurrency aspects of the application are modeled during the application specification phase. In addition, the specification methodology discussed in Chapter 3 is limited by the lack of system data classification. This may be avoided by determining the data (i.e., attributes) that each object of the system encapsulates. This issue is also subject to future work.

Regarding the methodology presented in Chapter 4, a more comprehensive investigation has to be carried out, with respect to how the internal behavior of data processes is transformed into specifications of the identified class/object methods.

In the architecture specification phase, the work presented in Chapter 5 is subject to several limitations, too. On the one hand, the system-level estimation support provided by the TACO Profile does not take into account the MATLAB estimates for all resources, therefore the estimation results for different TACO configurations are currently less accurate. Beside addressing this limitation, future work might look into an approach for enabling the integration of the performance estimates and physical characteristics of the components, within the TACO programming model. Such an approach would allow us to evaluate the impact of choosing a given functional primitive on a given configuration during the mapping process. Used in combination with an application specification process that would incorporate non-functional requirements, the approach would enable us to include in the mapping process both the application's non-functional requirements and the physical characteristics of the architecture.

The mapping process discussed in Chapter 5 is basically focused towards the characteristics of the TACO architecture. In our opinion, the main limitation here is that the process does not take into account the mapping of the data types between the specifications of the application and of the architecture. This limitation could be removed once the application analysis and the architecture specification phases are enhanced with a classification of the data-types in the system. The approach may enable us to narrow the implementation gap even more, and may improve the automation support.

In the end, we emphasize again that this thesis has addressed the development of programmable architectures, from requirements to the design space exploration phase, and a systematic approach has been proposed for each step of the development cycle. Some of these steps could be automated, others require additional work, although the prerequisites for automation have been provided, while other steps are performed manually. Consequently, further work has to be directed towards perfecting and integrating the steps of the proposed methodology.

Bibliography

[1] Domain Specific Modeling Forum web page. Online at `http://www.dsmforum.org/tools.html`. Last checked 20-02-2007.

[2] `http://www.abo.fi/~dtruscan/ipv6index.html`. Last checked 20-02-2007.

[3] The Software Modeling Workbench (SMW) toolkit. Available at `http://mde.abo.fi/confluence/display/MDE/Tools`. Last checked 20-02-2007.

[4] R. J. Abbott. Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894, 1983.

[5] T. Ahonen, S. Virtanen, J. Kylliäinen, D. Truscan, T. Kasanko, D. Sigüenza-Tortosa, T. Ristimäki, J. Paakkulainen, T. Nurmi, I. Saastamoinen, H. Isännäinen, J. Lilius, J. Nurmi, and J. Isoaho. *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 16. A Brunch from the Coffee Table: Case Study in NOC Platform Design. Kluwer Academic Publishers, April 2004.

[6] B. Alabiso. Transformation of Data Flow Analysis Models to Object Oriented Design. In *OOPSLA '88*, pages 335–53. ACM Press, 1988.

[7] J. Ali and J. Tanaka. Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. *Journal of Computer Science and Information Management*, 2(1):24–36, 2001.

[8] S. W. Ambler. What's Missing from the UML? *SIGS Publications, Object Magazine*, Oct. 1997.

[9] S. W. Ambler. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, 2002.

[10] M. I. Anwar and S. Virtanen. Mapping the DVB Physical Layer onto SDR-enabled Protocol Processor Hardware. In *Proceedings of the 23rd IEEE Norchip Conference*. IEEE Circuits and Systems Society, November 2005.

[11] C. Arpnikanondt and V. K. Madisetti. Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach. Technical Report YES-TR-99-01, Georgia Tech., Atlanta, Dec. 1999.

[12] C. Atkinson and T. Kühne. Rearchitecting the UML Infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12:209–321, 2002.

[13] C. Atkinson, T. Kühne, and B. Henderson-Sellers. Systematic Stereotype Usage. *Software and Systems Modeling*, 2:153–163, 2003.

[14] M. Awad, J. Kuusela, and J. Ziegler. *Octopus: Object-Oriented Technology for Real-Time Systems*. Prentice Hall, 1996.

[15] R. J. Back, L. Petre, and I. Porres. Analysing UML use cases as contracts. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.

[16] F. Balarin, L. Lavagno, C. Passerone, and Y. Watanabe. Processes, Interfaces and Platforms. Embedded Software Modeling in Metropolis. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software, Second International Conference, EMSOFT 2002*, volume 2491 of *Lecture Notes in Computer Science*, pages 407–21. Springer-Verlag, Oct. 2002.

[17] J. P. Barros and L. Gomes. From Activity Diagrams to Class Diagrams. In *≪UML≫ 2000 WORKSHOP Dynamic Behaviour in UML Models: Semantic Questions*, Oct. 2000.

[18] L. B. Becker, C. E. Pereira, O. P. Dias, I. M. Teixeira, and J. P. Teixeira. MOSYS: A Methodology for Automatic Object Identification from System Specification. In *3rd IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing*, pages 198–201. IEEE CS Press, Mar. 2000.

[19] S. Berner, M. Glinz, and S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages. In *≪UML≫ 1999*, pages 249–64, 1999.

[20] D. Björklund. *A Kernel Language for Unified Code Synthesis*. PhD thesis, Åbo Akademi University, Turku, Finland, Feb. 2005. ISBN 952-12-1478-3.

[21] M. Boasson. Embedded Systems Unsuitable for Object Orientation. In *Ada-Europe*, pages 1–12, 2002.

[22] G. Booch. *Object-Oriented Analysis and Design*. The Benjamin/Cummings Publishing, 2nd edition, 1994.

[23] G. Booch. *Best of Booch: Designing Strategies for Object Technology*. SIGS Books, Dec. 1997.

[24] G. Booch. UML in Action. *Communications of the ACM*, 42(10):26–28, 1999.

[25] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley Longman, 1999.

[26] T. Bruckhaus, N. H. Madhavji, I. Janssen, and J. Henshaw. The Impact of Tools on Software Productivity. *IEEE Software*, 13(5):29–38, 1996.

[27] J. Cabot and C. Gomez. A Simple Yet Useful Approach to Implementing UML Profiles in Current CASE Tools. In *Workshop in Software Model Engineering (WiSME), UML'2003 Conference*, pages 297–310, 2003.

[28] R. Chen, M. Sgroi, G. Martin, L. Lavagno, A. Sangiovanni-Vincentelli, and J. Rabaey. Embedded System Design Using UML and Platforms. In *Proceedings of Forum on Specification and Design Languages 2002 (FDL'02)*, Sept. 2002.

[29] R. Chen, M. Sgroi, G. Martin, L. Lavagno, A. Sangiovanni-Vincentelli, and J. Rabaey. *UML for Real*, chapter UML and Platform-based Design, pages 107–126. Kluwer Academic Publishers, 2003.

[30] R. Chen, M. Sgroi, G. Martin, L. Lavagno, A. L. Sangiovanni-Vincentelli, and J. Rabaey. *System Specification and Design Languages*, chapter Embedded System Design Using UML and Platforms. CHDL. Kluwer, 2003.

[31] H. Choi, S. H. Hwang, C.-M. Kyung, and I.-C. Park. Synthesis of application specific instructions for embedded DSP software. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 665–671. ACM Press, 1998.

[32] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA*. J. Wiley and Sons Ltd., West Sussex, England, 1998.

[33] G. de Jong. A UML-Based Design Methodology for Real-Time and Embedded Sytems. *2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, page 0776, 2002.

[34] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, Dec. 1998.

[35] J.-L. Dekeyser, P. Marquet, S. Meftali, C. Dumoulin, P. Boulet, and S. Niar. Why to do without Model Driven Architecture in embedded system codesign? In *The first annual IEEE BENELUXDSP Valley Signal Processing Symposium*, Antwerp, Belgium, Apr. 2005.

[36] T. DeMarco. *Structural Analysis and System Specification*. Yourdon Press, Englewood Cliffs, New Jersey, 1978.

[37] O. Dieste, M. Genero, N. Juristo, J. Maté, and A. Moreno. A Conceptual Model Completely Independent of the Implementation Paradigm. *The Journal of Systems and Software*, 68(3):183–198, 2003.

[38] Digital Equipment Corp., Intel Corp., and Xerox Corp. The Ethernet - A Local Area Network, ver. 1.0 , Sept. 1980.

[39] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Object, Frameworks and Patterns*. Addison-Wesley, 1999.

[40] B. P. Douglass. *Real Time UML: Advances in the UML for Real-Time Systems, 3rd Edition*. Object Technology. Addison-Wesley, 3rd edition, 2004.

[41] D. D'Souza, A. Sane, and A. Birchenough. First-Class Extensibility for UML-Profiles, Stereotypes, Patterns. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 265–277. Springer, 1999.

[42] C. Dumoulin, P. Boulet, J.-L. Dekeyser, and P. Marquet. MDA for SoC design, intensive signal processing experiment. In *Forum on Specification and Design Languages (FDL'03)*. ECSI, Sept. 2003.

[43] M. Edwards and P. Green. The Modelling of Embedded Systems Using HASoC. *2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, page 0752, 2002.

[44] G. Engels, R. Heckel, and S. Sauer. UML — A Universal Modeling Language? In *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 24–38. Springer-Verlag, June 2000.

[45] J. M. Fernandes. Functional and Object-Oriented Modeling of Embedded Software. Technical Report 512, Turku Centre for Computer Science (TUCS), Turku, Finland, 2003.

[46] J. M. Fernandes and J. Lilius. Functional and Object-Oriented Modeling of Embedded Software. In *Proceedings of the Intl. Conf. on the Engineering of Computer Based Systems (ECBS'04)*, Brno, Czech Rep., May 2004.

[47] J. M. Fernandes and R. J. Machado. From Use Cases to Objects: An Industrial Information Systems Case Study Analysis. In *7th International Conference on Object-Oriented Information Systems (OOIS '01)*, pages 319–28. Springer-Verlag, Aug. 2001.

[48] J. M. Fernandes, R. J. Machado, and H. D. Santos. Modelling Industrial Embedded Systems with UML. In *Proceedings of CODES 2000*, pages 18–22. ACM Press, May 2000.

[49] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and Z. Shuqing. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[50] H. Gall and R. Klösch. Finding Objects in Procedural Programs: An Alternative Approach. In *2nd Working Conference on Reverse Engineering*, pages 208–16. IEEE CS Press, July 1995.

[51] J. Gerlach and W. Rosenstiel. System Level Design Using the SystemC Modeling Platform. In *Proceedings of the 3rd Workshop on System Design Automation*. IEEE Press, Mar. 2000.

[52] R. L. Glass. The Naturalness of Object Orientation: Beating a Dead Horse? *IEEE Software*, 19(3):103–4, 2002.

[53] A. D. Gloria and P. Faraboschi. An evaluation system for application specific architectures. In *MICRO 23: Proceedings of the 23rd annual workshop and symposium on Microprogramming and microarchitecture*, pages 80–89. IEEE Press, 1990.

[54] M. Gogolla and B. Henderson-Sellers. Analysis of UML Stereotypes within the UML Metamodel. In *≪UML≫ 2002*, pages 84–99, Sept. 2002.

[55] B. Graph, M. Lormans, and H. Toetenel. Embedded Software Engineering: The State of the Practice. *IEEE Software*, 20(6):61–69, Nov/Dec 2003.

[56] D. J. Hatley and I. A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing Co., New York, USA, 1987.

[57] T. Henriksson. Hardware Architecture for Protocol Processing. Licentiate thesis, no. 911, Departament of Electrical Engineering, Linköping University, Sweden, Dec. 2001.

[58] T. Henriksson, U. Nordqvist, and D. Liu. Specification of a Configurable General-Purpose Protocol Processor. In *Intl. Symp. on Communication Systems, Networks And Digital Signal Processing (CSNDSP 2000)*, pages 284–289, July 2000.

[59] I. M. Holland and K. J. Lieberherr. Object-Oriented Design. *ACM Computing Surveys*, 28(1):273–5, 1996.

[60] Intel Press. MicroACE design document, rev 1.0, 2001.

[61] J. Isaksson, J. Lilius, and D. Truscan. A MOF-Based Metamodel for SA/RT. In N. Guelfi, editor, *Proceedings of the Rapid Integration of Software Engineering Techniques (RISE'04) workshop, November 2004, Luxembourg, revised selected papers.*, volume 3475 of *Lecture Notes in Computer Science*, pages 97–106. Springer-Verlag, 2005.

[62] ITU-T. Specification and Description Language (SDL): ITU-T Recommendation Z.100. Available online at `http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf`, 1996. Last checked 20-02-2007.

[63] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, June 1992.

[64] I. Jacobson. Basic Use Case Modeling (Continued). *Report on Object Analysis and Design*, 1(3):7–9, 1994.

[65] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[66] I. Jacobson and F. Lindström. Reengineering of Old Systems to an Object-Oriented Architecture. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 340–50. ACM Press, 1991.

[67] P. Jalote. Functional Refinement and Nested Objects for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 15(3):264–270, 1989.

[68] J. Janssen. *Compiler Strategies for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, Sept. 2001.

[69] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. *Computer Networks*, 38(3):277–293, 2002.

[70] S. Kent. Model Driven Engineering. In *Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[71] K. Keutzer. Programmable platforms will rule. EETimes, On line at `http://www.eetimes.com/story/OEG20020911S0063`, Sept. 2002. Last checked 22-02-2007.

[72] K. Keutzer and N. Shah. A Survey of Programmable Platforms - Network Procs. On line at `http://www.eecs.berkeley.edu/~mihal/ee244/lectures/soc-prog-plat-np.pdf`. Last checked 20-02-2007.

[73] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An Approach for Quantitative Analysis of Application Specific Dataflow architectures. In *Application-specific Systems, Architectures and Processors (ASAP'97)*, July 1997.

[74] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers. A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach. *Lecture Notes in Computer Science*, 2268:18, Jan. 2002.

[75] J. Kim and F. J. Lerch. Towards a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies. In *Conference on Human Factors in Computing Systems (CHI '92)*, pages 489–98. ACM Press, May 1992.

[76] P. Kukkala, J. Riihimäki, M. Hannikainen, T. D. Hämälainen, and K. Kronlöf. UML 2.0 Profile for Embedded System Design. In *Design, Automation and Test in Europe (DATE'05) Volume 2*, pages 710–715, 2005.

[77] J. Kuusela. Object Oriented Development of Embedded Systems with the Octopus Method. In *Lectures on Embedded Systems, European Educational Forum School on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 304–15. Springer-Verlag, 1998.

[78] L. Kuzniarz and M. Staron. On Practical Usage of Stereotypes in UML-Based Software Development. In *Forum on Design and Specification Languages (FDL'02)*, pages 262–70, 2002.

[79] C. Labovitz, G. R. Malan, and F. Jahanian. Internet Routing Instability. In *Proc. ACM SIGCOMM '97*, Cannes, France, Sept. 1997.

[80] T. Lillqvist. Subgraph Matching in Model Driven Engineering. Master's thesis, Turku Centre for Computer Science, Åbo Akademi University, Mar. 2006.

[81] R. J. Machado, J. M. Fernandes, P. Monteiro, and H. Rodrigues. Transformation of UML Models for Service-Oriented Software Architectures. In *Proceedings of 12th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2005)*, pages 173–82, Apr. 2005.

[82] G. Malkin and R. Minnear. *RIPng for IPv6*. Internet Engineering Task Force, Jan. 1997. RFC 2080.

[83] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: A Merger of Real-Time UML and Co-Design. In *9th ACM/IEEE/IFIP International Symposium on Hardware/Software Codesign (CODES '01)*, pages 23–8. ACM Press, Apr. 2001.

[84] C. Matsumoto. Net Processors Face Programming Trade-Offs. EETimes – iApplianceWeb, Sept. 2002. Available at `http://www.iapplianceweb.com/story/OEG20020922S0003.htm`. Last checked 20-02-2007.

[85] C. B. Matthias Gries, Scott Weber. The Mescal Architecture Development System (Tipi) Tutorial. Technical Report UCB/ERL M03/40, Electronics Research Lab, University of California at Berkeley, Oct. 2003.

[86] A. Mihal and K. Keutzer. *Mapping Concurrent Applications onto Architectural Platforms*, chapter 3, pages 39–59. Kluwer Academic Publishers, 2003.

[87] A. Mihal and K. Keutzer. A Processing Element and Programming Methodology for Click Elements. In *Workshop on Application Specific Processors (WASP 2005)*, pages 10–17, Sept. 2005.

[88] G. Moretti. The search for the perfect language. *EDN*, Feb. 2004. Available at `http://www.edn.com/article/CA376625.html`. Last checked 20-02-2007.

[89] T. Nurmi, S. Virtanen, J. Isoaho, and H. Tenhunen. Physical modeling and system level performance characterization of a protocol processor architecture. In *Proceedings of the 18th IEEE Norchip Conference*, pages 294–301, Turku, Finland, Nov. 2000.

[90] Object Management Group. `http://www.omg.org/`.

[91] OMG. XML Metadata Interchange (XMI) Specification. OMG Document formal/03-05-01. Available at `http://www.omg.org/cgi-bin/doc?formal/03-05-01`.

[92] OMG. Object Constraint Language (OCL) Specification, ver. 1.1, Sept. 1997. `http://www.omg.org/`.

[93] OMG. Model Driven Architecture, July 2001. Document ormsc/2001-07-01, available at `http://www.omg.org/mda`.

[94] OMG. The Unified Modeling Language specification, version 1.4, Sept. 2001. Document formal/05-04-01, Available at `http://www.omg.org/docs/formal/05-04-01.pdf`.

[95] OMG. Meta Object Facility, version 1.4, Apr. 2002. Document formal/2002-04-03, available at `http://www.omg.org/cgi-bin/doc?formal/02-04-03`.

[96] OMG. MDA Guide Version 1.0.1, June 2003. Document omg/2003-06-0. Available at `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[97] OMG. A UML Profile for SoC, Draft RFC Submission to OMG (Realtime/2005-04-12). Apr. 2005.

[98] OMG. UML Profile for Schedulability, Performance, and Time Specification Version 1.1, Jan. 2005. OMG Document formal/05-01-02.

[99] Open Core Protocol International Partnership (OCP-IP). Open Core Protocol (OCP) specification ver. 2.0. Available at `http://www.ocp-ip.org`. Last checked 20-02-2007.

[100] Open SystemC Initiative. *SystemC Specification*. `http://www.systemc.org`.

[101] A. Pnueli. Embedded Systems: Challenges in Specification and Verification. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software, Second International Conference, EMSOFT 2002*, volume 2491 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, Oct. 2002.

[102] I. Porres. A Toolkit for Manipulating UML Models. *Software and Systems Modeling, Springer-Verlag*, 2(4):262–277, Dec. 2003.

[103] I. Porres and M. Alanen. Generic Deep Copy Algorithm for MOF-Based Models. In A. Rensink, editor, *Model Driven Architecture*, pages 49–60. University of Twente, July 2003.

[104] B. Purves, A. Lyons, F. Leymann, B. Schätz, and J. Rozenblit. Panel Discussion - ”Will MDA really work?” Status and Perspectives of Model Based Development in the Engineering Context. In *13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS’2006)*, Mar. 2006. Potsdam, Germany.

[105] M. Radetzki and W. Nebel. Synthesizing Hardware from Object-Oriented Descriptions. In *Proceedings of the 2nd Forum on Design Languages (FDL’99)*, Lyon, France, Aug. 1999.

[106] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. In *Design, Automation and Test Europe (DATE’2002)*, volume 02, pages 704–709. IEEE Computer Society, 2005.

[107] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Object Technology. Addison-Wesley, 1999.

[108] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[109] K. Sandström. Microcode Architecture For A System On A Chip (SoC). Nokia Corporation Patent NC37341, 872.0167.P1(US) (Filing Date: 07.10.2003), Oct. 2002.

[110] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-based development. Technical Report TUM-10204, Institut für Informatik, Technische Universität München, May 2002.

[111] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development of Embedded Systems. Number 2426 in Lecture Notes in Computer Science, pages 298–311. Springer, 2002.

[112] A. Schleicher and B. Westfechtel. Beyond stereotyping: Metamodeling approaches for the UML. In R. H. Sprague, Jr., editor, *Proceedings of 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society, 2001.

[113] B. Selic. Turning Clockwise: Using UML in the Real-Time Domain. *Communications of the ACM*, 42(10):46–54, 1999.

[114] B. Selic. A Generic Framework for Modeling Resources With UML. *IEEE Computer*, 33(6):64–9, June 2000.

[115] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, New York, 1994.

[116] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Developement. *IEEE Software*, 20(5):42–45, Sep./Oct. 2003.

[117] N. Shah and K. Keutzer. Network Processors: Origin of Species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, Oct. 2002.

[118] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. NP-Click: A Productive Software Development Approach for Network Processors. *IEEE Micro*, 24(5):45–54, Sept. 2004.

[119] P. Shoval and J. Kabeli. FOOM: Functional- and Object-Oriented Analysis & Design of Information Systems — An Integrated Methodology. *Journal of Database Management*, 12(1):15–25, Jan. 2001.

[120] S. Sigfried. *Understanding Object-Oriented Software Engineering*. IEEE Press, 1996.

[121] E. Stambouly. Modeling Awareness: Assessing an Organization's Readiness for Use of Modeling in Software Development. Published online `http://www.cephas.cc/publish/Modeling_Awareness.pdf`, 2002. Last checked 20-02-2007.

[122] M. Staron and L. Kuzniarz. Guidelines for creating "good" stereotypes. In K. Koskimies, L. Kuzniarz, J. Nummenmaa, and Z. Zhang, editors, *3rd Nordic Workshop on UML and Software Modeling (NWUML2005)*, pages 1–17, Aug. 2005.

[123] N. A. Tariq and N. Akhter. Comparison of Model Driven Architecture (MDA) based tools. Master's thesis, Engineering and Management of Information Systems at Department of Computer and System Sciences (DSV), Royal Institute of Technology (KTH), Stockholm, Sweden, June 2005.

[124] Teja Technologies, Inc. *Network Processor Design: Issues and Practices*, volume 2, chapter Teja C: A C-based programming language for multiprocessor architectures, page 464. Morgan Kaufmann Publishers, Nov. 2003.

[125] The Middleware Company. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach: Productivity Analysis, 2003. `http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf`.

[126] T. Tran, K. Khan, and Y.-C. Lan. A Framework for Transforming Artifacts from Data Flow Diagrams to UML. In M. Hamza, editor, *The IASTED Conference on Software Engineering (SE2004)*, volume 418. Acta Press, Feb. 2004.

[127] D. Truscan, J. M. Fernandes, and J. Lilius. Tool Support for DFD to UML Model-based Transformations. Technical Report 519, TUCS, Turku, Finland, Apr. 2003.

[128] D. Truscan, S. Virtanen, and J. Lilius. SystemC Simulation Framework for Protocol Processing Architectures. In *Proceedings of International Symposium on Signals, Circuits and Systems (SCS 2003)*, July 2003.

[129] G. van Rossum. The Python Programming Language. Available at `http://www.python.org`.

[130] I. Vessey and S. A. Conger. Requirements Specification: Learning Object, Process, and Data Methodologies. *Communications of the ACM*, 37(5):102–13, 1994.

[131] A. S. Vincentelli. Defining Platform-based Design. *EEDesign of EETimes*, Feb. 2002.

[132] S. Virtanen. *A Framework for Rapid Design and Evaluation of Protocol Processors*. PhD thesis, University of Turku, Turku, Finland, September 2004.

[133] S. Virtanen, J. Lilius, T. Nurmi, and T. Westerlund. TACO: Rapid Design Space Exploration for Protocol Processors. In *the Ninth IEEE/DATC Electronic Design Processes Workshop Notes*, Monterey, CA, USA, Apr. 2002.

[134] S. Virtanen, D. Truscan, and J. Lilius. TACO IPv6 Router - A Case Study in Protocol Processor Design. Technical Report 528, Turku Centre for Computer Science, Turku, Finland, April 2003.

[135] S. Virtanen, D. Truscan, J. Paakkulainen, J. Isoaho, and J. Lilius. Highly Automated FPGA Synthesis of Application-Specific Protocol Processors. In *15th International Conference on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, Aug. 2005.

[136] P. T. Ward and S. J. Mellor. *Structured Development for Real-Time Systems*. Prentice Hall/Yourdon Press, 1985. Published in 3 volumes.

[137] D. Wolber. Reviving Functional Decomposition in Object-Oriented Design. *Journal of Object-Oriented Programming*, 10(6):31–38, 1997.

[138] L. Ying. From use cases to classes: a way of building object model with UML. *Information and Software Technology*, 45(2):83–93, Feb. 2003.

[139] V. D. Zivkovic and P. Lieverse. An overview of methodologies and tools in the field of system level design. In E. F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges: 2nd International Samos Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, number 2268 in Lecture Notes in Computer Science, pages 74–88. Springer-Verlag, 2002.

Acronyms

ADL	Architecture Description Language
ASIC	Application Specific Integrated Circuits
API	Application Programming Interface
AVI	Audio Video Interleave
CASE	Computer Added Software Engineering
CIM	Computational Independent Model
DFD	Data-Flow Diagram
DAVR	Digital Video/Audio Recording Application
DSL	Domain Specific Language
FU	Functional Unit
FP	Functional Primitive
GPP	General Purpose Processor
HDL	Hardware Description Language
IN	Interconnection Network (TACO)
IP	Intellectual Property
IPv6	Internet Protocol version 6
INC	Interconnection Network Controller (TACO)
LHS	Left-hand Side
MBD	Model Based Development
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta-Object Facility
MPEG	Moving Picture Expert Group
MICAS	Microcode Architecture For A System On A Chip (SoC)
MoC	Model of Computation
OMG	Object Management Group
OCL	Object Constraint Language
PE	Property Editor
PIM	Platform Independent Model
PM	Platform Model
PSM	Platform Specific Model
RIPng	Routing Information Protocol next generation
RHS	Right-hand Side
RTOS	Real-time Operating System
TACO	Tools for Application-specific hardware/software CO-design
TTA	Transport Trigger Architecture
UML	Unified Modeling Language
SA/RT	Structured Analysis for Real-Time systems
SDK	Software Development Kit
SoC	System on Chip

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics

Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-1858-3
ISSN 1239-1883